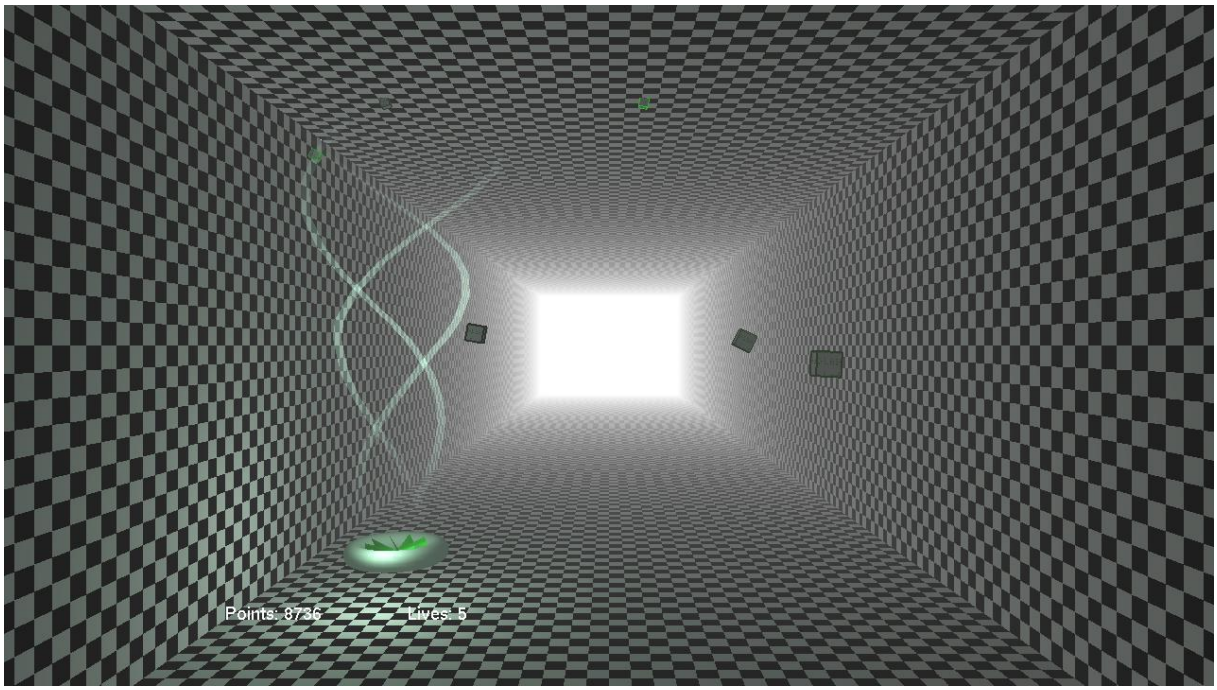
C++-Programmierung für Computergraphik – Übungsblatt 5

Übung 5
Ausgabe des Übungsblatts am: 25.06.2014
Abnahme der Lösung am: 16.07.2014

Auf diese Übungsblatt können insgesamt **20 Punkte** erworben werden. Die Teilaufgabe 5.7 erlaubt es Ihnen Bonuspunkte zu holen, mit denen Sie Punktabzug in den anderen Aufgaben ausgleichen können.

NICHT ERSCHRECKEN!

Das Aufgabenblatt ist sehr lang; es gibt viel zu lesen. Das liegt daran, dass wir wirklich alles erklären und beschreiben wollten was Sie brauchen. Die wirkliche Arbeit, die Sie zu erledigen haben ist absolut im Rahmen der Übung machbar.



Präamble

ACHTUNG: Dieses Aufgabenblatt lässt ihnen zur Lösung einige Freiheiten! Lesen Sie sich zunächst das komplette Aufgabenblatt durch, damit Sie wissen wo sie entsprechenden Spielraum haben.

Allgemeine Hinweise: Achten Sie darauf, die Quellcode-Dateien in den richtigen Ordnern abzulegen. Alle Header-Dateien (`*.hpp`) sollen je nach Funktion in `include/flappy_box/{model, controller, view}` abgelegt werden, alle Source-Dateien (`*.cpp`) in `src/flappy_box/{model, controller, view}`. Achten Sie ebenfalls auf die Verwendung der korrekten Namespaces `::flappy_box::{model, controller, view}`.

Das Ziel dieser letzten Übungsaufgabe ist die Realisierung eines Minispiels, bei dem (futuristische) Kisten mit Hilfe einer Art Luftstrom balanciert werden sollen. Spielziel ist, möglichst viele Kisten kollidieren und möglichst wenige herunterfallen zu lassen.

Klassen-Überblick – die Spiele-Entitäten

Im Folgenden werden die Klassen vorgestellt, die am Spiel beteiligt sind (die Klassennamen in der Reihenfolge Model, Controller, View):

Box, BoxObjectLogic, BoxGlDrawable/BoxAlAudible:

- Spielobjekt, Bewegungsberechnung physikalisch motiviert, als texturierte Kiste visualisiert, erzeugt Töne bei Erzeugung und Verschwinden

Paddle, PaddleLogic, PaddleGlDrawable/PaddleAlAudible:

- steuerbares Spielerobjekt, visualisiert durch Ventilator, kann Kraft auf Kisten ausüben, erzeugt Töne bei Bewegung

World, WorldLogic, WorldGlDrawable/WorldAlAudible:

- repräsentiert Spielwelt mit allgemeinen Eigenschaften wie Größe, Spielerpunkte und Spielerleben, kontrolliert die Interaktion zwischen Spielobjekten, startet das Spiel, visualisiert durch umgebenden Tunnel und Anzeige der Spielerpunkte/-leben, auralisiert durch Hintergrund-Musik

GameOver, GameOverLogic, GameOverGlDrawable/GameOverAlAudible:

- dient zur Anzeige des Spielverlusts und der erreichten Gesamtpunktzahl, auralisiert durch einmaligen Sound

Aufgabe 5.1 – Implementierung der Modell-Klassen (2 Punkte)

Zunächst sollen die Modellklassen implementiert werden, um innere Zustände der beteiligten Spielobjekte zu speichern. Vorgegeben ist die Klasse Box, an der Sie sich bei der Bearbeitung der Aufgaben orientieren können. Achten Sie auf den Namespace `::flappy_box::model`!

Aufgabe 5.1.1 – Die Klasse Paddle : public ::model::GameObject

Folgende (private) Membervariablen werden benötigt (alle vom Typ vec3_type):

- _position (beschreibt die momentane Position des Objekts)
- _maxPosition (beschreibt die maximal mögliche Position des Objekts im ersten Oktanten)
- _velocity (die aktuelle Geschwindigkeit)
- _acceleration (die aktuelle Beschleunigung)
- _size (die Größe in x-, y- und z-Richtung)
- _playerControl (beschreibt den Bewegungseinfluss durch den Nutzer)

Im Konstruktor soll die Klasse Paddle analog der Klasse Box einen `const std::string&` als Objektnamen übergeben bekommen. Initialisieren Sie die Members mit sinnvollen Werten ([0,0,0] bzw. [1,1,1]).

Schreiben Sie für alle Membervariablen die passenden Get- und Set-Methoden (vgl. `box.hpp`).

Aufgabe 5.1.2 – Die Klasse World : public ::model::GameObject

Legen Sie die beiden Dateien world.hpp und world.cpp in den entsprechenden Ordnern an (bzw. kopieren Sie box.hpp/cpp und ändern Sie ab).

Folgende (private) Membervariablen werden benötigt (alle vom Typ int):

- _playerPoints (halten den aktuellen Punktestand)
- _remainingLives (halten die verbleibenden "Leben" des Spielers)

Im Konstruktor soll analog der Klasse Box ein `const std::string&` als Objektname übergeben werden. Initialisieren Sie die Members mit sinnvollen Werten.

Schreiben Sie für alle Membervariablen die passenden Get- und Set-Methoden (vgl. `box.hpp`).

Außerdem werden zwei dedizierte Memberfunktionen benötigt, welche die Größe der Spielwelt wiedergeben:

```
double getWorldHalfHeight() const { return 30.0; }; /* gibt die halbe Ausdehnung der
Spielwelt in z-Richtung zurück */
double getWorldHalfWidth()  const { return 42.0; }; /* gibt die halbe Ausdehnung der
Spielwelt in x-Richtung zurück */
```

Anmerkung: Es sollen halbe Ausdehnungen verwendet werden, da die Spielwelt symmetrisch in allen Richtungen aufgebaut ist (mit dem Ursprung in der Mitte des Bildschirms).

Aufgabe 5.1.3 – Die Klasse GameOver : public ::mode::GameObject

Legen Sie die beiden Dateien game_over.hpp und game_over.cpp in den entsprechenden Ordnern an (bzw. kopieren Sie box.hpp/cpp und ändern Sie ab).

Folgende (private) Membervariable vom Typ const int wird benötigt:

- _playerPoints (repräsentiert den (unveränderbaren) Gesamtpunktestand)

Im Konstruktor soll neben einem const std::string& als Objektname ein const int übergeben werden, der als Initialisierungswert für _playerPoints dient.

Schreiben Sie für _playerPoints eine passende Get-Methode. Warum würde eine Set-Methode fehlschlagen?

Aufgabe 5.2 – Implementierung der Controller-Klassen (9 Punkte)

In dieser Aufgabe sollen die Controllerklassen implementiert werden, welche die inneren Zustände der beteiligten Spielobjekte ändern können. Hierzu steht die Membervariable `_model` als Shared Pointer zur Verfügung, mit der die Logic-Klasse auf das im Konstruktor zugewiesene Modell zugreifen kann. Zu Ihrer Orientierung ist die Klasse `BoxObjectLogic` vorgegeben. Achten Sie auf den Namespace `::flappy_box::controller`!

Aufgabe 5.2.1 – Vervollständigen Sie die advance-Methode in der Klasse BoxObjectLogic

Die `advance()`-Methode realisiert die Bewegungssimulation einer Box und orientiert sich dabei am physikalischen Vorbild, jedoch stark vereinfacht.

Zur Voranschreitung des Simulationsprozesses soll das explizite Euler-Verfahren implementiert werden, dass unter Beachtung von anliegenden externen Kräften die resultierenden Vektoren für Beschleunigung, Geschwindigkeit, Position etc. berechnet. Gehen Sie davon aus, dass die Membervariable `_model` aktuelle Werte enthält und setzen Sie sukzessive die neuen Modell-Werte mit Hilfe der alten Werte aus dem letzten Zeitschritt unter Anwendung folgender Gleichungen:

Gegeben:

- p_{alt} := Positionskoordinate des Box-Modells aus dem letzten Zeitschritt
- v_{alt} := Geschwindigkeitsvektor des Box-Modells aus dem letzten Zeitschritt
- a_{alt} := Beschleunigungsvektor des Box-Modells aus dem letzten Zeitschritt
- a_{grav} := konstanter Gravitationsvektor (es eignet sich ein Wert von (0, 0, -1.5))

- f_{ext} := Vektor der am Modell anliegenden externen Kräfte
- d := konst. skalarer Wert, der eine Dämpfung der aktuellen Beschleunigung bewirkt (z.B. 0.8)
- dt := Zeitschrittgröße als Berechnungsgrundlage für das Eulerschritt-Verfahren (in der Variable `timestep_sec` gespeichert)
- m := Masse des Würfels (der Einfachheit halber kann hierfür das Volumen der Box verwendet werden)
- a_{ext} := Beschleunigung resultierend aus anliegenden externen Kräften ($f_{ext} = m \cdot a_{ext}$)

Gesucht: $a_{neu}, v_{neu}, p_{neu}$

a_{neu} ergibt sich direkt als Summe aller Beschleunigungen: $a_{neu} := a_{alt} * d + a_{ext} + a_{grav}$

Die Bewegungsgleichungen werden mit Hilfe des expliziten Euler-Verfahrens wie folgt numerisch approximiert:

$$\begin{aligned} v_{neu} &= v_{alt} + dv \\ p_{neu} &= p_{alt} + dp \end{aligned} \quad \text{mit} \quad \begin{aligned} a &= \frac{dv}{dt} \rightarrow dv = a_{neu} \cdot dt \\ v &= \frac{dp}{dt} \rightarrow dp = v_{neu} \cdot dt \end{aligned}$$

Überprüfen Sie nach dem Berechnen der neuen Modell-Werte die Position und begrenzen Sie sie gegebenenfalls auf den Maximalbetrag `_model->maxPosition()`, falls größere Absolutwerte berechnet wurden (sodass die Box am "Weltrand" anstößt). Spiegeln Sie in diesen Fällen den Geschwindigkeitsvektor - je nach Fall entlang der x-Achse oder entlang der z-Achse. Multiplizieren Sie einen Velocity-Dämpfungswert an die gespiegelte Velocity-Komponente zur Simulation eines unelastischen Stoßes.

Setzen Sie anschließend die finalen Beschleunigungs-, Geschwindigkeits- und Positionsvektoren mittels der Set-Methoden des Modells.

Bonus-Teilaufgabe:

Implementieren Sie einen (einfachen) Mechanismus, der eine Rotationsbewegung entlang der y-Achse nach einem ähnlichem Prinzip wie in Aufgabe 5.2.1 realisiert. Die Klasse Box stellt hierfür die folgenden Member-Eigenschaften zur Verfügung:

```
double     _rotAcceleration; // only in y-direction
double         _rotVelocity; // only in y-direction
double               _angle;
```

Tipp:
Nutzen sie die Richtung des externen Kraftvektors (gegenüber der Paddle-Normalen) zur Bestimmung einer Links- bzw. Rechtsdrehung. Die Rotationsbeschleunigung ergibt sich wieder aus $F = m \cdot a$. Zur Verstärkung der Drehbewegung können Sie auch die Rotationsbeschleunigung durch Multiplikation mit einem konstanten Skalierungswert erhöhen.

Aufgabe 5.2.2 – Die Klasse PaddleLogic : public ::controller::Logic::ObjectLogic

Legen Sie die beiden Dateien paddle_logic.hpp und paddle_logic.cpp in den entsprechenden Ordnern an (bzw. kopieren Sie box_object_logic...).

Als einzige (private) Membervariable `_model` wird ein Shared Pointer auf ein `flappy_box::model::Paddle` benötigt. Sie wird durch ein im Konstruktor übergebenen `const std::shared_ptr< model::Paddle >&` initialisiert. Weiterhin muss die rein virtuelle Methode `advance` aus `::Logic::ObjectLogic` erneut deklariert und in der .cpp-Datei implementiert werden. Folgende Funktionalitäten sind hierbei gefordert:

Auswerten des Key-Events als Nutzereingabe:

Nutzen Sie die Felder `key` bzw. `special_key` sowie `key_state`, um das Drücken bzw. Loslassen einer Taste abzufangen. Beim Drücken soll ein normalisierter Wert in die `PlayerControl` des `_models` geschrieben werden. Das Spielerobjekt soll sich nur in x-Richtung bewegen können. Definieren Sie die Übergabe des Wertes 1.0 beim DRÜCKEN der *rechten Pfeiltaste*, -1.0 beim DRÜCKEN der *linken Pfeiltaste* sowie 0.0 beim LOSLASSEN einer Taste. Die Pfeiltasten werden als Special-Keys ausgelöst.

Setzen Sie diesen Wert per `_model->setPlayerControl()`.

Anschließend erfolgt eine Bewegungsberechnung wie in Aufgabe 5.2.1 mittels des expliziten Eulers. Als Beschleunigung a_{neu} dient `_model->playerControl()` mit einer geeigneten Skalierung (z.B. 1000). Die Geschwindigkeit v_{neu} wird wieder aus der alten Geschwindigkeit v_{alt} plus *dv* berechnet:

$$v_{neu} = v_{alt} + dv \text{ mit } dv = a_{neu} \cdot dt$$

Da die Bewegung des Paddle stark gedämpft sein soll, sobald der Nutzer die Taste loslässt ergibt sich v_{alt} aus der Multiplikation von `_model->velocity()` mit einem skalaren Dämpfungskoeffizient (z.B. 0.8). Begrenzen Sie außerdem den maximalen Geschwindigkeitsbetrag (z.B. auf 100.0). Damit das Paddle nicht die Spielwelt verlässt, muss dessen Position analog der Boxen auf den positiven bzw. negativen Maximalbetrag `_model->maxPosition()` begrenzt werden. Gehen Sie wie in Aufgabe 5.2.1 vor und setzen Sie anschließend die finalen Beschleunigungs-, Geschwindigkeits- und Positionsvektoren mittels der Set-Methoden des Modells.

Aufgabe 5.2.3 – Die Klasse WorldLogic : public ::controller::Logic::ObjectLogic

Legen Sie die beiden Dateien `world_logic.hpp` und `world_logic.cpp` in den entsprechenden Ordnern an (bzw. kopieren Sie box_object_logic...).

Neben der (privaten) Membervariablen `_model` vom Typ `std::shared_ptr< model::World >` (initialisiert durch ein im Konstruktor übergebenen `const std::shared_ptr< model::World >&`) wird eine boolesche Membervariable `_shallRestartTheGame` benötigt, die im Konstruktor mit true initialisiert wird. Sie wird im laufenden Spiel dazu benutzt, um anzugeben, ob das Spiel zurückgesetzt und neu gestartet werden soll.

Zusätzlich sind folgende private Hilfs-Memberfunktionen sinnvoll:

- `void addBoxToGame( ::controller::Logic& l )`, fügt dem Spiel eine neues Spielobjekt vom Typ `flappy_box::model::Box` hinzu
- `void setForce( std::shared_ptr< flappy_box::model::Box > & box, std::shared_ptr< flappy_box::model::Paddle > & paddle )`, realisiert die Interaktion zwischen Nutzer-Paddle und den Boxes
- `void restartGame( ::controller::Logic& l )`, setzt Spielobjekte zurück und startet das Spiel neu

Weiterhin muss die rein virtuelle Methode `advance()` aus `::Logic::ObjectLogic` erneut deklariert und in der .cpp-Datei implementiert werden. Sie setzt in diesem Falle die globale Spielelogik um. Folgender Ablauf ist hierbei gefordert:

1. Prüfen, ob neu gestartet werden soll, d.h. Auswerten von `_shallRestartTheGame`, im Falle von true die Hilfsfunktion `restartGame( l )` aufrufen

2. dem Spiel eine neue Kiste mittels `addBoxToGame( l )` anfügen. Dies soll in gleichabständigen Intervallen geschehen. Nutzen Sie die `std::chrono::steady_clock` zur Zeitmessung und definieren Sie ein konstantes Zeitfenster (z.B. als statische Variable innerhalb von `advance`()) sowie ein Laufvariable, um diese Funktionalität zu realisieren.
3. Die Spielerpunktelogik realisieren:
   a) Das Spielerpaddel in `game_model->objects()` finden (machen Sie sich mit `std::find_if` vertraut, als Suchprädikat dient der Objektname) und dynamisch auf `flappy_box::model::Paddle` casten
   b) Die Spielerpunkte inkrementieren (entweder auf Zeitbasis oder um festen Wert pro Aufruf von `advance`)
   c) Die Spielerleben prüfen, bei null Leben das Spielermodell invalidieren (`setAlive(false)`) und dem `l.game_model()` ein `game_over`-Objekt hinzufügen (Denken Sie daran, die aktuelle Punktzahl mit zu übergeben)
4. Interaktionslogik zwischen Spieler-Paddle und Boxes realisieren:
   Iterieren Sie durch alle `GameObjects` aus `l.game_model()->objects()` und casten Sie dynamisch auf `flappy_box::model::Box`. Sollte ein Box-Objekt BO existieren (und ebenso das Spieler-Paddle), soll zunächst die Hilfsmethode `setForce( box, player_paddle )` aufgerufen werden, welche die Kraftübertragung von Paddle zu Box berechnet. Anschließend wird geprüft, ob die z-Position von BO bereits kleiner als die z-Position des Paddles ist. Ist dies der Fall, so wird die Box invalidiert und die Spielerleben um eins dekrementiert.
5. Interaktionslogik zwischen den Boxes:
   Innerhalb derselben Schleife (d.h. nach dem Cast auf `flappy_box::model::Box`) soll auf Kollisionen getestet werden. Dazu wird in einer Unterschleife noch einmal über alle `GameObjects` aus `game_model()->objects()` iteriert und auf `flappy_box::model::Box` gecastet (jedoch nur, wenn beide, d.h. das BO und das aktuelle `GameObject` der Unterschleife bei einem Aufruf von `isAlive() true` zurückliefern). Anschließend wird eine Kollision über umhüllende Kugeln berechnet. Ihre Mittelpunkte sollen genau die Positionen der zu testenden Boxen sein und ihre Radien die jeweilige Boxgröße `box->size()`. Recherchieren Sie hierzu über Schnitt-Tests zwischen zwei Kugeln (der genaue Schnittkreis muss nicht berechnet werden). Schneiden sich die Hüllkugeln, so sind die Boxen kollidiert und werden beide mittels `setAlive(false)` invalidiert. Dem Spieler könnten an dieser Stelle Extrapunkte hinzugerechnet werden. Anschließend fährt die Unterschleife fort. (Hinweis: Achten Sie in der Unterschleife darauf, keine Kollision von BO mit sich selbst zu berechnen.)

Wichtiger Hinweis: Löschen Sie keine `GameObjects` manuell, sondern verwenden Sie immer die `setAlive()`-Methode. Das eigentliche Entfernen von Spielobjekten wird durch die Engine verwaltet.

Die Implementierung der Hilfsmethoden:

void addBoxToGame(::controller::Logic& l):

- Erzeugen Sie eine zufällige x-Koordinate innerhalb der Spielwelt oder wählen Sie zufällig aus einem Preset von x-Koordinaten
- Erzeugen Sie eine zufällige Größe oder wählen Sie zufällig aus einem Preset von Größen
- Erzeugen Sie einen neuen Shared Pointer auf eine `flappy_box::model::Box`
- Setzen Sie die Position der Box mit der eben erzeugten x-Koordinate und festen y- und z-Koordinaten
- Setzen Sie die maximale Position der Box für den ersten Oktanten der Spielwelt (die anderen Grenzen sind symmetrisch dazu)

- Bedenken Sie, dass die Position der Box ihren Mittel-/Schwerpunkt darstellt und bei der Ermittlung der Maximalposition auch die Größe der Kiste eine Rolle spielt
- Setzen Sie die Größe der Box mit der eben erzeugten Größe
- Fügen Sie das Box-Modell dem Spiel über `1.game_model->addGameObject()` hinzu

void setForce(std::shared_ptr< flappy_box::model::Box > & box, std::shared_ptr< flappy_box::model::Paddle > & paddle):

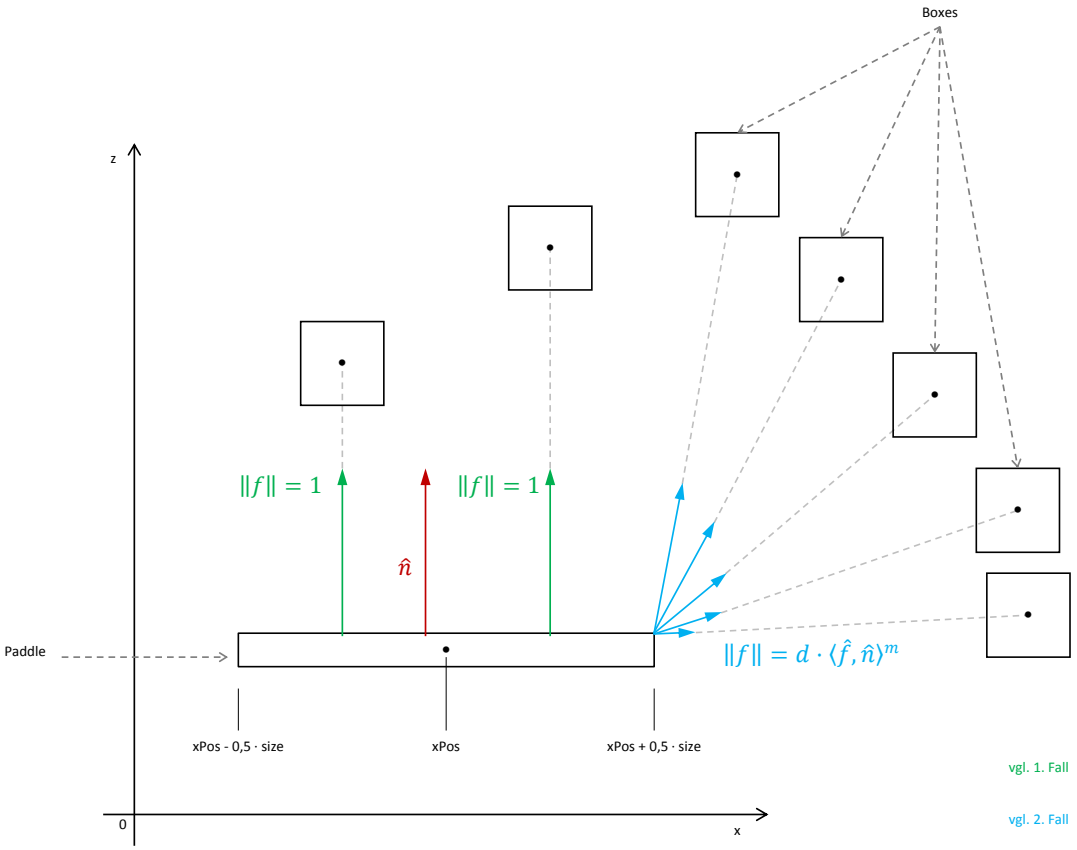
Vorüberlegung für eine vereinfachte Spielobjekt-Interaktion:

Es wird ein Kraftbereich angenommen, der von der Paddle-Fläche ausgeht und einen pro Flächenelement *dA* konstanten Krafteintrag auf die Boxen bewirkt - vergleichbar mit einem Luftstrom, welcher die Kisten schweben lässt. Der Einfachheit halber wird angenommen, dass die untere Seitenfläche einer Box stets orthogonal zum (normalisierten) Kraftvektor ausgerichtet ist, sodass ihr voller Flächeninhalt in die Berechnung eingeht. Es soll so skaliert werden, dass auf ein Einheits-Flächenelement (*dA* = 1) eben die Kraft wirkt, die eine vertikale Beschleunigung von $|\vec{a}| = 10$ (also ungefähr die negative Erdbeschleunigung) auslöst. Unter der vereinfachten Annahme der Masse *m* = 1 ergibt sich:

$$\left|\hat{f}(dA = 1) \cdot ds\right| = |m \cdot \vec{a}| = 10 \rightarrow ds = 10$$

Die auf eine Boxfläche wirkende Gesamtkraft kann nun als Produkt des normalisierten Kraftvektors und einem Skalierungsterm *s* formuliert werden, der sich aus der Summe der Flächenelemente einer Boxfläche ergibt: $s \coloneqq \sum ds$. Die Anzahl der Flächenelemente ergibt sich aus der quadratischen Beziehung zur Kantenlänge einer Kiste. Der resultierende Skalierungsterm *s* ist wie folgt beschrieben: $s \coloneqq \sum ds = 10 \sum dA = 10 \cdot size^2$

*tl;dr: Benutzen Sie zur Skalierung des normalisierten Kraftvektors den Term s = 10 * size^2*



Es sollen nun zwei Fälle betrachtet:

1. Die Box befindet sich oberhalb des Spieler-Paddles über der Paddlefläche, d.h. sie liegt innerhalb `-0.5*paddle->size()` und `+0.5*paddle->size()`, vgl. Skizze:
   - Setzen Sie den Kraftvektor gleich der Normale des Paddles: $\hat{f}$ = (0,0,1)
   - Skalieren Sie mit dem oben definierten Skalierungsterm
2. Die Box befindet sich oberhalb des Paddles, jedoch nicht mehr über der Fläche, sondern links bzw. rechts davon (vgl. Skizze):
   - Berechnen Sie den Vektor *f* zwischen der am nächsten liegenden Paddle-Ecke und der Kistenposition und normalisieren Sie ihn
   - Verwenden Sie als Kraftabschwächung *k* am Rand einen Term, bei dem das Skalarprodukt aus dem eben berechneten Kraftvektor und der Paddle-Normalen eingeht. Sie können den Kraftverlauf am Rand weiter verfeinern, indem Sie einen Term der folgenden Form verwenden: $f = d \cdot < \hat{n}, \hat{f} >^m$ mit *d, m > 0*
   - Skalieren Sie $\hat{f}$ mit *k* (Experimentieren Sie mit verschiedenen Werten bis Sie ein zufriedenstellendes Ergebnis haben)
   - Skalieren Sie mit dem oben definierten Skalierungsterm

Berechnen Sie die auf eine Kiste einwirkende Kraft entsprechend der zwei Fälle und übergeben Sie sie der Box mit Hilfe von `box->setExternalForce()`.

void restartGame(::controller::Logic& l):

Nutzen Sie diesen Code als Basis für ihre Implementierung.

```
void WorldLogic::restartGame( ::controller::Logic& l )
{
    // invalidate all game objects
    for ( auto o : l.game_model()->objects() )
    {
      o->setAlive( false );
    }
    // reject invalidation for world object
    _model->setAlive( true );
    _model->setPlayerPoints( 0 );
    _model->setRemainingLives( 5 );
    // create and configure new paddle object
    std::shared_ptr< flappy_box::model::Paddle > user_paddle
        = std::make_shared< flappy_box::model::Paddle >("PlayerPaddle");
    user_paddle->setSize( vec3_type( 10.0, 1.0, 2.5 ));
    user_paddle->setPosition( vec3_type( 0.0, 0.0, - _model->getWorldHalfHeight()
        + user_paddle->size()[2] * 2.0 ));
    user_paddle->setMaxPosition( vec3_type( _model->getWorldHalfWidth()
        - user_paddle->size()[0] * 0.5, 0.0, _model->getWorldHalfHeight() ) );
    // add paddle object
    l.game_model()->addGameObject( user_paddle );
    // unset restart flag
    _shallRestartTheGame = false;
}
```

Aufgabe 5.2.4 – Die Klasse GameOverLogic : public ::controller::Logic::ObjectLogic

Legen Sie die beiden Dateien `game_over_logic.hpp` und `game_over_logic.cpp` in den entsprechenden Ordnern an (bzw. kopieren Sie box_object_logic...).

Neben der (privaten) Membervariablen `_model` vom Typ `std::shared_ptr< model::GameOver >` (initialisiert durch ein im Konstruktor übergebenen `const std::shared_ptr< model::GameOver >&`) wird

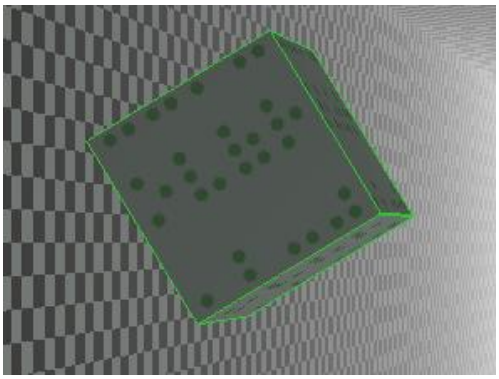
keine weiter Membervariable benötigt. Die Methode `virtual bool advance( ::controller::Logic&, ::controller::InputEventHandler::keyboard_event const & )` liefert lediglich ein false zurück. (Da das GameOver-Objekt nur zum Anzeigen des Gesamtpunktestands zuständig sein soll, ist keine Controller-Funktion von Nöten.)

Aufgabe 5.2.5 – Vervollständigen von FlappyEngine::init

Legen Sie in der `init`-Methode eine Variable vom Typ `std::shared_ptr<flappy_box::model::World>` an und fügen Sie sie dem Spiel mittels `game_model()->addGameObject()` zu. (Das Rücksetzen und Neustarten inkl. Anlegen aller weiteren Spielelemente wird dann über dieses Objekt gesteuert).

Registrieren Sie die Delegates der Logic-Klassen zu allen Modell-Klassen mittels `logic_factory.registermodule<...>()`.

Aufgabe 5.3 – Rendering der Box (2 Punkte)



Nachdem das Spiel nun eigentlich funktioniert, widmen wir uns nun der Computergraphik, indem Schritt für Schritt die OpenGL-Renderer der einzelnen Komponenten ausgetauscht werden. Beginnen wir mit dem `::flappy_box::view::BoxGLDrawable`. Diese Klasse besitzt im Wesentlichen drei Methoden, den Konstruktor, den Destruktor und die Methode „visualize“.

Aufgabe 5.3.1 – Vertex Arrays

Zunächst soll die Implementierung der Methode „visualize“ ersetzt werden. Anstatt des bereits bekannten Drahtgitterwürfels soll nun ein massiv aussehender Würfel gezeichnet werde. Hierfür nutzen wir als OpenGL-API *Vertex Arrays*. Anstatt, wie bisher mit `glBegin` und `glVertex` einzelne Datenwerte an die Graphikkarte zu übergeben, erlauben Vertex Arrays alle Daten gemeinsam mit einem Befehl abzusetzen. Vor allem für komplexere Modelle (also nicht unbedingt für unsere kleine Box) ist diese API deutlich schneller.

Kernpunkt sind die Methoden `glEnableClientState`, `glVertexPointer`, `glNormalPointer` und `glDrawArrays`. Lesen Sie sich die entsprechenden Dokumentationen auf OpenGl.org[1] durch. Von der Grundidee her bereiten Sie einfach alle Daten die sie mit `glVertex` einzeln transportieren würden in einem einzigen Array vor und schicken diese dann mit nur einem Zeichenbefehl gemeinsam zur Graphikkarte.

1. Legen Sie sich zwei Arrays an, eines für die Vertex-Daten (mehrere Float-Tripple als Vektoren) und eines für die Normalen-Daten. (lesen Sie zunächst noch die Aufgabe weiter, bis ihnen klar ist was diese Daten genau enthalten müssen.)
2. Aktivieren Sie über `glEnableClientState` das *Vertex Array* und das *Normal Array*.
3. Geben Sie per `glVertexPointer` und glNormalPointer ihre Daten-Pointer an die OpenGL-API weiter. OpenGL merkt sich hier nur die Pointer; es findet noch kein Datentransfer statt!
4. Rufen Sie `glDrawArrays` auf um die Daten an die Graphikkarte zu übertragen und zu zeichnen. Nutzen Sie als Zeichenprimitiv `GL_QUADS`. Jede Seite des Würfels soll durch ein Quad gezeichnet werden. Jede Seite soll außerdem einen eindeutigen Normalenvektor besitzen, damit der Würfel flach aussieht aber korrekt beleuchtet wird. Sie benötigen in ihren beiden Datenarrays also jeweils 24 Vektoren.

[1] z.B. https://www.opengl.org/sdk/docs/man2/xhtml/glDrawArrays.xml

Zeichnen Sie so einen Würfel mit Kantenlänge 1 (jede Koordinate läuft von -0,5 bis 0,5). Nutzen Sie die bereits vorgegeben Transformationen (inklusive Scale) um die Boxen korrekt zu positionieren. Achten Sie bei der Angabe der Vertices auf die Reihenfolge, damit ihre Würfel auch bei aktivierten Back-Face-Culling korrekt dargestellt werden (`glEnable(GL_CULL_FACE)`).

Setzen Sie zunächst eine Lichtquelle (`GL_LIGHT0`) zum Testen an eine beliebige Stelle. Aktivieren Sie die Beleuchtung. Achten Sie auf die korrekten Einstellungen für die Lichtquellen und Materialeigenschaften. Nutzen sie `GL_COLOR_MATERIAL`, `glColorMaterial`, `GL_LIGHTING`, `GL_LIGHT0`, `glLightf`. Stellen Sie die Beleuchtung so ein, damit ambiente und diffuse Beleuchtung berechnet wird[2]. Setzen Sie die Farbe der Box am besten auf Weiß, damit Sie die Beleuchtungsergebnisse gut sehen können. Da sich die Farbe nicht per Vertex ändert, können Sie diese einfach wie gewohnt per `glColor` setzen.

Vergessen Sie nicht am Ende der Methode „visualize“ den OpenGL-State wieder in den Zustand zu versetzen, den die anderen Zeichenroutinen erwarten, vor allem deaktivieren Sie die VertexArrays mit `glDisableClientState`.

Aufgabe 5.3.2 – Überlagerter Wire-Frame

Je nach Größe, Orientierung und Beleuchtung der Box ist es teilweise nicht einfach die Kanten zu sehen. Daher sollen nun die Kanten mit einer überlagerten Wire-Frame-Darstellung betont werden. Wird die Beleuchtung ausgeschaltet und für die Kanten eine deutliche Farbe gesetzt, so sind diese gut zu sehen.

Sie könnten nun, ähnlich zum alten Rendering-Code, einfach per GL_LINE die Kanten des Würfels selber nachzeichnen. Hierdurch würden Sie allerdings den Datenaufwand und Verwaltungsaufwand verdoppeln. OpenGL bietet ihnen aber die Möglichkeit ihr neu erstelltes Modell auch für eine Wire-Frame-Darstellung zu nutzen. Die Funktion `glPolygonMode` bietet ihnen diese Funktionalität. Schalten Sie diesen Modus auf GL_LINE um und zeichnen Sie ihre Vertex Arrays einfach ein zweites Mal. (Vergessen Sie nicht den Modus auch wieder zurück zu schalten. Lesen Sie die Dokumentation um herauszufinden was die korrekten Parameterwerte sind.) Nutzen Sie `glLineWidth` um die Dicke der Linien in Pixel zu verändern.

Aufgabe 5.3.3 – Texturierung

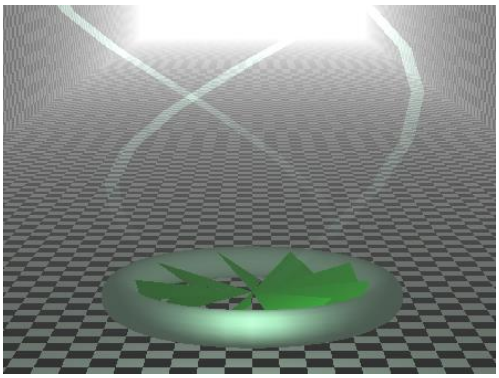
Zuletzt soll nun die Box texturiert werden. Der Einfachheit halber möchten wir eine Textur laden und auf allen sechs Seiten der Box anzeigen. Hierfür braucht die Box nun Texturkoordinaten. Legen Sie entsprechend Aufgabe 5.2.1 ein weiteres Datenarray an, in dem Sie für jeden Vertex den 2D-Vektor für die Texturkoordinaten ablegen. Nutzen Sie `glEnableClientState(GL_TEXTURE_COORD_ARRAY)` und `glTexCoordPointer` um die neuen Daten zu übergeben.

1. Im Konstruktor möchten wir die Textur erzeugen. Legen Sie sich zunächst ein Datenarray an welches 128 × 128 Pixel mit RGB Werten als Bytes speichert. Befüllen Sie dieses Array zunächst mit einfachen Werten, z.B. jeden Pixel mit einer konstanten und erkennbaren Farbe.
2. Erzeugen Sie mit `glGenTextures` ein neues Textur-Objekt. Zur Speicherung der Textur-ID brauchen Sie eine neue Member-Variable in ihrem Objekt. Rufen Sie im Destruktor `glDeleteTextures` auf, um das Textur-Objekt nachdem es nicht mehr benötigt wird auch wieder verzugeben.
3. Binden Sie das neu erzeugte Textur-Objekt an die Standard-Textur-Einheit von OpenGL mit `glBindTexture` und `GL_TEXTURE_2D`.
4. Stellen Sie die Parameter dieses Textur-Objekts ein. Nutzen Sie `glTexParami`. Wrapping soll in beide Richtungen auf GL_REPEAT gestellt werden. Die Texturfilterung soll sowohl für Vergrößerung als auch für Verkleinerung auf GL_NEAREST gestellt werden.

[2] http://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_shading_model

5. Nutzen Sie glTexImage2D um ihr Texturbild aus ihrem vorbereiteten Pixel-Array in das Textur-Objekt hoch zu laden.
6. In ihrem Zeichen-Code in „visualize“ nutzen Sie `glEnable(GL_TEXTURE_2D)` und `glBindTexture` um Texturen zu aktivieren und ihr Textur-Objekt für die folgenden Zeichenbefehle auszuwählen. Vergessen Sie nicht vor dem Zeichenen der Wire-Frame-Darstellung die Textur wieder auszuschalten.
7. Ersetzen Sie nun das Befüllen des Pixel-Array im Konstruktor. Öffnen Sie mit dem Befehl `fopen`[3] die Datei „thehead.raw“ aus dem „res“-Unterordner. Achten Sie auf die korrekte Angabe des relativen Pfades vom Ausführungsverzeichnis aus. Achten Sie auch darauf das `fopen` im Binär-Modus arbeiten muss. Das Bild ist als rohe-Pixeldaten abgelegt, ohne Header-Informationen. Laden Sie einfach die 128 × 128 × 3 Bytes direkt aus der Datei in ihr Pixel-Array (mit `fread`). Schließen Sie die Datei mit `fclose`.

Aufgabe 5.4 – Rendering des Paddle (3 Punkte)



Als nächstes soll nun das Paddle des Spielers mit einem neuen Rendering-Code ausgestattet werden. Als Paddle soll eine einfache Turbine darstellt werden, bestehend aus drei Teilen: dem äußeren Ring, den Rotorblättern und dem aufsteigenden Luftwirbel.

Für eine gesteigerte Dynamik setzen Sie die Position der Lichtquelle `GL_LIGHT0` abhängig von der Paddle-Position. Dies erzeugt eine dynamische Beleuchtung vor allem auch für die Boxen. Nutzen Sie `glLightfv(GL_LIGHT0, GL_POSITION...)` und setzen Sie die Position z.B. auf $(x, y - r_0 \cdot 1.5, z + r_1 \cdot 3, 1)$
Die Bedeutung der Variablen r_0 und r_1 wird in der folgenden Teilaufgaben erklärt.

Aufgabe 5.4.1 – Der äußere Ring

Der äußere Ring ist einfach ein Torus. Dieser soll als statisches 3D-Mesh erzeugt und gezeichnet werden. Hierzu nutzen wir Vertex Buffer Objects (VBOs). VBOs sind auf der der Graphikkarte angelegter Speicher. Die Datenübertragung zur Graphikkarte ist vom eigentlichen Zeichnen komplett entkoppelt. Verändern sich die Daten nicht, so bieten VBOs die beste Darstellungsgeschwindigkeit. Das Arbeiten mit VBOs entspricht aktueller Computergraphik-Praxis, unabhängig von der Programmier-API (also sowohl mit OpenGL als auch mit Direct3D, etc.). Das grundsätzliche Vorgehen ist zur Benutzung von Vertex Arrays sehr ähnlich und benutzt teilweise sogar die gleichen Befehle.

Für den äußeren Ring benutzen wir drei VBOs: Vertex-Daten, Normalen-Daten und Index-Daten. Erzeugen Sie sich diese VBOs mit einem Aufruf von `glGenBuffers`. Erzeugen Sie nun in einer neuen Klassenmethode „`updateVBOs`“ die Daten die in diese VBOs gefüllt werden sollen. Vergessen Sie auch nicht im Destruktor die VBOs mit `glDeleteBuffers` wieder freizugeben.

Als Parameter für den Torus benötigen Sie zwei Radien und die Unterteilungsstufen in zwei Richtungen (entlang dem großen Radius und entlang dem kleinen Radius). Als großen Radius r_0 nutzen Sie die Hälfte des Maximums aus der Paddle-Breite und der –Tiefe. Als kleinen Radius r_1 nutzen Sie die Hälfte der Paddle-Höhe, mindestens aber den gleichen Werte wie der große Radius. Für die Unterteilungsschritte nutzen Sie konstanten die Sie als Member ihrer Klasse deklarieren (z.B. über `static const`

[3] http://www.cplusplus.com/reference/cstdio/fopen/

`unsigned int`). Erzeugen Sie entlang des großen Radius 40 Unterteilungen und entlang des kleinen Radius 15 Unterteilungen. Erzeugen Sie in zwei geschachtelten Schleifen die Vertex-Daten, indem Sie aus den (integer) Laufvariablen die notwendigen Winkel berechnen, z.B.:

```
α = 2.0 * M_PI * static_cast<double>(i) / static_cast<double>(40)
```

Da in diesem Beispiel i nie 40 erreicht wird der Wert 2π nie erreicht, sondern genau ein Schritt davor abgebrochen. Angenommen Sie laufen mit α über den großen Radius r_0 und mit β über den kleinen Radius r_1, so lässt sich die jeweilige Vertex-Position wie folgt berechnen (vgl. Polarkoordinaten):

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} r_0 \cos\alpha \\ r_0 \sin\alpha \\ 0 \end{pmatrix} + r_1 \left(\cos\beta \begin{pmatrix} \cos\alpha \\ \sin\alpha \\ 0 \end{pmatrix} + \sin\beta \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \right)$$

Der Ausdruck in der letzten großen Klammer (nach r_1) ist übrigens die notwendige Oberflächennormale.

Um ein VBO zu befüllen, müssen Sie es zunächst, ähnlich wie die Textur aus Aufgabe 5.2.3, binden. Nutzen Sie hierfür `glBindBuffer(GL_ARRAY_BUFFER…)`. Nun können Sie, ähnlich wie bei der Benutzung von Vertex Arrays ihre Daten in einem Speicher vorbereiten und anschließend mit `glBufferData` auf die Graphikkarte hoch laden. Lesen Sie die Dokumentation und vergessen Sie nicht ihren lokalen CPU-Speicher zu löschen sobald Sie ihn nicht mehr brauchen, denn die Daten liegen nun ja auf der Graphikkarte. Da wir die Daten nur einmalig erzeugen und anschließend beliebig oft Rendern nutzen Sie den Modus `GL_STATIC_DRAW`.

Im Gegensatz zur Box (vgl. Aufgabe 5.3.1) speichern Sie nun jeden Vertex nur ein einziges Mal. Um zwischen diesen Vertices nun Primitive aufzuspannen erzeugen Sie nun die Daten für das dritte VBO: Index-Daten. Diese Index-Daten sind ein Array von `unsigned-int`-Werten. Erzeugen Sie für jedes Dreieck das Sie zeichnen wollen die drei (null-basierten) Index-Werte zu den drei Vertices die das Dreieck aufspannen wollen. Bei den vorgegebenen Unterteilen wären die ersten zwei Dreiecke z.B.: 0, 15, 1, 1, 15, 16, … Schreiben Sie Schleifen mit denen die korrekten Indices ausgerechnet und in dieses VBO geschrieben werden. Zur Kontrolle: dieses VBO muss `40 * 15 * 3 * 2 = 3600` `unsigned-int`-Werte speichern.

Nachdem die Daten nun erzeugt sind können Sie den Rendering-Code in „visualize“ ersetzen. Nutzen Sie zunächst diesen Code-Auszug als Basis:

```
::glEnableClientState(GL_VERTEX_ARRAY);
::glEnableClientState(GL_NORMAL_ARRAY);
::glColor3f(.6f, .9f, .7f);
::glBindBuffer(GL_ARRAY_BUFFER, this->ring_vbuf[0]);
::glVertexPointer(3, GL_FLOAT, 0, NULL);
::glBindBuffer(GL_ARRAY_BUFFER, this->ring_vbuf[1]);
::glNormalPointer(GL_FLOAT, 0, NULL);
::glBindBuffer(GL_ELEMENT_ARRAY_BUFFER_ARB, this->ring_vbuf[2]);
::glDrawElements(GL_TRIANGLES, ring_seg1 * ring_seg2 * 6, GL_UNSIGNED_INT, NULL);
```

In diesem Beispiel-Code ist ring_vbuf ein Array mit drei VBO-IDs und die variable ring_seg1 und ring_seg2 speichern die Unterteilungszahlen. Der einzige Unterschied zum Zeichenaufruf mit Vertex Arrays besteht darin, das vor dem Aufruf einer Pointer- Funktion (z.B. `glVertexPointer`) ein Array Buffer gebunden wird, und dass anstatt des Daten-Pointer `NULL` an die Pointer-Funktion übergeben wird. Hiermit wird OpenGL informiert die Daten aus dem jeweils entsprechenden VBO zu nutzen. Gleiches gilt auch für das VBO mit den Index-Daten welches den letzten Parameter in `glDrawElements` ersetzt. Lesen Sie für die Details die entsprechenden Dokumentationen.

Ergänzen Sie zusätzlich die Methode „visualize“ so, dass sollte sie die Paddle-Größe, und damit die beiden Radien, ändern, dass dann die Methode „updateVBOs“ erneut aufgerufen wird.

Aufgabe 5.4.2 – Die Rotorblätter

Der grundsätzliche Aufbau zum Rendern der Rotorblätter ist mit dem des äußeren Rings aus der letzten Teilaufgabe identisch: legen Sie drei VBOs an (Vertices, Normalen, Index-Daten). Jedes Rotorblatt besteht aus einem Dreieck. Alle Blätter berühren sich in dem gemeinsamen Mittelpunkt (0, 0, 0). Die beiden Außenpunkte jedes Rotorblatts ergeben sich aus der Schleifenvariable, die über die Anzahl der Rotorblätter läuft:

$$v_0 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, v_1 = \begin{pmatrix} (\mathrm{r}_0 - \mathrm{r}_1)\cos\alpha_1 \\ (r_0 - r_1)\sin\alpha_1 \\ r_1 \end{pmatrix}, v_2 = \begin{pmatrix} (r_0 - r_1)\cos\alpha_2 \\ (r_0 - r_1)\sin\alpha_2 \\ -r_1 \end{pmatrix}$$

Die beiden Winkel α_1 und α_2 ergeben sich durch leichte Offsets der Schleifenvariable:

```
α1,2 = ((static_cast<double>(i) ± 0.3) / static_cast<double>(blade_cnt))
```

Berechnen Sie darüber hinaus sich korrekte Oberflächennormalen für jedes dieser Dreiecke:

$$n = |(v_2 - v_0) \times (v_1 - v_0)|$$

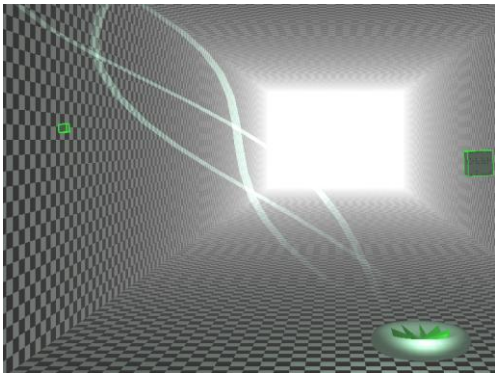
Da in der Computergraphik tricksen durchaus erlaubt ist, wenn man es nicht oder kaum sieht, können Sie sich überlegen, ob sie den zentralen Punkt v_0 nur einmal abspeichern und für jedes Rotorblatt verwenden. Dann können Sie für diesen Punkt natürlich keine korrekte Normale angeben, aber das wird man im Endergebnis kaum sehen, wenn Sie hier eine „alternative Normale“ sinnvoll wählen.

Erzeugen Sie auch die Index-Daten wieder entsprechend. Erzeugen Sie einen Rotor mit 9 Blättern.

Da alle diese Berechnungen wieder von den zwei Radien abhängen die wir schon für den äußeren Ring benutzt haben, implementieren Sie ihren Code zur Erzeugung der Geometrie einfach auch in der Methode „updateVBOs“.

Fügen Sie nun den Zeichencode entsprechend Teilaufgabe 5.4.1 nun auch für die Rotorblätter ein. Was nun noch fehlt, ist dass der Rotor sich schnell aber sichtbar dreht. Da diese Rotation nichts mit der Spielmechanik zu tun hat, sondern nur die Optik beeinflusst ist es in Ordnung den entsprechenden Zustand direkt in PaddleGlDrawable abzulegen. Legen Sie eine neue Membervariable an, die den aktuellen Rotationswinkel speichert. Durch `glRotated(blades_ang, 0.0, 0.0, 1.0);` kann der Rotor nun entsprechend drehen. Achten Sie darauf diese Rotation erst nach dem Zeichnen des äußeren Rings durchzuführen. Basieren auf dem TimeStep des Modells können Sie nun den Rotor sich drehen lassen indem sie die Variable `blades_ang` entsprechend erhöhen oder erniedrigen. Stellen Sie ihre Konstanten so ein, dass der Rotor sich pro Sekunde um 720 Grad dreht.

Aufgabe 5.4.3 – Die aufsteigenden Luftwirbel



Das letzte Element des Paddles ist die Darstellung eines aufsteigenden Luftwirbels. Diese soll allerdings nicht statisch sein und sich nur drehen, so wie die Rotorblätter, sondern soll sich abhängig von der Paddle-Position in globalen Koordinaten aktualisieren. So wie in dem Bild rechts zu sehen soll der Luftwirbel „nachgezogen“ werden, wenn das Paddle sich beispielsweise schnell nach rechts bewegt. Statische VBOs, so wie wir sie bisher benutzt haben, werden dieser Anforderung nicht gerecht. Stattdessen benutzen wir nun dynamische VBOs deren Daten wir für denen Rendering-Aufruf aktualisieren.

Jeder der Luftwirbel besteht aus deinem Triangle-Strip der Länge „len“, nutzt also ein VBO mit 2 * len Vertices. Wir möchten drei Wirbel der Länge 20 erzeugen. Die Wirbel werden nicht beleuchtet, werden aber mit Transparenz gezeichnet. Wir brauchen daher keine Normalen-Daten, aber dafür Farbdaten. Da jedes VBO einfach ein einzelner Triangle-Strip ist, brauchen wir auch keine Index-Daten. Die Transparenz soll sich einfach aus der Höhe ergeben, ist daher also für alle drei Wirbel identisch. Für alle Wirbel kann also das gleiche VBO für die Farbdaten genutzt werden. Sie brauchen für die drei Wirbel also nur vier VBOs.

Das Farb-VBO ist statisch und von nichts abhängig. Erzeugen und befüllen Sie es im Konstruktor. Als Farbdaten nehmen sie vier Floats (RGB und Alpha). Geben Sie jeden Farbwert zweimal an, da wir bei den Vertex-Daten ja auf jeder Höhe auch zwei Punkte brauchen um das Triangle-Strip aufzuspannen. Die RGB-Komponenten der Farbe sollten konstant sein. Benutzen Sie für den Alpha-Wert folgende Formel (a ist die Schleifenvariable):

$$\alpha = 0.25\ (\sin(\pi\ \text{static\_cast<double>(a)} / \text{static\_cast<double>(len -1)}\ )^2$$

Da sich diese Farbdaten nicht ändern werden nutzen sie als Modus für das VBO wieder `STATIC_DRAW`. Die Vertexdaten hingegen werden wir dynamisch ausrechnen. Legen Sie, ebenfalls im Konstruktur, den dafür notwendigen Speicher auf der Graphikkarte bereits an. Nutzen Sie, wie gewohnt `glBufferData`, übergeben Sie allerdings anstatt eines Pointers auf die Daten im Hauptspeicher einfach `NULL`. Durch diesen Aufruf wird der notwendige Speicherbereich im Graphikkartenspeicher allokiert, aber nicht gefüllt.

Um die aufsteigenden Wirbel zu verwalten, nutzen wir einfach zu benutzende Datenstrukturen im Hauptspeicher:

```
// air vortex data : [a][b][0] = core line, [a][b][1] = vortex line
vec3_type vortex_dat[vortices_cnt][vortex_line_len][2];
```

vortices_cnt ist die Anzahl der Wirbellinien die gezeichnet werden sollen; vortex_line_line ist die schon erwähnte Länge der Triangle-Strips. Für jede Position (Höhe) des Bandes speichern wir zwei Vektoren: mit Index 0, einmal die Position des Mittelpunkts des Spieler-Paddles und zum zweiten, mit Index 1, die konkrete Position des Bandes. Das erlaubt und die Wirbel nach oben etwas zu öffnen, was auch dem breiteren Einflussbereich in der Spiellogik nachempfunden ist. Initialisieren Sie im Konstruktur alle Vektoren dieses mehrdimensionalen Arrays mit dem Mittelpunkt (der Position) des Spieler-Paddles.

In der Methode „visualize“ sollen nun diese Daten für jedes Luftwirbelband aktualisiert und anschließend in das korrekte VBO hochgeladen werden. Die „Startpunkte“ für jedes dieser Bänder sollen sich

wie die Turbinenblätter (vgl. Aufgabe 5.3.2) verhalten, also gleichmäßig auf einem Kreis mit Radius ($r_0 – r_1$) an der Oberseite des Paddles positioniert werden.

1. Schreiben Sie eine Schleife über alle Wirbelbänder
2. Schreiben Sie eine Schleife in der Sie die bisherigen Vektoren des aktuellen Wirbelbandes nach oben laufen lassen. Nutzen Sie folgenden Teil-Code als Ausgangspunkt:

```
for (unsigned int t = vortex_line_len - 1; t > 0; --t) {
    vortex_dat[l][t][0] = vortex_dat[l][t - 1][0]
        + vec3_type(0, 0, timestep_sec * vortex_speed);
    vortex_dat[l][t][1] = vortex_dat[l][t][0]
        + (vortex_dat[l][t - 1][1] - vortex_dat[l][t - 1][0]) * 1.075;
}
```

   Beachten Sie die Schleifengrenzen, und machen Sie sich klar warum die Daten in dieser Richtung durchlaufen werden müssen. Was müssten Sie ändern wenn Sie aufgrund äußerer Zwänge die Schleife in positiver Richtung durchlaufen müssten? Nutzen Sie für die Konstante `vortex_speed` einen Wert von 100. Der Wert 1.075 führt die Spreizung des Luftwirbels durch. Probieren Sie hierfür auch andere Werte und Formeln aus.
3. Abschließend müssen Sie neue Werte für die Startpositionen berechnen (Index 0). Setzen Sie den Vektor für den Mittelpunkt `vortex_dat[l][0][0]` einfach auf die aktuelle Position des Spieler-Paddles. Den Vektor für das eigentliche Wirbelband setzen Sie abhängig der Bandnummer auf eine Kreisposition wie oben beschrieben. Achten Sie darauf, dass diese Startpositionen sich genau gleich schnell wie die Rotorblätter bewegen. Achten Sie darauf, dass `glRotate` mit Winkelangabe in Grad arbeitet, die Funktionen `sin` und `cos` aber mit Winkelangaben im Bogenmaß.

Nun müssen aus diesen Daten die Daten in den VBOs erzeugt werden. Eine effiziente Möglichkeit dynamische Daten in ein VBO hochzuladen bietet die Möglichkeit den Graphikkartenspeicher in den virtuellen Addressbereich Hauptspeicher zu mappen. Nutzen Sie `glMapBuffer`[4] hierfür. Sie erhalten einen normalen Pointer auf den Speicherbereich der den GPU-Speicher des VBOs repräsentiert. Da wir nur neue Daten schreiben möchten wählen Sie als Modus `GL_WRITE_ONLY`. Hierdurch werden keine Daten von der Graphikkarten zurückgelesen, was sehr teuer wäre, sondern nur die neuen Daten an die Graphikkarte geschickt.

Laufen Sie nun in einer Schleife ihr Band nach oben durch und erzeugen Sie für jeden Vektor zwei 3D-Pointe die Sie in den gemappten Speicher schreiben. Verschieben Sie beide Punkte etwas nach links und rechts auf der X-Achse, damit das Band eine gewisse Dicke erhält. Nutzen Sie diesen Quellcode als Basis für ihre Implementierung:

```
float * dat = static_cast<float*>(::glMapBuffer(...));
for (...) {
    vec3_type p = vortex_dat[l][t][1];
    p[0] -= 0.5 * air_vortex_band_width;
    *(dat++) = static_cast<float>(p[0]);
    *(dat++) = static_cast<float>(p[1]);
    *(dat++) = static_cast<float>(p[2]);
    p[0] += air_vortex_band_width;
    *(dat++) = static_cast<float>(p[0]);
    *(dat++) = static_cast<float>(p[1]);
    *(dat++) = static_cast<float>(p[2]);
}
::glUnmapBuffer(...);
```

[4] http://www.opengl.org/sdk/docs/man/docbook4/xhtml/glMapBuffer.xml

Nun sind alle notwendigen Daten auf der Graphikkarte um die Bänder zu zeichnen. Für den Effekt der Durchsichtigkeit müssen die Blending aktivieren `GL_BLEND` und die `glBlendFunc`[5] auf `GL_SRC_ALPHA` und `GL_ONE` setzen. Damit diese materielosen Bänder nicht im Z-Puffer andere Objekte Verdecken deaktivieren Sie das Schreiben des Tiefen-Puffers während Sie die Bänder zeichnen (nutzen Sie `glDepthMask`). Nutzen Sie die VBOs als `GL_VERTEX_ARRAY` und `GL_COLOR_ARRAY` und zeichnen Sie jedes Band als Triangle-Strip mit einem Aufruf von `glDrawArrays` (da wir keine Index-Daten nutzen).

Achten Sie darauf, dass die Vertex-Daten dieser Bänder bereits in globale Weltkoordinaten gespeichert sind und nicht in lokalen Objektkoordinaten wie alle anderen graphischen Elemente des Paddles. Wählen Sie daher die passende Stelle um die Bänder zu zeichnen.

Aufgabe 5.5 – Rendering der Welt (2 Punkte)

Die letzten graphischen Elemente unseres Spiels sind die Welt und die notwendige Textausgabe. Die Welt selbst, in `WorldGLDrawable`, soll ein einfacherer, texturierter Korridor sein. Für die Textur können Sie entweder ein einfaches Schachbrettmuster im Code erzeugen (so wie in unserer Lösungsvorgabe), oder entsprechend Aufgabe 5.3.3 eine Textur laden. Für einen besseren Tiefeneindruck verwenden Sie `glFog`. Für die Textausgaben verwenden Sie die entsprechenden Funktionen der GLUT.

Achten Sie bei ihrem Design für den Korridor darauf nicht nur einfach wenige große Polygone (z.b. einfach nur vier Qudas) zu zeichnen. Da wir in diesem Übungsblatt mit der klassischen OpenGL-Pipeline arbeiten wird die Beleuchtung per Vertex berechnet und pro Dreieck oder Viereck interpoliert. Um die Auswirkungen der Lichtquelle korrekt sehen zu können brauchen Sie also auch auf großen, flachen Wänden genügend Vertices. Unsere Beispielimplementierung nutzt in der Tiefe 1000 Quads und in der Breite und der Höhe entsprechend viele Quads basierend auf der Weltgröße aus dem Modell, so dass die Quads annähernd quadratisch sind.

Aufgabe 5.5.1 – glFog

Die Idee von `glFog`[6] ist bei Szenen die eigentlich unendlich weit nach hinten reichen würden die Existenz der Far-Clipping-Plane zu verbergen. Abhängig vom Tiefenwert eines Fragments das durch den Rasterisierer der Graphikkarte erzeugt wird, wird der Farbwert des Fragments mit einem Fog-Farbwert verrechnet. So können Szeneobjekte die weiter hinten sind „ausgeblendet“ werden. In moderner Computergraphik (Spielen) wird versucht weitgehend auf Fog zu verzichten, weil er häufig unangenehm auffällt. Zur Verstärkung des Tiefeneindrucks wird er jedoch in einfacheren Graphiken nach wie vor eingesetzt.

Mit den Methoden `glFogi`, `glFogf` und `glFogfv` können Sie die Parameter des Fogs einstellen. In unserer Beispiellösung nutzen wir einen linearen Fog zwischen den Werten 0 und 500 mit weißer Farbe. Mit `glEnable(GL_FOG)` können Sie den Fog aktivieren. Stellen Sie den Fog in ihrer Weltszene so ein, dass ein guter Tiefeneindruck des Korridors entsteht.

Aufgabe 5.5.2 - Textausgabe

Die GLUT bietet einfache Hilfsfunktionen für simple Textausgabe. Diese sollen für unsere Zwecke ausrechen.

[5] https://www.opengl.org/sdk/docs/man2/xhtml/glBlendFunc.xml
[6] https://www.opengl.org/sdk/docs/man2/xhtml/glFog.xml

OpenGL besitzt zusätzlich zur 3D-API, die wir inzwischen umfangreich genutzt haben, auch Anteile für 2D-Graphik (teilweise historisch bedingt). So zum Beispiel die RasterPos (Raster Position). Dieser OpenGL-interne Zustand speichert eine 2D-Position im Bildraum des Fensters. Sie können mit dem Befehlen `glRasterPos`[7] diese Position setzen.

Für die Textausgabe nutzen Sie glutBitmapCharacter[8]. Diese Funktion gibt ein einzelnes ASCII-Zeichen an der aktuellen Rasterposition aus und verschiebt die Rasterposition entsprechend der Breite des Zeichens weiter. Achten Sie darauf den OpenGL-State in einen für diese Ausgabe vernünftigen Zustand zu versetzen, z.B. die Farbe. Nutzen Sie für die Ausgabe von ganzen Strings z.B. diesen Hilfscode:

```
void renderBitmapString( const char *string )
{
    for ( const char *c = string; *c != '\0'; c++ )
    {
      glutBitmapCharacter(GLUT_BITMAP_HELVETICA_18, *c);
    }
}
```

Geben Sie in „WorldGLDrawable:: visuzalize“ nach dem Zeichnen der 3D-Szene noch aktuellen Punkte und die aktuelle Anzahl der Leben entsprechend aus, z.B. in der unteren, linken Ecke des Fensters. Um die Strings mit den Werten zusammenzusetzen können Sie z.B. die C-Funktion `sprintf`[9] benutzen. Alternativ, und eigentlich auch vorzuziehen, können Sie die `std::string`-Klasse aus der Standardlibrary benutzen, zusammen mit Hilfsfunktionen wie `std::to_string`. Die klassische C++-Alternative ist die Nutzung des `std::stringstream`[10].

Implementieren Sie nun auch „GameOverGLDrawable::visuzalize“, so dass eine „nette“ Game-Over-Nachricht angezeigt wird. Zusätzlich sollen die erreichten Punkte angezeigt werden und die Information das mit der Taste „r“ das Spiel neu gestartet werden kann.

Aufgabe 5.6 – Soundausgabe (2 Punkte)

Als letzte Aufgabe sollen die Audible-Klassen implementiert werden, um die beteiligten Spielobjekte zu auralisieren. Vorgegeben ist die Klasse `BoxAlAudible`, die Ihnen zur Orientierung im Umgang mit dem `SoundProvider` dienen soll. Dieser verwaltet intern das Laden von Audio-Dateien im Wav-Format, das Erzeugen von Soundbuffer-Handles sowie das Anlegen und Löschen von OpenAL-Sources. Des Weiteren stellt die Klasse `SoundProvider` eine öffentliche Methode `playSound( const char * filename, double x_position, double gain, double pitch, bool loop )` bereit, mit der bequem Wav-Dateien abgespielt werden können. Der Rückgabewert ist ein Handle auf das OpenAl-Source-Objekt, dass innerhalb der Funktion erzeugt wird. Als Parameter werden erwartet:

- `const char * filename`: ein String-Literal, dass den Dateinamen enthält
- `double x_position`: ein Wert im Intervall [-1,1] zur Definition der horizontalen Position der Schallquelle (-1 entspricht ganz links, +1 entspricht ganz rechts)
- `double gain`: ein Wert im Intervall [0,1] zur Skalierung der Lautstärke, eine Halbierung (Verdoppelung) des Wertes führt zur einer Absenkung (Anhebung) der Lautstärke um 6dB.
- `double pitch`: ein Wert zur Tonhöhen- und Geschwindigkeitsskalierung des Audiosamples - hierbei entspricht 1.0 der normalen Tonhöhe (Geschwindigkeit). Ein Wert von 0.5 würde das

[7] https://www.opengl.org/sdk/docs/man2/xhtml/glRasterPos.xml
[8] http://www.opengl.org/documentation/specs/glut/spec3/node76.html
[9] http://www.cplusplus.com/reference/cstdio/sprintf/
[10] http://www.cplusplus.com/reference/sstream/stringstream/str/

Sample auf die halbe Tonhöhe (halbe Geschwindigkeit) skalieren, 2.0 entsprechend auf die doppelte Tonhöhe und Geschwindigkeit.

- `bool loop`: ein Flag, dass angibt, ob das Sample wiederholt abgespielt werden soll (loop auf true) oder nicht (loop auf false). Achtung: Falls das Sample nicht wiederholt abgespielt werden soll, wird nach dessen Beendigung das zugehörige OpenAl-Source-Objekt gelöscht. Das Handle wird damit ungültig.

Allgemeiner Hinweis: Sollten Sie eigene Sound-Dateien verwenden wollen, achten Sie auf ein durchgehend einheitliches Sample-Format. Getestet wurden Mono-Dateien mit 44.1kHz Samplingrate und 16-bit Genauigkeit.

Achten Sie auf den Namespace `::flappy_box::view`!

Aufgabe 5.6.1 – Die Klasse WorldAlAudible : public ::view::AlRenderer::Audible

Legen Sie die beiden Dateien `world_al_audible.hpp` und `world_al_audible.cpp` in den entsprechenden Ordnern an (bzw. kopieren Sie box_al_audible...).

Der Audible-Delegate des Weltobjektes soll die Hintergrundmusik abspielen. Dazu werden Wav-Dateien im Konstruktor über die `playSound()`-Methode des `SoundProviders` geladen und mit Wiederholung abgespielt. Experimentieren Sie mit verschiedenen Kombinationen der Beispiel-Sounds aus dem Ordner "res" (beat.wav, high.wav, low.wav, synth.wav) unter verschiedenen Parametereinflüssen von Position, Gain oder Pitch! Falls Sie eigene Soundsamples verwenden wollen, achten Sie darauf, dass sie die gleiche zeitliche Länge oder Vielfache davon aufweisen, sodass beim Loopen keine rhythmischen Verschiebungen entstehen.

Die Membermethode `auralize()` kann leer bleiben.

Zusatz: Nutzen Sie die `auralize()`-Methode für musikalische Experimente, indem Sie die Lautstärke/Position/Tonhöhe während des Spielverlaufes dynamisch ändern (z.B. bzgl. der bisher erreichten Punkte). Hierzu müssen Sie sich die Handles auf die Sources speichern, die von `playSound()` zurückgegeben werden. Mit Hilfe der Handles können Sie nun die Eigenschaften des entsprechenden Sound-Objekts über `alSourcef()`/`alSource3f()` ändern.

Aufgabe 5.6.2 – Die Klasse GameOverAlAudible : public ::view::AlRenderer::Audible

Legen Sie die beiden Dateien `game_over_al_audible.hpp` und `game_over_al_audible.cpp` in den entsprechenden Ordnern an (bzw. kopieren Sie box_al_audible...).

Der Audible-Delegate des `GameOver`-Objektes soll einen nicht geloopten Sound bei seiner Erzeugung abspielen. Dieser Sound wird analog zur Klasse `WorldAlAudible` im Konstruktor mittels `playSound()` des `SoundProviders` geladen und direkt abgespielt. Verwenden können Sie den Sound "laugh.wav" aus dem Ordner "res" oder einen eigenen. Die Membermethode auralize() bleibt leer.

Aufgabe 5.6.3 – Die Klasse PaddleAlAudible : public ::view::AlRenderer::Audible

Legen Sie die beiden Dateien `paddle_al_audible.hpp` und `paddle_al_audible.cpp` in den entsprechenden Ordnern an (bzw. kopieren Sie box_al_audible...).

Der Audible-Delegate des Paddle-Objektes soll dessen Bewegung auralisieren. Hierbei werden Tonhöhe und Position dynamisch zur Laufzeit des Spiels geändert. Als Soundsample soll dieses Mal eine mit ALUT generierte synthetische Wellenform dienen.

Als (private) Membervariable `_model` wird ein Shared Pointer auf ein `flappy_box::model::Paddle` benötigt (vgl. `BoxAlAudible`). Sie wird durch ein im Konstruktor übergebenen `const std::shared_ptr< model::Paddle >&` initialisiert. Als weitere Membervariablen sind zu deklarieren (alle vom Typ `ALuint`):

- `_alPaddleSource`, speichert das Handle auf ein OpenAl-Source-Objekt
- `_alPaddleBuffer`, speichert das Handle auf einen OpenAl-Soundbuffer

Realisieren Sie im Konstruktor zunächst das Erstellen und Binden des Soundsamples. Lassen Sie mittels `alutCreateBufferWaveForm()` eine Sinuswelle mit einer tiefen Frequenz (z.B. 100 Hz) und einer Länge von 1 Sekunde erzeugen. Die Phasenlage soll 0 betragen. Weisen Sie das zurückgegebene Handle der Membervariablen `_alPaddleBuffer` zu. Generieren Sie dann ein gültiges Source-Handle mit `alGenSources()` und speichern Sie dieses in `_alPaddleSource`. Als nächstes wird die Source konfiguriert (Recherchieren Sie dazu die Befehle `alSourcei`/`alSourcef`/`alSource3f`):

- weisen Sie der Source den `_alPaddleBuffer` zu
- setzen Sie das Looping auf `true`
- setzen Sie die Lautstärke (`Gain`) auf einen mittleren Wert (z.B. 0.4)
- lassen Sie die Source mit `alSourcePlay()` losspielen

In der überschriebenen Methode `auralize()` sollen die Source-Parameter für Position und Pitch je nach Zustand des Paddles dynamisch verändert werden:

- Position: Die Links-Rechts-Position der Source soll mit der Position des Paddles wandern. Dazu muss diese auf das Intervall [-1,1] abgebildet werden, sodass die linke Maximalposition des Paddles gleich (-1,0,0) und die rechte Maximalposition gleich (1,0,0) entspricht. Nach dem Mapping setzen Sie die Schallquellenposition mittels `alSource3f()`
- Pitch: Die Tonhöhe der Schallquelle soll die Geschwindigkeit des Paddles widerspiegeln. Hierzu muss ein Skalierungswert s berechnet werden, der sich im Interval [0.5,1.5] bewegen soll. Finden Sie die Konstanten m und n, sodass sich folgender linearer Zusammenhang zwischen der Pitch-Skalierung s und der Geschwindigkeit v_x in x-Richtung ergibt:

$$s(v_x) = m \cdot v_x + n \text{ für } v_x \text{ in } [0, v_{max}] \qquad n, m = \text{const., sodass } s \text{ in } [0.5, 1.5]$$

Hinweis: Übernehmen Sie den Wert der Maximalgeschwindigkeit des Paddles direkt aus der Klasse `PaddleLogic` oder schreiben Sie ein Get-Methode.

Setzen Sie zum Schluss den berechneten Skalierungswert *s* mittels `alSourcef()`.

Um den Paddle-Sound bei Spielverlust zu stoppen, muss in der `auralize()`-Methode das alive-Flag des Modells mit `_model->isAlive()` abgefragt werden. Hat dieses den Wert `false`, so verwenden Sie `alSourceStop()`, `alDeleteSources()` und `alDeleteBuffers()` in der richtigen Reihenfolge, um alle Ressourcen wieder freizugeben.

Zusatz: Perturbieren Sie die Pitch-Skalierung mit einem kleinen zufälligen Offset für ein dynamischeres Klangerlebnis.

Aufgabe 5.6.4 - Vervollständigen von FlappyEngine::init

Registrieren Sie die Delegates der Audible-Klassen zu allen Modell-Klassen mittels `logic_factory.registermodule<...>()`.

Aufgabe 5.7 – Kür (3 Bonus-Punkte)

Hiermit haben wir das Ende der Übung der Veranstaltung „C++-Programmierung für Computergraphik“ erreicht. Wir hoffen, dass wir alles was wir vermitteln wollten auch vermitteln konnten und dass Sie Vor- und Nachteile der Programmiersprache C++ kennengelernt haben. Als letztes möchten wir ihnen die Möglichkeit geben noch Bonuspunkte zu sammeln indem Sie das kleine Spiel was in diesem

Aufgabenblatt entstanden ist erweitern. Diese Bonuspunkte können nur Punktabzug aus den anderen Teilaufgaben dieses Übungsblatts ausgleichen und nicht Punktabzug aus anderen Übungsblättern.

Was genau Sie für ihre Bonuspunkte machen möchten ist völlig ihnen überlassen. Die Anzahl der Punkte wird bei der Abnahme durch den Tutor abgeschätzt. Basis hierfür ist der geschätzte Aufwand der Umsetzung, was nicht mit dem Aufwand zusammenhängen muss, den Sie für diese Lösung wirklich aufgewendet haben. Wir sind aber gerne bereit ihnen die Punkte zu geben, wenn wir sehen, dass Sie sich mit einem Thema oder einer Idee auseinandergesetzt haben.

Mögliche Idee für Ihre eigenen Erweiterungen:

- Weitere Spielobjekte: Zusätzlich zu den Boxen könnten Sie auch Bonus-Gegenstände in das Spiel einbringen. Diese könnten sich zwar wie die Boxen verhalten, können aber vom Spieler durch das Paddle aufgesammelt werden. Diese Bonus-Gegenstände können Punkte bringen oder Effekte auslösen (z.B. temporäre Änderung der Größe des Paddles, das Entfernen aller aktuellen Boxen oder Änderung der Parameter der Boxen, wie Erscheinungsgeschwindigkeit)
- Weitere graphische Elemente: Sie könnten z.B. auch die graphische Darstellung des Spiels weiter aufwerten. Der Hintergrund könnte durch bewegte Objekte dynamischer gestaltet werden. Dies hat keinen Einfluss auf das Spiel selber würde aber die Graphik ansprechender machen. Anstatt des statischen Korridors könnte z.B. sich das gesamte Spiel (optisch) durch diesen Korridor bewegen. Sie könnten auch die Geometrie der Boxen ersetzen und anstatt statischer Boxen dynamische Objekte zeigen, z.B. Wassertropfen die sich aufgrund ihrer Bewegung von ihrer neutralen Kugelform her verformen.
- Weitere graphische Effekte: Die graphische Darstellung kann auch durch weitere Effekte aufgewertet werden. Beispielsweise könnten Sie mit einem einfachen Partikelsystem bei der Kollision zweier Boxen oder dem Herausfallen einer Box eine Art „Explosion“ zeigen. Alle Graphikobjekte könnten auch Schatten auf den Korridor der Welt werfen.

Diese Liste ist jedoch nur als Anregung gedacht. Zur Klarstellung: Sie entscheiden selber was genau sie machen möchten. Es reicht wenn Sie eine gute, umfangreiche Erweiterung implementieren um alle Bonuspunkte zu erhalten.

Viel Erfolg und viel Spaß!