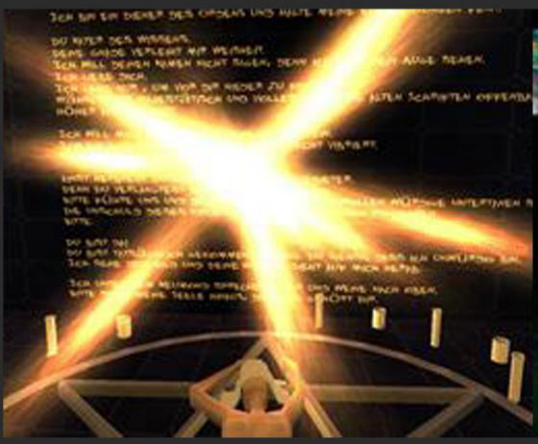
Johannes Winkler
contact@it-winkler.com

Portfolio



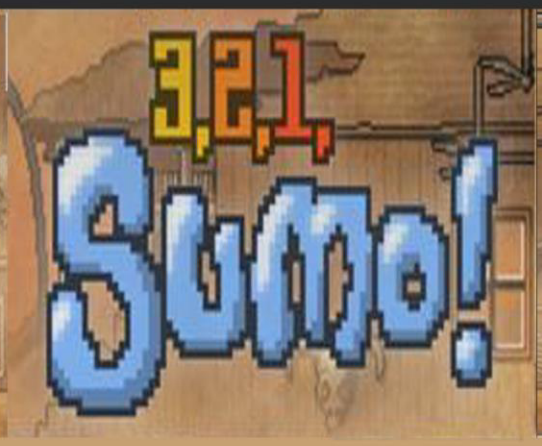
Titel:	The House
Genre:	Horror-Adventure (3D)
Engine:	Unity3D
Arbeitsbereiche:	Programmierung und Level-Design des gesamten Protoypen
Teamgröße:	2
Auszeichnung:	Demo Day WS 2014/2015 der TU München

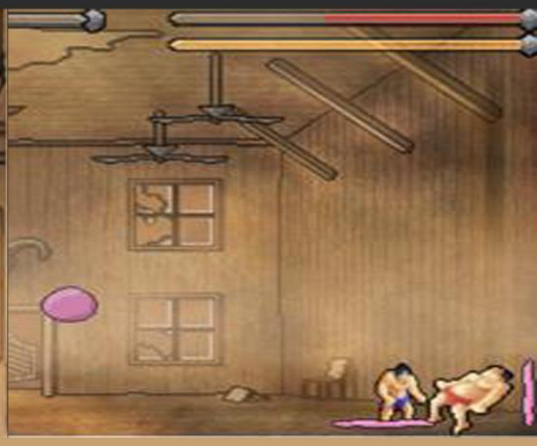
Konzept:

„Was wäre, wenn dich ein Dämon in deiner eigenen Traumwelt gefangen hält?“

Link zum Prototype: goo.gl/WDc0Gn

Trailer: goo.gl/o0zjA5





Titel:	Peacemaker
Genre:	Arcade-Fighter (2D)
Engine:	Unity3D
Arbeitsbereiche:	Programmierung und Game-Design des gesamten Protoypen
Teamgröße:	1

Besonderheiten:

- 3 Spieler Arcade-Fighter
- Asymmetrisches Spielprinzip (zwei Kämpfer + „Tripod-Spieler“)
- Online-Modus

Link zum Prototype: goo.gl/IyVPgG

Erster Online-Build: goo.gl/uLqCpx

Titel:	Outerspace
Genre:	Space-Shooter (2D)
Engine:	Eternity-Engine (selbst geschrieben)
Arbeitsbereiche:	Programmierung des GUI-Frameworks sowie weitere Teile der Engine (KI & Trigger-System)
Teamgröße:	2

Besonderheiten:

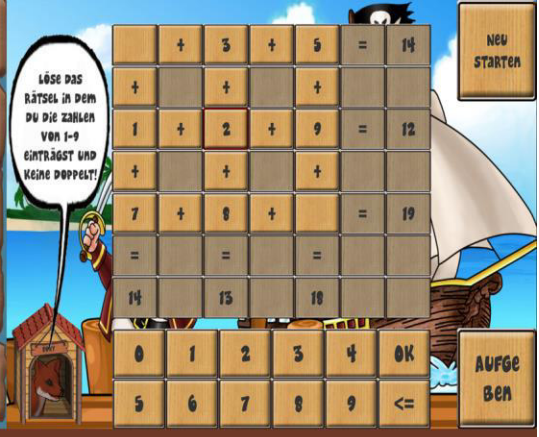
- Selbst geschriebene Engine auf Basis von SFML in C++
- Standalone GUI-Framework

Link zum Prototype: goo.gl/nYZpdc

Titel:	Batoru
Genre:	Location based social mobile game
Engine:	Android SDK
Arbeitsbereiche:	Teile der Programmierung der Client-Server-Kommunikation sowie das „Score&Elo“-Systems
Teamgröße:	5
Auszeichnung:	Demo Day SS 2015 der TU München
Link zur APK:	goo.gl/i8wDDA

Idee:

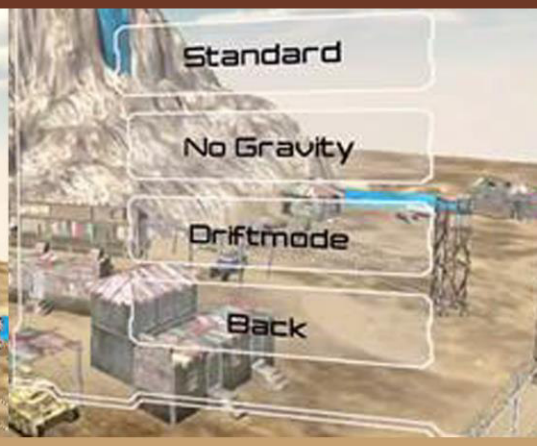
- „Hide&Seek“-Prinzip auf der Grundlage der echten Position des Spielers
- Spieler jagen einander in der echten Welt



Titel: Mathefuchs
Genre: Mathematik Lernspiel (2D)
Engine: Unity3D
Arbeitsbereiche: Teile der Programmierung der Rechenrätsel sowie Teile des Interface-Designs
Teamgröße: 8
Auszeichnung: Demo Day WS 2015/2016 und Schülertag 2016 der TU München
Link zum Prototype: goo.gl/Ckyz2b

Idee:

Mobile Lernspiel zur Unterstützung des Mathematikunterrichts in der Grundschule/Mittelstufe



Titel: TAAANKS!
Genre: Coop-Arcade-Action (3D)
Engine: Unity3D
Arbeitsbereiche: Programmierung und Game-Design der Schützen-Steuerung und des Score-Systems
Teamgröße: 3

Trailer: goo.gl/pzCIDv

Besonderheiten:

- Steuerung per „Kaktus"-Controller Slider zum Fahren, Rädchen und Buttons zum Schießen, LEDs zur Status-Anzeige

```
=begin
Folgende Annahme:
Es existiert eine unbestimmte Anzahl von Assets (technischen Anlagen). Jedes dieser Assets besteht aus 1 bis n Sub-Assets.
Jedes Sub-Asset hat entweder 1 ODER 4 verschiedene Zeitreihen, welche Auskunft über verschiedene technische Informationen geben, in Form
von float-Werten.

Die untenstehende Methode soll ein 1D-float-Array zurückgeben, in dem für ein Asset pro Zeitschritt ein Wert enthalten ist.
Wird zum Beispiel ein Zeitraum betrachtet, welcher 5 Zeitschritte lang ist, dann könnte eine mögliche Rückgabe folgendermaßen aussehen:
[1.0, 2.0, 3.0, 4.0, 5.0]

Es können folgende 3 Fälle eintreten:
1. Für den Fall, dass ein Asset nur aus einem Sub-Asset besteht und dieses nur 1 Zeitreihe besitzt, kann diese einfach zurückgeben werden.
2. Für den Fall, dass ein Asset nur aus einem Sub-Asset besteht und dieses 4 Zeitreihen besitzt, müssen diese Zeitreihen zu einer
"kombiniert" werden.
3. Für den Fall, dass ein Asset aus mehreren Sub-Assets besteht und diese jeweils 4 Zeitreihen besitzen, müssen die Zeitreihen zu einer
pro Sub-Asset und anschließend zu einer "gesamt" Zeitreihe "kombiniert" werden

Die Herausforderung an diesem Problem war, dass es festgelegt war, wie die Zeitreihen-Daten übergeben werden (1D-float-Array) und wie der
Output aussehen soll (ebenfalls 1D-float-Array).
Die Werte aus dem 1D-float-Array mussten nun möglichst effektiv und kompakt kombiniert werden.
Die ganze Methode ist recht kompliziert im Detail, was ein Manko ist. Aber ich finde sie ist eine ziemlich coole Variante mit relativ
wenigen Zeilen ans Ziel zu kommen,
was sie zu der kompaktesten Variante macht, die ich gefunden habe.

=end

#Methode kompakt, nur kurz kommentiert:
#Variante mit Beispielwerten und umfangreich kommentiert auf der nächsten Seite.
  def self.build_pav_array(rts_identifier, from_time, to_time, extended_formular, num_sub_assets, time_series)

    # Anfordern der Zeitreihen-Werte als 1D-float-Array
    pav_array       =   DailyEvent.get_time_series_as_float(rts_identifier, from_time, to_time, time_series)

    num_timestamps  =   pav_array.count / (num_sub_assets * time_series.count)
    pav_array       =   pav_array.each_slice(num_timestamps).to_a

    if extended_formular
      # Kombinieren der Zeitreihen-Werte für jeden Zeitschritt zu einem Wert pro Zeitschritt und Sub-Asset
      pav_array     =   pav_array.each_slice(time_series.count).to_a.
                            map{|ses_chunk, pav_chunk, avc_chunk, rl_chunk| pav_chunk.zip(rl_chunk, ses_chunk, avc_chunk).
                                map{|pav_element, rl_element, ses_element, avc_element| (pav_element + rl_element * ses_element ) *
avc_element}}
    end

    # Zusammenfassen der Werte jedes Sub-Asset zu einem Wert pro Zeitschritt
    return  pav_array.transpose.map {|element| element.reduce(:+)}
  end
```

```ruby
#Das untenstehende Beispiel ist für den kompliziertesten Fall 3.
#Beispielhafte Werte:
#   input:  rts_identifier      =>  "Asset1"
#           from_time           =>  t+0 (vereinfacht, normalerweise als Datetime z.B. 01.01.2016 00:00 +0100)
#           to_time             =>  t+2 (vereinfacht, normalerweise als Datetime z.B. 01.01.2016 00:03 +0100)
#           extended_formular   =>  true
#           num_sub_assets      =>  2
#           time_series         =>  [SES, PAv, AvC, RL]
#   output: [8.0, 8.2, 8.4]
def self.build_pav_array(rts_identifier, from_time, to_time, extended_formular, num_sub_assets, time_series)

    # Anfordern der Daten als float-Array
    # Die Daten kommen immer als ein großes 1D-Array zurück und sind nach Sub-Asset und nach Zeitreihen-Typ(beide absteigend) sortiert
    # Die "Herausforderung" war es an dieser Stelle das Array "korrekt" zu bearbeiten, da die Daten nicht anders "beschafft" werden können
    # SES, PAv, AvC und RL sind die Bezeichnungen der Zeitreihen, ihre Bedeutung ist nicht relevant für dieses Beispiel
    pav_array       =   DailyEvent.get_time_series_as_float(rts_identifier, from_time, to_time, time_series)
    #     | Sub-Asset1                                                || Sub-Asset2                                                |
    #     |t+0 t+1  t+2| t+0  t+1  t+2| t+0  t+1  t+2| t+0  t+1  t+2||t+0  t+1  t+2| t+0  t+1  t+2| t+0  t+1  t+2| t+0  t+1  t+2|
    #     |    SES     |      PAv     |      AvC     |      RL      ||     SES     |      PAv     |      AvC     |      RL      |
    # => [0.0, 0.0, 0.0, 5.0, 5.1, 5.2, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 3.0, 3.1, 3.2, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0]

    # Zu Testzwecken einfach die obere Zeile auskommentieren und die Zeile unten benutzen:
    # pav_array = [0.0, 0.0, 0.0, 5.0, 5.1, 5.2, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 3.0, 3.1, 3.2, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0]


    # Hier wird die Anzahl der Werte pro Zeitreihe ermittelt.
    # Jede Zeitreihe hat zwar gleich viele Elemente, aber allein aus der from_time und to_time lässt sich die Anzahl nicht ablesen.
    # Da die Auflösung der Zeitschritte unterschiedlich sein kann, z.B. minutenweise oder auch stundenweise (ein Wert pro Minute bzw.
      ein Wert pro Stunde)
    num_timestamps  =   pav_array.count / (num_sub_assets * time_series.count)
    # => 3

    # Es wird nun jede Zeitreihe in einen eigenen Block/Chunk "zerschnitten"
    # Da jede Zeitreihe in diesem Beispiel 3 Werte beinhaltet, werden 3er Blöcke/Chunks gebildet
    pav_array       =   pav_array.each_slice(num_timestamps).to_a
    #     | Sub-Asset1                                                  || Sub-Asset2                                                  |
    #     |t+0  t+1  t+2|  t+0  t+1  t+2 | t+0  t+1  t+2|  t+0  t+1  t+2 || t+0  t+1  t+2| t+0  t+1  t+2 |  t+0  t+1  t+2| t+0  t+1  t+2 |
    #     |     SES     |        PAv     |       AvC    |        RL      ||      SES     |        PAv    |       AvC     |        RL     |
    # => [[0.0, 0.0, 0.0],[5.0, 5.1, 5.2],[1.0, 1.0, 1.0],[0.0, 0.0, 0.0],[0.0, 0.0, 0.0],[3.0, 3.1, 3.2],[1.0, 1.0, 1.0],[0.0, 0.0, 0.0]]
```

```ruby
if extended_formular

    # Nun werden die Zeitreihen-Chunks aufgeteilt in Chunks pro Sub-Asset
    pav_array     =   pav_array.each_slice(time_series.count).to_a
    #  | Sub-Asset1                                                    ||  Sub-Asset2                                                  |
    #  |   t+0  t+1  t+2|  t+0  t+1  t+2|  t+0  t+1  t+2|  t+0  t+1  t+2  ||t+0  t+1  t+2   |t+0  t+1  t+2| t+0 t+1 t+2| t+0  t+1  t+2 |
    #  |     SES        |       PAv     |       AvC     |        RL       ||      SES        |   PAv        |    AvC     |      RL       |
    # =>[[[0.0, 0.0, 0.0],[5.0, 5.1, 5.2],[1.0, 1.0, 1.0],[0.0, 0.0, 0.0]],[[0.0, 0.0, 0.0],[3.0,3.1,3.2],[1.0,1.0,1.0],[0.0,0.0,0.0]]]

    # Jetzt muss für jeden Zeitschritt ein Wert pro Sub-Asset errechnet werden
    # Also in unserem Beispiel muss am Ende ein 2D-Array herauskommen
    # Je ein Wert pro Zeitschritt und pro Sub-Asset also 2x3 Werte ([[5.0, 5.1, 5.2], [3.0, 3.1, 3.2]])
    # Das Komplizierte hierbei ist, dass die Zeitreihen z.B. nicht einfach aufsummiert werden können,
    # sondern mit einer speziellen Formel zusammen gerechnet werden müssen:
    # Formel: (PAv + RL * SES) * AvC

    # map ist ähnlich zu einer for-each-loop aus anderen Programmier-Sprachen
    # Es werden nun zuerst "Zeitschritt-Chunks" gebildet, das bedeutet, je ein Wert aus jeder Zeitreihe eines Sub-Assets wird zu einem
      Chunk zusammengefasst
    # Z.B. aus [[SES_1, SES_2], [PAv_1, PAv_2]], ... wird [[SES_1, PAv_1, ...], [SES_2, PAv_2, ...]]
    # Die untenstehenden drei Code-Zeilen sind eine, sie wurden der übersichthalber mit Absätzen getrennt
    pav_array     =   pav_array.map{|ses_chunk, pav_chunk, avc_chunk, rl_chunk| pav_chunk.
      zip(rl_chunk, ses_chunk, avc_chunk).
    #      | Sub-Asset1                                                  || Sub-Asset2                                                  |
    #      |       t+0        | |        t+1       | |      t+2         |  ||       t+0        | |         t+1     ||       t+2        |
    #      |PAv, RL,  SES, AvC| |PAv, RL,  SES, AvC| |PAv, RL,  SES, AvC|  ||PAv, RL,  SES, AvC| |PAv, RL, SES, AvC|| PAv,RL,SES,AvC|  |
    # => [[[5.0, 0.0, 0.0, 1.0],[5.1, 0.0, 0.0, 1.0],[5.2, 0.0, 0.0, 1.0]],[[3.0, 0.0, 0.0, 1.0],[3.1, 0.0,0.0, 1.0],[3.2,0.0,0.0,1.0]]]

    # Anschließend wird aus den 4 Werten eines "Zeitschritt-Chunks" ein einzelner Wert berechnet
      map{|pav_element, rl_element, ses_element, avc_element| (pav_element + rl_element * ses_element ) * avc_element}}
    #     | Sub-Asset1   || Sub-Asset2   |
    #     |t+0  t+1  t+2 || t+0  t+1  t+2|
    #     |   new PAv    ||    new PAv   |
    # => [[5.0, 5.1, 5.2], [3.0, 3.1, 3.2]]
  end

  # Zu guter Letzt können die Werte der Sub-Assets einfach pro Zeitschritt aufsummiert werden
  return  pav_array.transpose.map {|element| element.reduce(:+)}
  #    | Sub-Asset1+Sub-Asset2 |
  #    |t+0   t+1   t+2 |
  #    |     new PAv    |
  # => [8.0,  8.2,  8.4 ]
end
```

```
(*Das untenstehende OCaml Beispiel ist zur Überprüfung der Korrektheit von AVL-Bäumen (https://de.wikipedia.org/wiki/AVL-Baum)
Es ist recht kompliziert auf den ersten Blick, aber ich finde meine Variante als recht elegant und effizient,
da jeder Knoten im Baum nicht mehr als einmal besucht wird.
Die ersten Deklarationen und Funktionen sind nur Hilfs-Funktionen.
Die eigentliche Funktion beginnt auf Seite 2
Unten eingefügt befindet sich auch noch ein Beispiel Baum, um das Ganze zu testen.*)

(*Typen Deklaration eines AVL-Baumes
Er besteht aus einem leeren Leaf (Blatt) oder einer Node, welcher durch den Typen avl_tree_node repräsentiert wird.
avl tree node speichert einen Int-Wert, seinen Balance-Faktor und jeweils einen linken und rechten Teilbaum
Der Balance-Faktor gibt, welche Seite eines Baumes tiefer ist.
    -1 bedeutet die linke Seite ist um eine Stufe tiefer als die rechte
     2 bedeutet die rechte Seite ist um zwei Stufen tiefer als die linke
     0 bedeutet beide Seiten sind gleich tiefer*)
type avl_tree =
    Node of avl_tree_node
  | Leaf
and avl_tree_node =
{
  key : int;
  balance : int;
  left : avl_tree;
  right : avl_tree;
}

(*Hilfs-Funktionen zum Vergleich des Schlüssel am aktuellen Knoten mit dem seines Parent-Knoten,
  um auf inkonsistente Werte zu testen *)
let less_eq = fun x y -> x <= y;;
let greater_eq = fun x y -> x >= y;;
let equals  = fun x y -> x =  y;;

(*Hilfs-Funktion, um die Validierungs-Informationen der linke
und rechten Seite eines Teilbaumes zu einem Tupel zusammen zu fassen.
Darüber hinaus wird der Balance-Faktor des aktuellen Knotens berechnet*)
let merge_validation_tupels_with_balance lhs_tupel rhs_tupel =
    match lhs_tupel,rhs_tupel with
    (lhs_depth, is_lhs_valid), (rhs_depth, is_rhs_valid) ->
        ((Pervasives.max lhs_depth  rhs_depth), (*1*)is_lhs_valid && is_rhs_valid, rhs_depth-lhs_depth)
;;

(*Hilfs-Funktion zum Überprüfen der Korrektheit des Balance-Faktors des aktuellen Knotens
Der berechnete Balance-Faktor muss immer gleich dem sein, der schon im Knoten steht
und er darf nicht größer als 1 bzw. kleiner als -1 sein, ansonsten ist der Baum nicht korrekt.*)
let check_balance cur_balance calcd_balance =
    (*4.*)cur_balance = calcd_balance && (*5*)abs cur_balance <= 1
;;
```

```
(*(Haupt-)Funktion zur Überprüfung eines AVL-Baumes, ob er korrekt ist
Ein AVL-Baum ist dann korrekt, wenn alle der folgenden Bedingungen erfüllt sind:
 1. Alle Teilbäume sind valide
 2. Alle Schlüssel im linken Teilbaum sind höchstens so groß wie der Schlüssel des Wurzelknotens
 3. Alle Schlüssel im rechten Teilbaum sind mindestens so groß wie der Schlüssel des Wurzelknotens
 4. Die Balancierung des Baumes wird korrekt im Feld balance gespeichert
 5. Die Balancierung ist ein Wert zwischen -1 und 1

 Die Idee dieser Implementation ist die, dass jeder Knoten nur ein einziges Mal besucht wird
 und dann alle Bedingungen auf einmal überprüft werden, um die Methode möglichst effizient zu gestalten.

 Die grundsätzliche Funktionsweise ist folgende:
 A: Wenn der Schlüssel des aktuellen Knotens valide ist, also die 2. bzw. 3. Bedingung erfüllt,
 B: dann wird für jeden Knoten die innere Validierungs-Funktion rekursiv für jeweils seinen linken und rechten Knoten aufgerufen
 C: Dies wird solange rekursiv weitergeführt, bis jeweils das Ende eines Teilbaums erreicht wird, was durch ein Leaf (Blatt)
signalisiert wird
 D: Nun wird angefangen beim Blatt bei jedem Schritt ein Tupel bestehend aus einem Int- und einem Boolean-Wert zurückgegeben
    - Der Int-Wert gibt die aktuelle Tiefe des Teilbaumes an (begonnen bei 0) und der Boolean-Wert, ob der Teilbaum valide ist
 E: An jedem Knoten werden nun die beiden Rückgabe-Tupel des linken und rechten Teilbaums zu einem verrechnet (siehe
merge_validation_tupels_with_balance)
    und anschließend die Bedingungen 1., 4. und 5. überprüft
    F: Dies geschieht folgendermaßen:
      Sind der linke und rechte Teilbaum valide (der Boolean-Wert ihres Rückgabe-Tupel ist true) und ist
      Anhand der beiden Tiefen-Werte der linken und rechten Teilbäume wird der Balance-Faktor berechnet,
      ist dieser identisch mit dem gespeicherten Wert und ist er ein valider Balance-Wert (Int-Wert zwischen -1 und 1)
 G: Dies wird solange getan, bis die Rekursion wieder zum Ursprungsknoten zurückgekehrt ist, welcher das finale Tupel zurückgibt
 H: Aus diesem Tupel wird zu Letzt der Boolean-Wert extrahiert und von der Haupt-Funktion zurückgegeben, welcher nun angibt, ob der
übergebene AVL-Baum korrekt ist*)
let valid_avl avl_tree =
    match (let rec inner_valid subtree parent_key key_check_function =
        match subtree with
        Node(node) ->
            (*A*)if (*1./2.*)(key_check_function node.key parent_key)
                then
                    match (*E*)(*1*)merge_validation_tupels_with_balance ((*B*)inner_valid node.left node.key less_eq)
                                                                         ((*B*)inner_valid node.right node.key greater_eq)
                    with
                    depth, is_tree_valid, calcd_balance ->
                        (*D, hier erfolgt die Tupel-Rückgabe für Nodes*)
                        (depth+1, (*F*)((*4.&5.*)check_balance node.balance calcd_balance) && (*1*)is_tree_valid)
            else
                (0, false)
        | (*C*)Leaf -> (*D*)(0,true)
    in (*G*)inner_valid avl_tree (match avl_tree with Node(root) -> root.key) equals) with
    depth, is_tree_valid -> (*H*)is_tree_valid
;;
```