STATISCHE FORMEN & PARAMETER

Die erste Übung war ein statisches Bild, das zwei geometrische Formen darstellte: Linien und Ellipsen. Mit folgenden Befehlen werden sie aufgerufen:

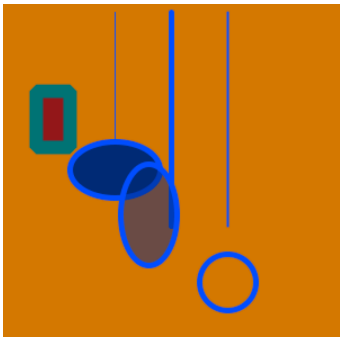
```
1    line(Startpunkt X Koordinaten, Startpunkt Y Koordinaten, Endpunkt X, Endpunkt Y);
2
3    ellipse(Position X, position Y, Breite, Höhe);
```

Setzen wir VOR die Befehle Ausdrücke wie:

```
1    strokeWeight(Strickdicke);
2
3    //oder
4
5    fill(Wert Rot, Wert Grün, Wert Blau, Wert Alpha);
```

können wir einzelne Parameter der Formen genauer definieren. Mit dem Ausdruck „smooth(Stärke des aniti-aliasing, e.g. 2, 4, 8) wird die Darstellung weniger "pixelig". Laut der Reference-Seite von Processing, ist diese Einstellung aber schon per "default" aktiviert. Ich veränderte die Verschiedenen werte der Funktion und fand noch weitere Formen und Parameter wie:

```
1    rect(PosX, PosY, Breite, Höhe); // ein Viereck
2    //oder
3    strokeJoin(MITER / BEVEL / ROUND) // definiert die Übergänge von Linien
```



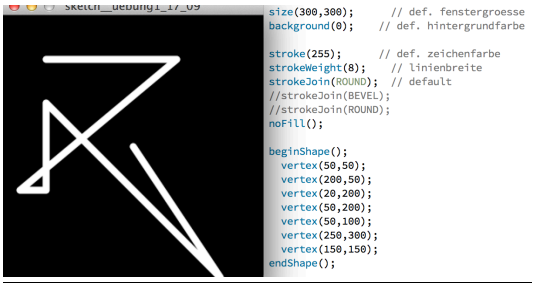
Notiert: Die Parameter einer Form (Farbe, Dicke, etc.) werden vor der Eingabe der Form definiert.
Wie beim Zeichnen mit Buntstiften: Zuerst wähle ich die Farbe und Dicke des Stifts, dann was ich damit zeichnen will.

Blogeintrag & Sourcecode:
http://blogs.iad.zhdk.ch/codingspace/2013/09/20/1-lektion-statische-formen-und-parameter/

VERTEX

Um etwas komplexere Formen als Quader, Linien oder Ellipsen zu kreieren, eignet sich die Funktion “vertex(x,y)” gut. Sie kreiert Linien, die jeweils dort beginnen, wo die vorangehende aufgehört hat - so können wir uns die Eingabe der Startkoordinaten sparen. Auch hier lasssen sich die Übergänge per “strokeJoin” verändern. Anschliessend schauten wir uns noch Bögen (arc) und die Funktion „begin/endShape()“ an.

Notiert: Eine Umfangreiche Bibliothek an Tutorials, Beispielen, Begriffserklärungen, Übungen und Templates findet man auf processing.org und andernorts im Internet.



```
size(300,300);      // def. fenstergroesse
background(0);    // def. hintergrundfarbe

stroke(255);      // def. zeichenfarbe
strokeWeight(8);    // linienbreite
strokeJoin(ROUND);  // default
//strokeJoin(BEVEL);
//strokeJoin(ROUND);
noFill();

beginShape();
  vertex(50,50);
  vertex(200,50);
  vertex(20,200);
  vertex(50,200);
  vertex(50,100);
  vertex(250,300);
  vertex(150,150);
endShape();
```

Blogeintrag & Sourcecode:
http://blogs.iad.zhdk.ch/codingspace/2013/09/20/vertex/

DRAWSOME!

Mit den Variablen “mouseX” und “mouseY” kreierten wir eine Ellipse, deren Position von den Koordinaten des Maus-Crusors abhängig ist.

Dazu verwendeten wir die Funktion “draw”. Sobald das Programm ausgeführt wird alles was sie umfasst fortlaufen – also 60 mal in der Sekunde – ausgeführt. Die Position der Maus wird also ständig neu berechnet und und auf jene der Ellipse übertragen. Dies könnten wir mit dem Befehl “noLoop()” auch verhindern, was wir in diesem Fall aber nicht wollen.

Ein Mitstudent half mir, eine neue Variabel zu definieren (“grow”) zu der – jedes mal wenn “draw” ausgeführt wird – der Wert 1 addiert wird. Dadurch vergrössert sich die Ellipse ins Unendliche sobald Processing den Code ausführt. Um dem Abhilfe zu verschaffen, setze ich die Variabel “sin” an Stelle von „+1“ ein. Nun nahm die Grösse

der Ellipse, kontinuierlich in der Sinuskurve eines vorgegebenen Wertes zu und ab. Des Weiteren lernte ich die Funktion mousePressed() kennen. Sie führt bei Mausklicks beliebig definierbare Befehle aus. In unserem Beispiel Fall nannte sie uns die aktuellen Koordinaten des Crusors.

Notiert: Im Vergleich zu "menschlichen" Sprachen, scheinen Programmiersprachen auf den ersten Blick zwar sehr komplex; sie sind dafür immer logisch nachvollziehbar und kennen kaum "irrationale" Ausnahmen. Wie bei jeder Sprache braucht man aber gewisse Basics um durch in der Anwendung seine Kenntnis selbstständig zu erweitern. Vorerst ist wohl noch eher "Voki büffle" angesagt.

Blogeintrag & Sourcecode:
http://blogs.iad.zhdk.ch/codingspace/2013/09/20/9325/

BEDINGUNGEN, MOUSEPRESSED() & SWITCH()

Ziel war es, ein einfaches Zeichenprogramm zu erstellen, das eine Auswahl an Farben, sowie eine Löschfunktion bietet.

Im Code definierte ich nach dem "void setup()" erstellte ich erst vier verschiedene Variabeln (x,y,z,q) denen ich später die gewünschten Farbwerte zuordnen konnte, sowie eine um alles bisherige zu überschreiben/löschen (d).

Nun ging's ums eigentliche Zeichnen; die Funktion "void draw()" und darin verschachtelte Bedingungen codierte ich wie folg (Beschreibung direkt im Code):

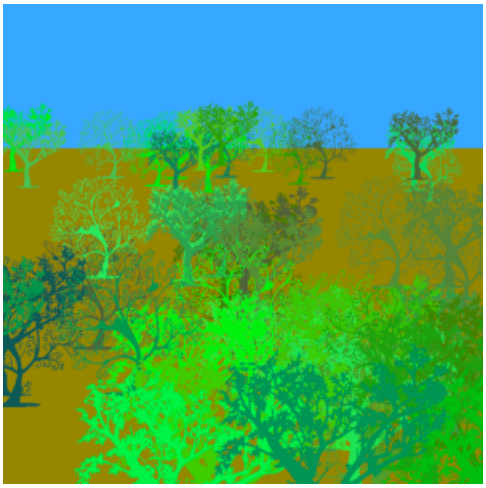
Notiert: Als ich den Code schrieb, war mir oft nicht ganz klar, was ich da genau tat. Es war sehr hilfreich mir das Ganze – im Rahmen dieser Dokumentation – nochmals Schritt für Schritt anzuschauen.

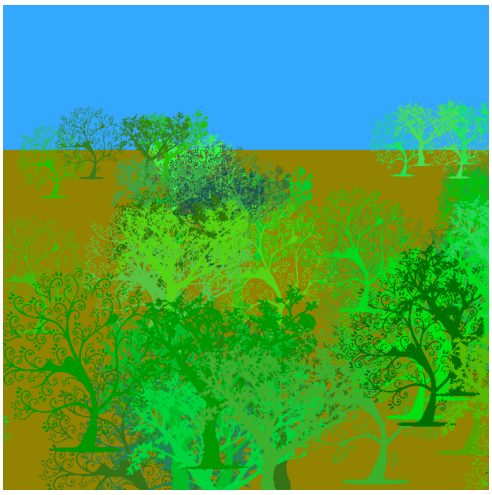
Blogeintrag & Sourcecode:
http://blogs.iad.zhdk.ch/codingspace/2013/09/23/bedingungen-mousepressed-switch/

LOADIMAGE, REDRAW() & RANDOM

Ziel: Ein “Waldgenerator” der durch Klicken der Maus einen unregelmässigen aber “kontrollierbaren” Wald erschafft.

Als erstes initialisierte ich ein paar globale Variabeln und erstellte ein “noLoop()” im “void setup” um eine ständige Wiederholung der Funktion zu verhindern. Anschliessend definierte ich eine neue Funktion die mir eine kleine Anzahl Bäume - in zufälliger Anordnung und Farbe, doch bestimmter Grösse - generieren sollte. Ich nannte sie "waldi".





Mit einer "if"-Bedingung im folgenden "void draw()" konnte ich nun die Reaktion auf Mausklicks bestimmen. Ausserdem verhinderte einer Bedingung innerhalb der bestehenden, jegliche Reaktion sofern der Wert mouseY kleiner als 150 ist. Da das Wiedeholen der Vorgänge unterbunden war [noLoop()], verwendete ich den Befehl "redraw" in der Funktion "mousePressed()" um bei jedem Klick "waldi()" erneut auszuführen.

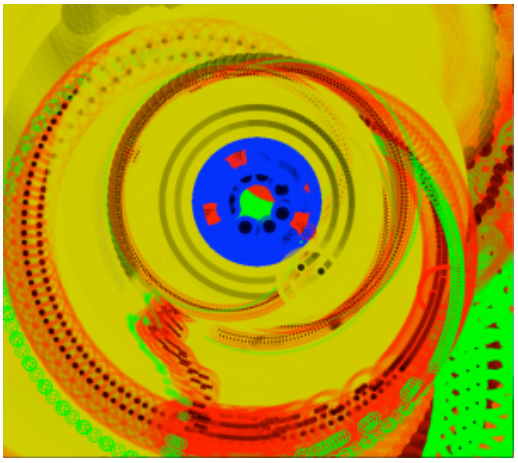
Notiert:
Beim Programmieren scheint sehr viele Wege zu geben dasselbe Problem zu lösen. Langsam leuchtet mir ein, weshalb es sich lohnt, den geplanten Lösungsansatz im Voraus durchzudenken.

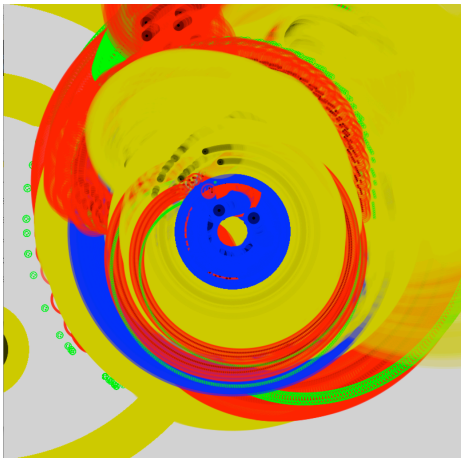
Blogeintrag & Sourcecode:
http://blogs.iad.zhdk.ch/codingspace/2013/09/23/loadimage-redraw-random/

PUSHMATRIX, PUSHSTYLE UND TRIGONOMETRIE

Ziel war es, das vorgefertigte Template eines – der Maus entsprechend rotierend und skalierendes – Smileys so zu verändern, dass es von seinem Ebenbild fortlaufen umkreist wird.

Erst mal musst ich aus dem bestehenden Template schlau werden, was mir aber nur teilweise gelang. Was es mit dem "pushMatrix" auf sich hatte verstand ich ziemlich schnell, sie erlauben eine zeitweilige Neudefinition des Koordinatensystems bis zum Befehl "popMatrix". Die trigonometrischen Funktionen blieben mir weitgehend schleierhaft doch immerhin verstand ich, dass ich dies zum Lösen der Aufgaben gar nicht zu wissen brauchte. Es reichte, die "pushMatrix" des statischen Smileys richtig in jener des Grossen Smileys zu verschachteln und seine "rotation()" von einer zunehmenden Variabel abhängig zu machen, um den gewünschten Effekt zu erzielen. Euphorisiert von den neuen Möglichkeiten verschachtelte ich weitere Smiley-Funktionen in die bestehenden und fügte einen Button zum Ändern der Richtung und Geschwindigkeit ein (Key '1' und '2'). Toll. Aus Neugier löschte ich dann den "background()" und gab den Smileys verschiedene halbtransparente Farben – nun wurden die Spuren der kreisenden Objekte sichtbar.





Notiert:

Es lassen sich mit wenig Kenntnis schon viele interessante Dinge simulieren. Mit etwas mehr Kenntnis bestimmt auch relevantere Dinge als eine "Wahrnehmung unter Drogeneinfluss", welche meine Smileys wiederzugeben scheinen.

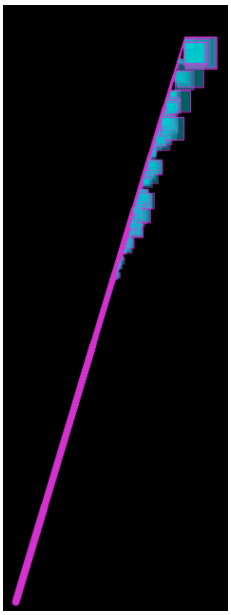
Blogeintrag & Sourcecode:

http://blogs.iad.zhdk.ch/codingspace/2013/09/23/pushmatrix-pushstyle-und-trigonometrie/

REKURSIVE FUNKTIONEN – WURZELMURX

Rekursive Funktionen sind solche, die – wenn sie ausgeführt werden – sich selbst aufrufen. Da dies alleine aber auf eine Endlosschleife hinausläuft, ist es notwendig diese Wiederholung zu begrenzen. Wir definieren also, wie weit sich die Rekursion fortsetzen, und was danach geschehen soll. In unserem Beispiel, waren es die Äste eines baumähnlichen Konstrukts, die sich mehrfach selbst aufriefen und, bei Erreichen einer gewissen Anzahl Verzweigungen, in einer Ellipse endeten.

Aufgabe war es das Template zu verändern und damit neue Darstellungen zu erreichen. Die tat ich fleissig, scheiterte jedoch sehr oft und was mich besonders störte: Meistens fand ich bis zuletzt nicht heraus warum und lernte deshalb nicht sehr viel aus meinen Fehlern. Mit gemässigten Ambitionen, setzte ich mir kleinere Ziele welche ich eher erreichen konnte. Ich wandte die zuvor erlernte Methode (siehe “loadImage, redraw & random”) an um Bilder von Politikern am Ende jedes Asts erscheinen zu lassen.



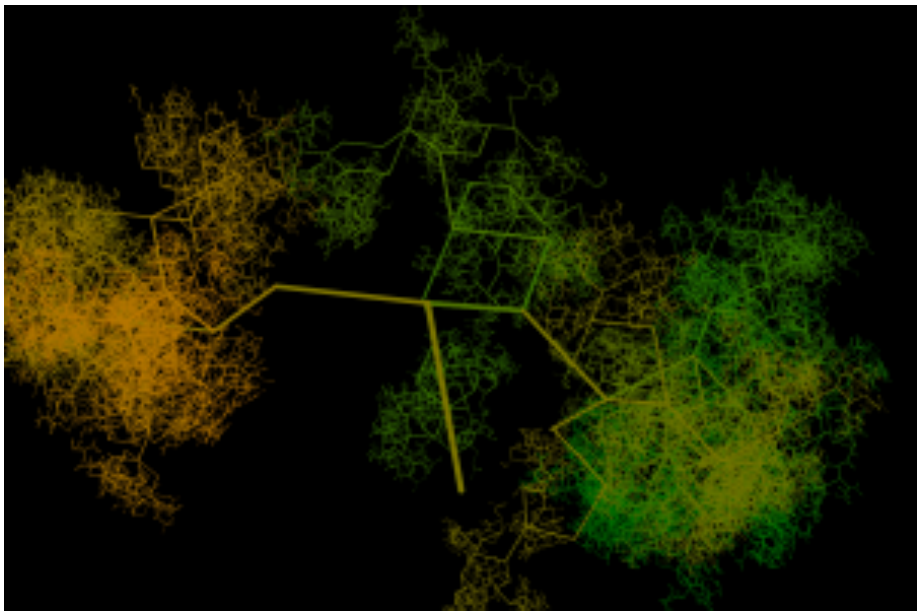
Mein Ziel nächstes Ziel war es die Abweichewinkel der Äste zu vereinheitlichen, wozu ich aber Werte generieren musste, die mit einer 50%-Chance den einen oder anderen Wert (und NUR diesen) annahmen. David riet mit zu folgender Codezeile:

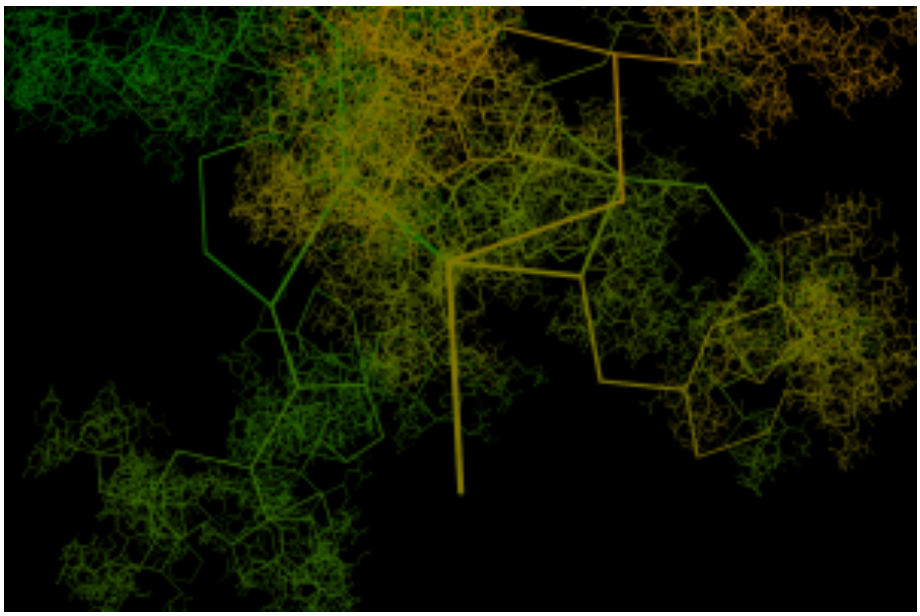
1.<code>

2.(random(0, 1)>0.5) ? 200 : -200;

Von da an ging es zügig voran und ich verbrachte noch sehr viel Zeit mit fraktalen Spielereien. Meine Faszination galt besonders der Eigendynamik - die ich zwar beeinflussen - aber nie gänzlich kontrollieren konnte.

Als ich auf einer Zugreise dabei war, fünfeckige fraktale Muster mit bestimmten Farbverläufen zu generieren, wurde ich allen Ernstes von einem Chemielaboranten darauf angesprochen, ob ich mich schon lange mit „der Struktur von Calcium“ befasse. Welch glorreicher Tag: Ich habe nicht nur die gewünschte Ästhetik erreicht, sondern das Vorkommen fraktaler Muster in der Naturwissenschaft täuschend echt imitiert.





Blogeintrag & Sourcecode:
http://blogs.iad.zhdk.ch/codingspace/2013/09/24/wurzelmurx-rekursive-funktionen/

VEKTOREN – ELEGANZ DANK ALLIANZ

Eine Übung zum Thema Vektoren – diese verstand ich in der Theorie sehr gut, ihre Anwendung in der Praxis blieb mir aber schleierhaft. Deshalb arbeitete ich eng mit Jonas zusammen, der hier den Durchblick hatte. Ich schaute zuerst wie er das Problem anging und folgte seinem Beispiel als es darum ging den Anfangs- und Endpunkt der Linie per Mausklick festzulegen. Er zeigte mir, wie man mit einem „clickcount" den ersten Mausklick vom Zweiten unterscheiden konnte, was ich in späteren Übungen erneut anwandte.

Als es darum ging, die Bewegung des Balls, bei Erreichen des Endpunktes, umzukehren, schaute ich ebenfalls auf seinen Bildschirm und mir fiel mir auf, dass Jonas dazu eine ziemlich umständliche Methode anwandte. Ich begann nach einfacheren Wegen zu suchen was mir – zu meinem eigenen Erstaunen – nach kurzem Tüfteln selbstständig gelang. So konnte ich mich umgehend für seine Hilfe revanchieren, „die eine Hand wusch die andere" und wir waren beide ab der eleganten Lösung erfreut. Hier seine Sicht der Dinge.

Erkenntnis: Aus den Fehlern anderer zu lernen, bringt fast so viel wie sie selber zu begehen.

Blogeintrag und Sourcecode:
http://blogs.iad.zhdk.ch/codingspace/2013/10/01/vektoren-eleganz-dank-allianz/

BREAK OUT – FREI ZUM ABSCH(L)USS

Wir erhielten ein Template des Arcade-Klassikers „Break Out" und die Aufgabe, dieses nach unseren Vorstellungen und Möglichkeiten zu neu gestalten.

Mein oberstes Ziel war es, das altbekannte Gameplay abwechslungsreicher und komplexer zu machen. Mir war bewusst, dass dies meine Fähigkeiten schnell übersteigen konnte – der Code war mir von Anfang an schon komplex genug. Abwechslungsreichere Interaktionen und „aktivere Gegner" (als statische Klötze) konnte ich alleine noch nicht programmieren. Mit einer simplen aber wirksamen Methode versuchte ich, dem Spieler trotzdem ein meist intelligent und komplex reagierendes Gegenüber zu schaffen: Ein zweiter User sollte ihn herausfordern können (fortan „Player2"). Dieser sollte den Angriffen des ehemaligen Einzelspielers

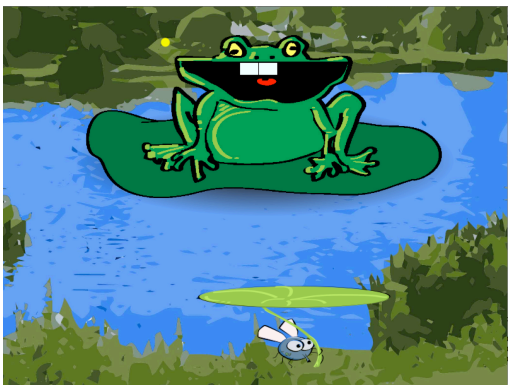
(„Player1“) ausweichen und ihm den Sieg streitig machen. Als Eingabegerät nutzte ich aus – aus Mangel an Kenntnis und Alternativen - schlicht die Pfeiltasten des Keyboards.

Zu meinem Erfreuen gelang es mir, die Grundfunktion des Ausweichens in „Eigenregie“ zu schreiben. Einzig für das kontinuierliche Fahren bei gedrückt gehaltener Pfeilaste musste ich David um Rat Fragen. Es war mir grundsätzlich sehr wichtig, dass ich alle Programmierungen selbstständig umsetzten - oder notfalls zumindest durchschauen - konnte.

Das Spiel hatte vorerst noch das Problem, dass sich ein findiger/feiger Player2 unversehrt aus der Bildfläche bewegen und wieder zurückkommen konnte, um den Schüssen auszuweichen. Nach unzähligen Fehlschlägen programmierte ich unabsichtlich eine Lösung die zwar nicht optimal, dafür aber wirksam und für mich umsetzbar war: Wenn Bricks den Bildrand überschritten, wurden sie gelöscht. Eine konsequente Bestrafung für das Ausnutzen eines Bugs.

Erst jetzt machte ich mir Gedanken über die „Story“ des Spiels und deren grafische Umsetzung. Dieses Vorgehen ist bei mir - und bei richtigen Programmierern wohl auch – weder üblich noch sehr sinnvoll. Der Grund dafür war, dass ich der grössten Herausforderung – dem Ändern des Gameplays – hohe Priorität gab; solange ich nicht wusste, ob ich den gewünschten Spielmodus überhaupt hinkriegte, war es sinnlos dessen grafische Umsetzung zu planen.

Ich interpretierte die ursprünglichen Bricks als Zähne und die beiden Spieler – da sie sehr unterschiedliche Aufgaben hatten – als zwei ungleiche Gegner: Einen Frosch und eine Fliege. Wie mir später auffiel, war diese Idee sicher nicht der Weisheit' letzter Schluss. Zwar habe ich die Entscheidung nicht sehr schwer gewichtet und eher willkürlich gefällt, wie ich aber bei der Darstellung der Bricks als Zähne ausgerechnet auf einen Frosch kam, konnte ich mir schon in der Umsetzungsphase nicht mehr erklären. Passend zu der beschränkten Bedienung (2 Spieler an 1 PC) und der fragwürdigen Story (eben), wählte ich den grafischen Stil früher „in-browser games“ (e.g. Bubble Trouble, Gold Miner, etc.).



Als ich das Spiel mit Klassenkameraden testete, merkte ich bald, dass hier mit ungleichen Spiessen gekämpft wurde: Noch immer war es Player1 fast nicht möglich die „Zähne“ des Froschs zu treffen, da sich der Ball nur sehr begrenzt Steuern liess. Um seine Waffe trickreicher zu machen, gab ich Player1 die Möglichkeit den Spielball durch Mausklicks im Flug anzuhalten. Doch wie sich herausstellte reichte das nicht, also beschloss ich anstelle eines Stopps, einen Richtungswechsel als Reaktion auf Mausklicks zu programmieren. Wie bei fast allen Änderungen die ich vornahm, verbrachte ich auch bei dieser sehr viel Zeit damit, den Code zu analysieren um die entscheidende Funktion zu finden. Trotz allem gelang es mir leider nicht, die Bewegung umzudrehen, da ich es hier mit Vektoren zu tun hatte welche ich nicht durch simples „*-1“ ins Negative kehren konnte. Ich erhielt Hilfe und muss eingestehen, dass ich wohl noch lange nicht selbstständig auf die richtige Lösung gekommen wäre. Ebenso wenig, schaffte ich es den letzten gravierenden Bug aus dem Programm zu kriegen: sobald alle „Froschzähne“ zerstört waren, sollte sich ein Bild mit Glückwünschen an die Fliege öffenen, was aber nicht geschah.

Ich versuchte mit einer „count“ Funktion die Anzahl der übrigen Zähne zu eruieren, diese gab aber oft unsinnige Werte an. Die Zeitlimite und mein Ziel möglichst vieles selbst zu programmieren, verhinderten dass ich das Problem noch lösen konnte. Ich vermute aber dass es mit dem zuvor programmierten „Löschen der Zähne bei Verlassen des Spielfelds“ zusammenhängt. Dies ist wohl der Preis, den man für meine „unsauberen doch zweckmässigen“ Lösungen bezahlen musste.

Blogeintrag & Sourcecode:

http://blogs.iad.zhdk.ch/codingspace/2013/10/07/breakout-abschlussprojekt/