

Aufgabe 2

33.Bundeswettbewerb Informatik 2014/'15 2.Runde

Tim Hollmann

30. März 2015

1 Aufgabe 1

1.1 Theorie

Am Anfang der Bearbeitung dieser Aufgabe lies mich bereits beim Lesen der Aufgabenstellung folgende Passage stützen:

Die Kegel sind punktförmig, und jeder wird, unabhängig von den anderen, an einem Punkt platziert, der zufällig und gleichmäßig aus der gesamten Kreisscheibe gezogen wird.

Wie ist diese „gleichmäßige“ Verteilung möglich? Eine Gleichmäßigkeit der Punkte setzt doch eine Position eines Punktes in Relation zu den anderen Punkten voraus; unabhängig voneinander kann man doch gar nicht gleichmäßig verteilen. Es bestand die Möglichkeit, dass sich diese Gleichmäßigkeit nicht direkt auf die Verteilung der Punkte bezog, sondern z.B. auf den Schwerpunkt; dass der Schwerpunkt der Punkte immer genau auf der Kreismitte liegt. Oder bezog sie sich auf den Zufall; alle möglichen Punkte auf der Kreisscheibe müssen gleich(mäßig) wahrscheinlich sein? Dies unterstützt auch der folgende Satz aus der Angabe:

Genauer wählt Randy vor jedem seiner Kegelwürfe einen zufällig und gleichmäßig verteilten Winkel v [...] und einen zufällig und gleichmäßig verteilten Punkt x [...].

Da bei diesem Winkel und diesem einzelnen Punkt keine Menge in Relation stehen kann, bezieht sich „gleichmäßig“ wohl auf den Zufall (beziehungsweise auf die durch den Zufall theoretisch entstehende gleichmäßige Verteilung). Ich habe diese Frage auch in EI-Community-Forum gestellt, bekam aber genau wie bei meinem anderen Beitrag „nur“ einen freundlichen Hinweis auf den „BwInf-Dreisprung“.

Hinzu kommt noch ein weiteres Problem: die gleichmäßige Wahrscheinlichkeit der Punktverteilung führt nur theoretisch auch zu einer gleichmäßigen Verteilung der Punkte auf dem Spielfeld. Um Ballungsgebiete zu vermeiden habe ich die Aufgabe erweiternd durch einen minimalen Abstand der Punkte zueinander, der nie unterschritten werden

darf, nachgeholfen. So sind die Punkte nicht nur theoretisch, sondern auch in der Praxis gleichmäßig verteilt.

Das Problem besteht hierbei in der Ermittlung dieses minimalen Abstandes.

Dazu habe ich mir zuerst angeguckt, welchen Abstand die Punkte maximal zueinander haben können. Der maximale Abstand ist der zweifache Radius von so vielen Kreisen im Spielfeld wie Kegeln, mit dem der Spielfeld-Kreis gerade so gefüllt werden kann;

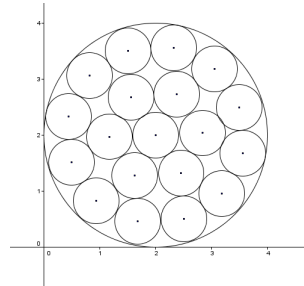


Abbildung 1: Füllen des Spielfeldes mit Kreisen zur Ermittlung des maximalen Abstandes

Der Abstand zwischen zwei Punkten beträgt dann an den engsten Stellen $2r$;

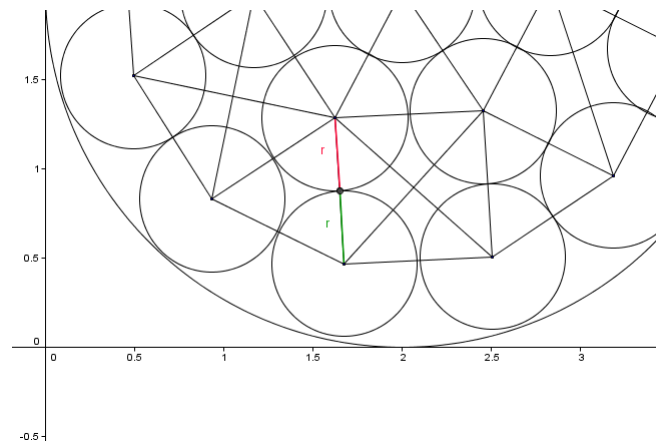


Abbildung 2: An den engsten Stellen beträgt der Abstand $2r$

Wäre der Radius r nur minimal größer, würden die Kreise nicht mehr hineinpassen, sodass der maximale Punktabstand $2r$ ist. Also gilt $0 < \text{minimaler Punktabstand} < 2r$. Hier kommt es darauf an, was man als „gleichmäßig verteilt“ empfindet. Ich persönlich finde einen Abstand von $1r$ für Gleichmäßigkeit ausreichend. Aber wie errechnet man nun r ?

1.1.1 Errechnen des minimalen Kegelabstandes

Dazu ermittelt man zunächst die Fläche des Grundkreises anhand seines Radius

$$\text{Kreisfläche} = \pi \cdot r^2$$

Dann teilt man diese Fläche durch die Anzahl der Kegel, um die Fläche pro Kegel zu erhalten.

$$\text{Fläche pro Kegel} = \frac{\text{Kreisfläche}}{\text{Anzahl der Kegel}} = \frac{\pi \cdot r^2}{\text{Anzahl der Kegel}}$$

Dann ermittelt man den Radius des Kreises, der genau diese Fläche hätte. Dafür formt man zunächst die allgemeine Flächenformel nach r um und setzt dann ein;

$$\begin{aligned} \text{Kreisfläche} &= \pi \cdot r^2 && | : \pi \\ \frac{\text{Kreisfläche}}{\pi} &= r^2 && | \sqrt{} \\ \sqrt{\frac{\text{Kreisfläche}}{\pi}} &= r \end{aligned}$$

Einsetzen:

$$\text{minimaler Kegelabstand} = \sqrt{\frac{(\pi \cdot r^2) / \text{Anzahl der Kegel}}{\pi}}$$

Anmerkung: Man muss hierbei aber beachten, dass dieser errechnete Radius nur eine Annäherung ist, da zwischen den Kreisen noch Platz ist, der nicht von Kreisen benutzt werden kann aber trotzdem mitgerechnet wurde. Eigentlich wäre der Radius deshalb minimal kleiner, dies kann aber vernachlässigt werden

1.1.2 Erzeugen eines zufälligen Punktes

Um einen Punkt zufällig auf der Kreisscheibe zu positionieren, habe ich zuerst einen zufälligen Winkel $0 \leq \alpha \leq 360$ und einen zufälligen Abstand zur Kreismitte $0 \leq r \leq \text{Kreisradius}$ erzeugt. Da r nie größer als der Kreisradius sein kann, kann der Punkt auch nicht außerhalb des Kreises liegen (vgl. Abb. 3).

Um nun die Koordinaten dieses Punktes zu ermitteln, zeichnet man daraus eine Gerade mit Länge r und Steigung α ausgehend vom Kreismittelpunkt und dann eine horizontale auch aus der Kreismitte und verbindet diese Linien durch eine Senkrechte auf die horizontale; so bekommt man ein Dreieck mit 90° -Winkel. (siehe Abb. 4). Die Länge der Hypotenuse ist bekannt (r); die Länge der Ankathete ist die x -Koordinate und die Länge der Gegenkathete die y -Koordinate.

Da gilt

$$\sin(\alpha) = \frac{\text{Gegenkathete}}{\text{Hypothense}}$$

und

$$\cos(\alpha) = \frac{\text{Ankathete}}{\text{Hypothense}}$$

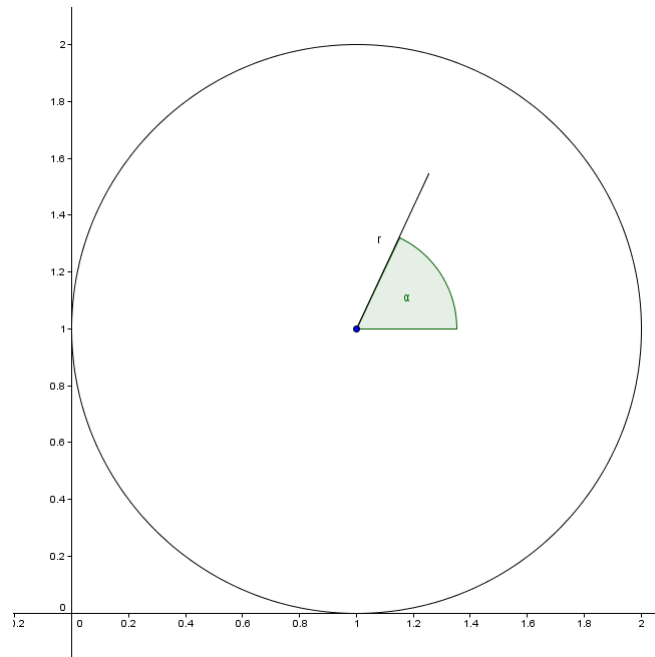


Abbildung 3: Erzeugung eines zufälligen Punktes; Winkel α und Abstand r .

und die Länge der Hypotenuse (r) und der Winkel α bekannt ist, können die Gegenkathete und die Ankathete errechnet werden, indem man die Formeln danach umstellt und ausrechnet:

$$\begin{aligned} \text{Gegenkathete} &= \sin(\alpha) \cdot \text{Hypotenuse} && = y\text{-Koordinate} \\ \text{Ankathete} &= \cos(\alpha) \cdot \text{Hypotenuse} && = x\text{-Koordinate} \end{aligned}$$

Die x - und y -Koordinaten sind dann aber noch relativ zum Kreismittelpunkt; addiert man den Radius hinzu, werden die Koordinaten absolut (bzw. relativ zum Koordinatenursprung), denn bei mir befindet sich der Kreismittelpunkt im Koordinatensystem auf dem Punkt $(r|r)$, wenn r der Radius des Kreises ist.

1.2 Umsetzung

Ausschlaggebend für die Wahl der Programmiersprache war bei dieser Aufgabe natürlich die Grafikfähigkeit. In den engeren Auswahlkreis kamen deshalb vor allem Java und Visual Basic. Schlussendlich habe ich mich aber für JavaScript entschieden, welches mit der neuen HTML5-Canvas-Umgebung zu sehr guten grafischen Ausgaben in der Lage ist, ohne JFrames, paint-Methoden, APIs oder ähnliches bemühen zu müssen. Von Vor- und Nachteilen von JavaScript wie Plattformunabhängigkeit oder Offenlegung des Quellcodes soll aber hier keine Rede sein.

Grob besteht der Aufbau des Skriptes aus drei Funktionen:

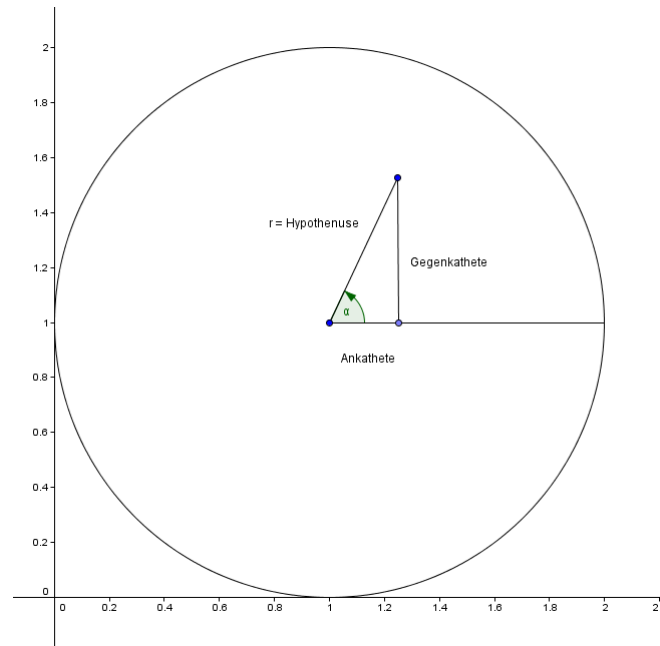


Abbildung 4: Es lässt sich ein rechtwinkliges Dreieck konstruieren.

1. Funktion zur zufälligen Generierung der Kegel-Punkte (`createRandomPoints`)
2. Funktion zur grafischen Ausgabe der Punkte und des Kreises (`draw`)
3. Hauptfunktion, die die beiden Funktionen nacheinander aufruft und selbst beliebig oft aufgerufen werden kann, um jeweils neue Punkte zu erzeugen (`main`)

1.2.1 Die *main*-Funktion

Die *main*-Funktion kann beliebig oft aufgerufen werden; sie generiert dann immer neue Punkte und verwirft, falls vorhanden, die bestehenden Kegel.

Bei jedem Start der *main*-Funktion werden folgende Einstellungen geladen:

- Anzahl der Kegel
- Radius des Kreises
- Grafische Optionen:
 - Farbe des Kreises
 - Farbe der Kegel
 - Größe der Kegel

Die Einstellungen werden per `document.getElementById("<id>")` aus den Eingabefeldern des HTML-Formulars gelesen und dann ggf. zu INTs geparkt.

Dreh- und Angelpunkt der grafischen Ausgabe ist das HTML5-Objekt `canvas`:

```
<canvas id="canvas" width="300" height="300">..</canvas>
```

Dieses Element wird ebenfalls per `document.getElementById("canvas");` ermittelt (siehe Z.15).

Da die Zeichenfläche eine feste Größe hat (in diesem Fall 300x300 Pixel) und der Radius des Kreises variabel sein soll, verwende ich einen vom Kreisradius abhängigen Darstellungsfaktor, mit dem die Koordinaten der Punkte und des Kreises verrechnet werden. So passt jeder beliebige Kreisradius auf die konstante Zeichenfläche. Je größer der Kreisradius, desto kleiner der Darstellungsfaktor; die Grafik wird auf die Größe der Zeichenfläche skaliert. Dieser Faktor wird in der *main*-Funktion errechnet (siehe Z.27).

In der *main*-Funktion wird daraufhin die Funktion *createRandomPoints* aufgerufen, die ein Array mit zufällig und gleichmäßig verteilten Kegelpunkten zurück gibt.

Anschließend wird die Funktion *draw* aufgerufen, um die soeben erzeugten Punkte mitsamt Kreis auf der Zeichenfläche darzustellen. Die Funktion füllt die Tabelle mit den Koordinaten rechts von der Zeichenfläche automatisch mit.

```
12 function main(){
13
14     /* Zeichenfläche und ihren 2D-Kontext laden */
15     var canvas = document.getElementById("canvas");
16     var ctx = canvas.getContext("2d");
17
18     /* Sonstige Einstellungen laden */
19     var anzahlKegel = ←
20         parseInt(document.getElementById("anzahlKegel").value);
21     var kreisRadius = ←
22         parseInt(document.getElementById("kreisRadius").value);
23
24     var kreisFarbe = document.getElementById("kreisFarbe").value;
25     var kegelFarbe = document.getElementById("kegelFarbe").value;
26     var kegelRadius = ←
27         parseInt(document.getElementById("kegelRadius").value);
28
29     /* Darstellungsfaktor anhand der Einstellungen errechnen */
30     var faktor = canvas.width/(2 * kreisRadius);
31
32     /* Punkte generieren */
33     var kegel = new Array(createRandomPoints(kreisRadius, anzahlKegel));
34
35     /* Kreis mit Punkten zeichnen */
36     draw(canvas, ctx, faktor, kreisRadius, kegel, kreisFarbe, kegelFarbe, ←
37         kegelRadius);
38 }
```

Aufrufen der *main*-Funktion

Die *main*-Funktion wird bei jedem Seitenaufruf automatisch ausgeführt:

```
<body onLoad="javascript:main();">...</body>
```

Das selbe geschieht bei jedem Klick auf den Button „Neue Kegel generieren“. Sie können die *main*-Funktion sowie alle anderen Funktionen auch direkt manuell über die Web-Konsole ihres Browsers aufrufen:

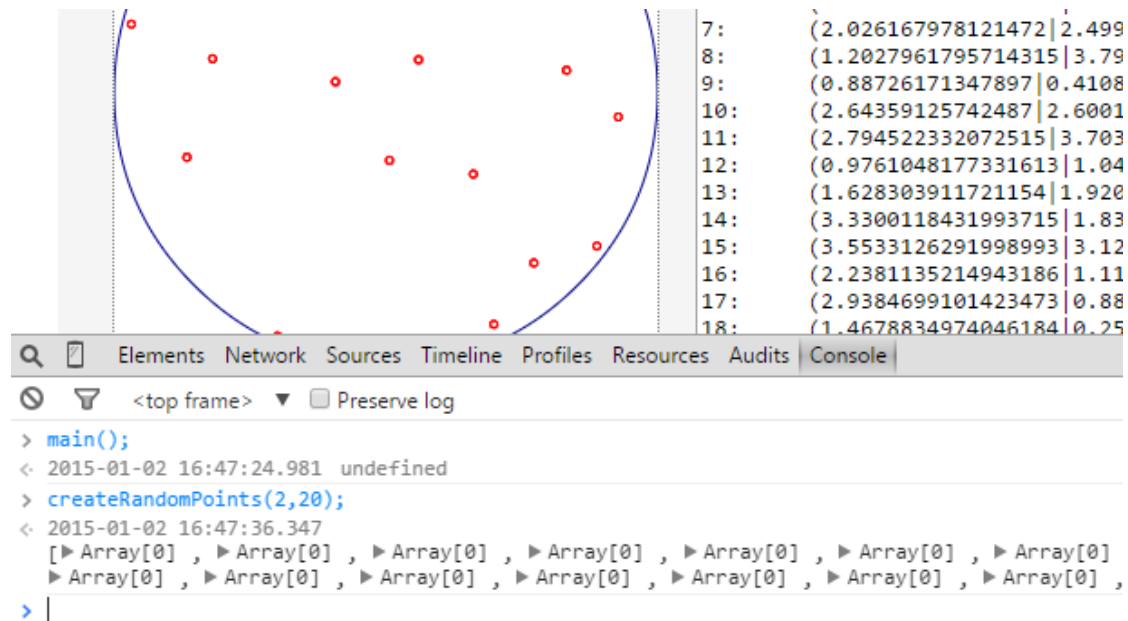


Abbildung 5: Die Webkonsole erreicht man sowohl unter Chrome als auch unter Firefox mit **Ctrl** + **Shift** + **J** / **K**. Sie erlaubt es, während der Laufzeit Funktionen auszuführen, Variablenwerte auszulesen und zu setzen oder Laufzeitanalysen zu machen.

1.2.2 *createRandomPoints*

Diese Funktion ist für die zufällige und gleichmäßige Zeugung von Kegelpunkten zuständig; sie übernimmt als Parameter den Radius des Kreises und die Anzahl der zu erzeugenden Kegel und gibt diese als Array zurück.

Zuerst wird genau nach (1.1.1) die Fläche des Kreises, die Fläche pro Kegel und schließlich der minimale Kegelabstand errechnet. (vgl. Z. 84-94). Dieser minimale Abstand wird schon vor der *for*-Schleife errechnet, da er für alle zu erzeugenden Punkte gleichermaßen gilt.

```
87 /* Kreisfläche ausrechnen */
88 var kreisflaeche = Math.PI * kreisRadius * kreisRadius;
89
```

```
90  /* Kreisfläche pro Kegel */
91  var flaecheProKegel = kreisflaeche / anzahlKegel;
92
93  /* Minimaler Abstand berechnen */
94  var minimalerAbstand = Math.sqrt(flaecheProKegel / Math.PI);
```

Anschließend werden per *for*-Schleife nach 1.1.2 so viele Punkte erzeugt, wie angefordert wurden.

```
96  for(var x = 0; x < anzahlKegel; x++){
97      var ok = false;
98
99      while (ok != true){
100
101          /* zufälliger Abstand zur Kreismitte */
102          var abstand = random(1,100)/(100/kreisRadius);
103
104          /* zufälliger Winkel */
105          var winkel = random(1,360);
106
107          /* Winkel in Bogenmaß umrechnen (wird von den Sinus- und ↵
            Kosinus-Funktionen benötigt) */
108          var winkelBogen = (winkel * (Math.PI / 180));
109
110          /* Errechnen, wo die x- und y-Koordinaten lägen */
111          var xPos = kreisRadius + (Math.cos(winkelBogen) * abstand);
112          var yPos = kreisRadius + (Math.sin(winkelBogen) * abstand);
113
114          ok = true;
115
116          /* Alle bereits vorhandenen Punkte durchgehen, ob ihr Abstand zum ↵
            aktuellen Punkt niedriger ist als der Minimalwert. */
117          for(var i = 0; i < tempArrayKegel.length; i++){
118              var xAbstand = Math.abs(tempArrayKegel[i]["x"] - xPos);
119              var yAbstand = Math.abs(tempArrayKegel[i]["y"] - yPos);
120
121              /* Abstände berechnen (Satz des Pythagoras) */
122              var abstand = Math.sqrt(xAbstand * xAbstand + yAbstand * ↵
                yAbstand);
123
124              if (abstand < minimalerAbstand){ ok = false; }
125          }
126      }
127
128  }
129
130  /* Die Abstände sind OK -> Punkt aufnehmen */
131  var temp = new Array();
132  temp["x"] = xPos;
133  temp["y"] = yPos;
134
135  tempArrayKegel.push(temp);
136
```


137
138

}

Für jeden neuen Punkt wird wie in 1.1.2 vorgegangen; Erzeugen eines zufälligen Abstandes zum Mittelpunkt (Z.102) und eines zufälligen Winkels (Z.105). Dann mit Kosinus und Sinus die x - und y -Koordinate errechnen (Z.111+112). Allerdings arbeiten die Funktionen `Math.sin` und `Math.cos` mit Winkeln im Bogenmaß. Deshalb wird der im Gradmaß erzeugte Winkel `winkel` gemäß der Formel

$$radian = degree \cdot \frac{\pi}{180}$$

ins Bogenmaß umgerechnet (vgl. Z.108).

Dann werden alle bereits erzeugten Punkte durchgegangen und überprüft, ob ihr Abstand zum aktuell erzeugten Punkt niedriger als der Minimalwert ist. Dazu ermittle ich zunächst Δx und Δy unter Verwendung der Betragsfunktion `Math.abs()` (Z.119f.).

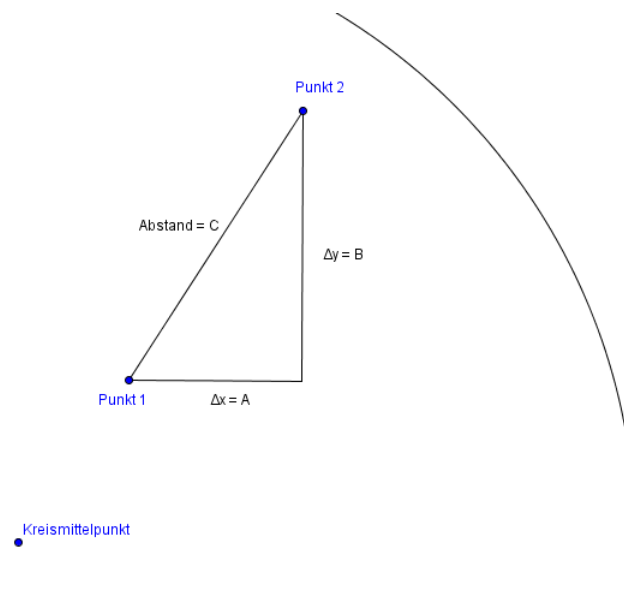


Abbildung 6: Um zwei beliebige Punkte lässt sich ein rechtwinkliges Dreieck konstruieren, dessen Katheten Δx , Δy und der Abstand entsprechen.

Den Abstand der Punkte errechne ich dann mit dem Satz des Pythagoras (Z.123);

$$\sqrt{\Delta x^2 + \Delta y^2} = \text{Punktabstand}$$

123

```
var abstand = Math.sqrt(xAbstand * xAbstand + yAbstand * yAbstand);
```

Wenn dann dieser Abstand bei irgendeinem Punkt kleiner als `minimalerAbstand` ist, wird `ok` auf `false` gesetzt und die zufällige Erzeugung des Punktes durch die `while`-Schleife in

Zeile 99 wiederholt (vgl.Z.125). Andernfalls wird der Punkt in das Array `tempArrayKegel` aufgenommen, welches am Ende der Funktion zurückgegeben wird;

```
140 return tempArrayKegel;
```

Die Gleichmäßigkeit der Punkteverteilung belegt auch folgendes Bild, bei dem 5000 Kegel platziert wurden: (Abb.7)

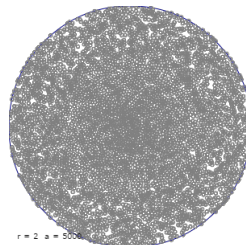


Abbildung 7: Die Verteilung der Kegel ist gleichmäßig mit zufälliger Varianz

1.2.3 draw

Schließlich „zeichnet“ die Funktion `draw` die Punkte mitsamt Kreis auf die Canvas-Zeichenfläche. Zusätzlich trägt sie die Koordinaten der Kegel in eine Tabelle neben der Zeichenfläche ein. Der Radius des Kreises sowie die Anzahl der Kegel werden zudem in die untere linke Ecke des Canvas geschrieben.

```
36 function draw(canvas, canvasContext, faktor, kreisRadius, kegel, ↵
37     kreisFarbe, kegelFarbe, kegelRadius){
38
39     /* Variablen für die Beschriftung des Canvas */
40     var textabstand = 10;
41     var fontsize = 10;
42
43     /* Die Punkte und ihre Koordinaten werden zusätzlich noch in eine ↵
44        Tabelle eingetragen;
45        diese wird ermittelt, geleert und mit der Überschrift gefüllt */
46     var kegelBox = document.getElementById("kegelBox");
47     kegelBox.value = "Kegel:";
48
49     kegel = kegel[0];
50
51     /* Zeichenfläche leeren */
52     canvasContext.clearRect(0, 0, canvas.width, canvas.height);
53
54     /* Den Kreis zeichnen */
55     canvasContext.beginPath();
56     canvasContext.strokeStyle = kreisFarbe;
57     canvasContext.arc(faktor * kreisRadius, faktor * kreisRadius, faktor * ↵
58         kreisRadius, 0, 2*Math.PI);
```

```

57     canvasContext.stroke();
58
59     /* Die Punkte aus dem Array auslesen und zeichnen */
60     for(var i = 0; i < kegel.length; i++){
61
62         canvasContext.beginPath();
63         canvasContext.strokeStyle = kegelFarbe;
64         canvasContext.moveTo(faktor * kegel[i]["x"] + kegelRadius, faktor *
        * kegel[i]["y"]);
65         canvasContext.arc(faktor * kegel[i]["x"], faktor * kegel[i]["y"],
        kegelRadius, 0, 2*Math.PI);
66         canvasContext.stroke();
67
68         /* Die Punkte und ihre Koordinaten werden zusätzlich noch in eine
        Tabelle eingetragen */
69         kegelBox.value += "\n" + (i+1) + ": (" + kegel[i]["x"] + "|" +
        kegel[i]["y"] + ")";
70
71     }
72
73     /* Unten in die Darstellungsfläche die groben Informationen schreiben:
        Kreisradius und Kegelanzahl */
74     canvasContext.beginPath();
75     canvasContext.font = fontsize + "px Verdana";
76     canvasContext.fillText("r = " + kreisRadius + " a = " + kegel.length,
        10, canvas.height - textabstand);
77     canvasContext.stroke();
78
79 }

```

Ähnlich wie bei nahezu allen Bildbearbeitungsprogrammen gibt es bei Canvas eine Art „Pinsel“, den man über die gesamte Zeichenfläche bewegen kann und der entweder zeichnet, oder eben nicht. Dieser Pinsel wird per `beginPath()` geladen und per `moveTo()` auf dem Bild bewegt. Zeichnen kann Canvas Linien (`lineTo()`), Rechtecke (`rect()`) und Kurven (`arc()`). Hat man gezeichnet (oder auch nicht), setzt man den Pinsel mit `stroke()` ab, die Änderungen werden übernommen und das Canvas dargestellt.

Per

```

54     canvasContext.beginPath();
55     canvasContext.strokeStyle = kreisFarbe;
56     canvasContext.arc(faktor * kreisRadius, faktor * kreisRadius, faktor *
        kreisRadius, 0, 2*Math.PI);
57     canvasContext.stroke();

```

wird der Kreis gezeichnet. Hier müssen die Koordinaten - wie bereits erwähnt - mit dem Darstellungsfaktor `faktor` multipliziert werden, um die Grafik auf die Zeichenfläche zu skalieren.

Der Rest der Funktion ist weitestgehend selbstverständlich.

1.2.4 Das HTML-Gerüst

JavaScript selbst hat keinerlei Möglichkeiten zur Ausgabe von irgendwelchen Informationen (wenn man von `alert` und `console.log` einmal absieht). Stattdessen muss es auf Elemente des ihm zugeordneten HTML-Dokumentes zugreifen. Dabei spielt es keine Rolle, ob diese bereits vorhanden sind; per `document.createElement()` können beliebige Elemente erzeugt werden. Ich verwende ein HTML-Gerüst, in dem sich bereits Elemente befinden, da sich von Javascript erzeugte Elemente nur mit Mühe in ein derartiges Design verschachteln lassen würden. Im Wesentlichen besteht das Gerüst aus

- Einem Canvas-Zeichenelement

```
<canvas id="canvas" width="300" height="300">...</canvas>
```

- Einem Textarea-Feld, das zur Ausgabe der Kegelkoordinaten verwendet wird

```
<textarea id="kegelBox" style="width: 500px; height: 300px;"></textarea>
```

- Zwei Buttons zum erneuten Generieren von Kegeln und Download des Canvas

```
<input type="button" class="button" value="Bild Speichern" ↵  
  onClick="javascript:saveCanvas()" />  
<input type="button" class="button" value="Neue Kegel generieren" ↵  
  onClick="javascript:main();" />
```

- Mehreren Eingabeboxen für die Anzahl der Kegel, den Kreisradius und andere Einstellungen

```
<!-- Anzahl der Kegel -->  
<input value="20" id="anzahlKegel" placeholder="Anzahl der Kegel" />  
<!-- Radius des Kreises -->  
<input value="2" id="kreisRadius" placeholder="Radius des Kreises" />  
<!-- Farbe des Kreises -->  
<input value="darkblue" id="kreisFarbe" placeholder="Farbe des ↵  
  Kreises" />  
<!-- Farbe der Kegel -->  
<input value="red" id="kegelFarbe" placeholder="Farbe der Kegel" />  
<!-- Radius der Kegel-Kreise -->  
<input value="2" id="kegelRadius" placeholder="Radius der ↵  
  Kegel-Kreise" />
```

- Ein CSS-Stylesheet, das für das Layout verantwortlich ist.

```
<!-- Stylesheet einbinden -->  
<link href="style_1.css" type="text/css" rel="stylesheet" />
```

1.2.5 Weitere Features

Anpassen der Zeichnung Über die Eingabefelder auf der unteren Hälfte der Seite haben Sie die Möglichkeit, neben Änderungen am Spielaufbau auch Änderungen an der Darstellung des Spielfeldes vorzunehmen. Bei allen Farbfeldern können sowohl Farben auf Englisch, als auch Hexadezimal kodiert eingegeben werden.

Wenn Sie die Kegel lieber als Punkte anstatt als Kreis dargestellt haben wollen, geben Sie bei „Radius der Kegel-Kreise“ den Wert 1 ein. (Es wird zwar trotzdem ein Kreis gezeichnet, da er aber einen Radius von nur einem Pixel hat, wird er von Canvas als ein einzelnes Pixel gezeichnet.)

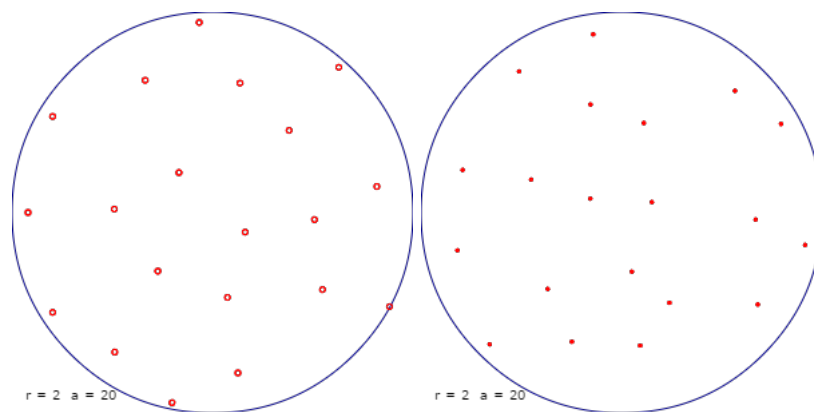


Abbildung 8: Einmal mit Kreisen und einmal mit Pixeln

Speichern der Grafik Das Abspeichern der erzeugten Grafik ist sehr leicht möglich; klicken Sie einfach mit der Rechten Maustaste auf das Canvas-Element und wählen Sie „Grafik speichern unter“. Dies ist mit jedem Browser möglich, denn jeder HTML5-fähige Browser muss zumindest das Speichern als PNG unterstützen. Ich habe zusätzlich eine Funktion geschrieben, die das Canvas auf Knopfdruck direkt herunterlädt, allerdings ohne Dateieindung.

1.2.6 Starten des Skriptes

Dazu öffnen Sie das HTML-Dokument „1.html“ mit einem Browser ihrer Wahl, der HTML5-Fähig ist (also mit jedem außer dem Internet Explorer).

1.3 Beispiele

⇒ (Abb.10).

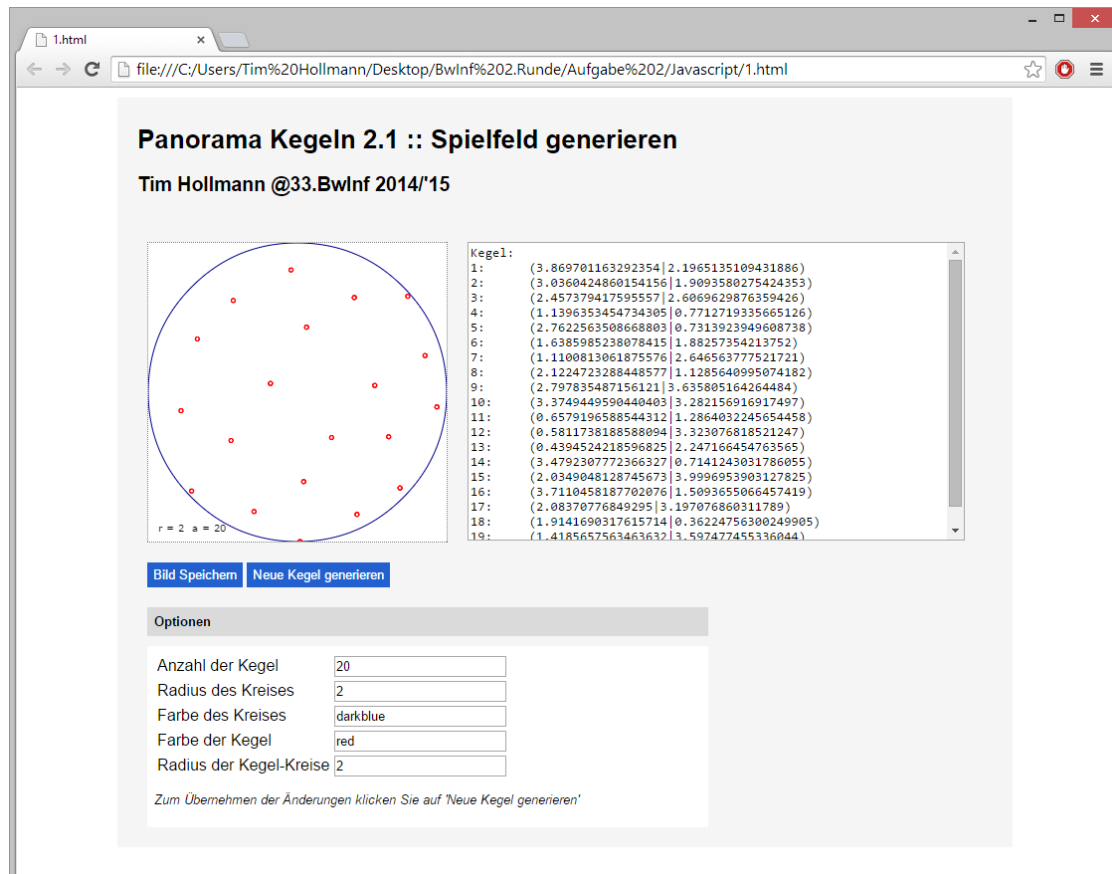


Abbildung 9: So sollte es aussehen, wenn Ihr Browser HTML unterstützt und JavaScript aktiviert hat.

2 Aufgabe 2

Ziel dieser Unteraufgabe ist es

- Das **simulieren des gesamten Spielablaufes** bei gleichzeitiger
- stetiger und nachvollziehbarer **grafischer Visualisierung** für den Benutzer, und (darin verbundene)
- **Interaktionsmöglichkeiten** zur Durchführung eines Wurfes.

2.1 Grundlegendes zur Programmstruktur

Für die Interaktion mit dem Benutzer ist es logischer Weise nötig, auf die Eingaben des Benutzers zu reagieren. Da ich JavaScript verwende, geschieht dies mit event-Handlern; eigenständigen Funktionen die bei einem Event (z.B. `click`) ausgeführt werden. Folglich

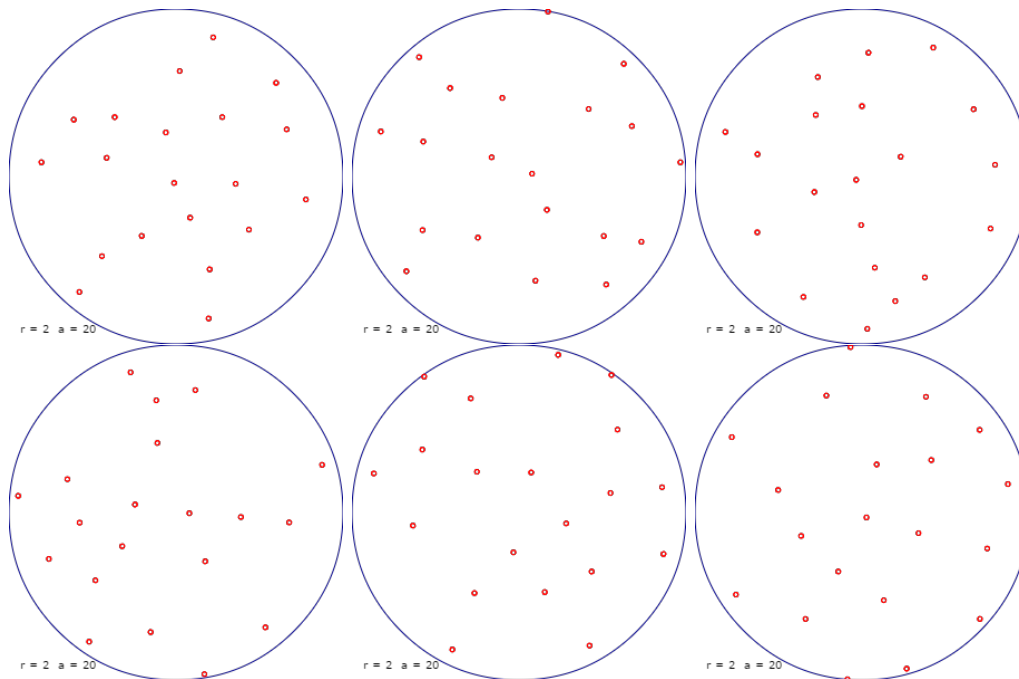


Abbildung 10: Beispiele

ist es innerhalb einer Kontrollstruktur nicht möglich, auf die Eingabe durch den Benutzer zu warten, wodurch z.B. folgender Ansatz hinfällig wird:

```
function Spiel(){
  while(true){ /* Runden */
    createRandomPoints(); /* Ein Spielfeld generieren */

    while(KegelAufDemSpielfeld && !EntscheidungRundeBeenden){
      randyZug(); // Randys Zug
      annaZug(); // Annas Zug (Warten auf Eingabe innerhalb der ↔
        Funktion nötig!!)
    }
  }
}
```

Folglich muss der komplette Spielablauf dezentral um die event-Handler aufgebaut werden, woraus sich folgendes ergibt:

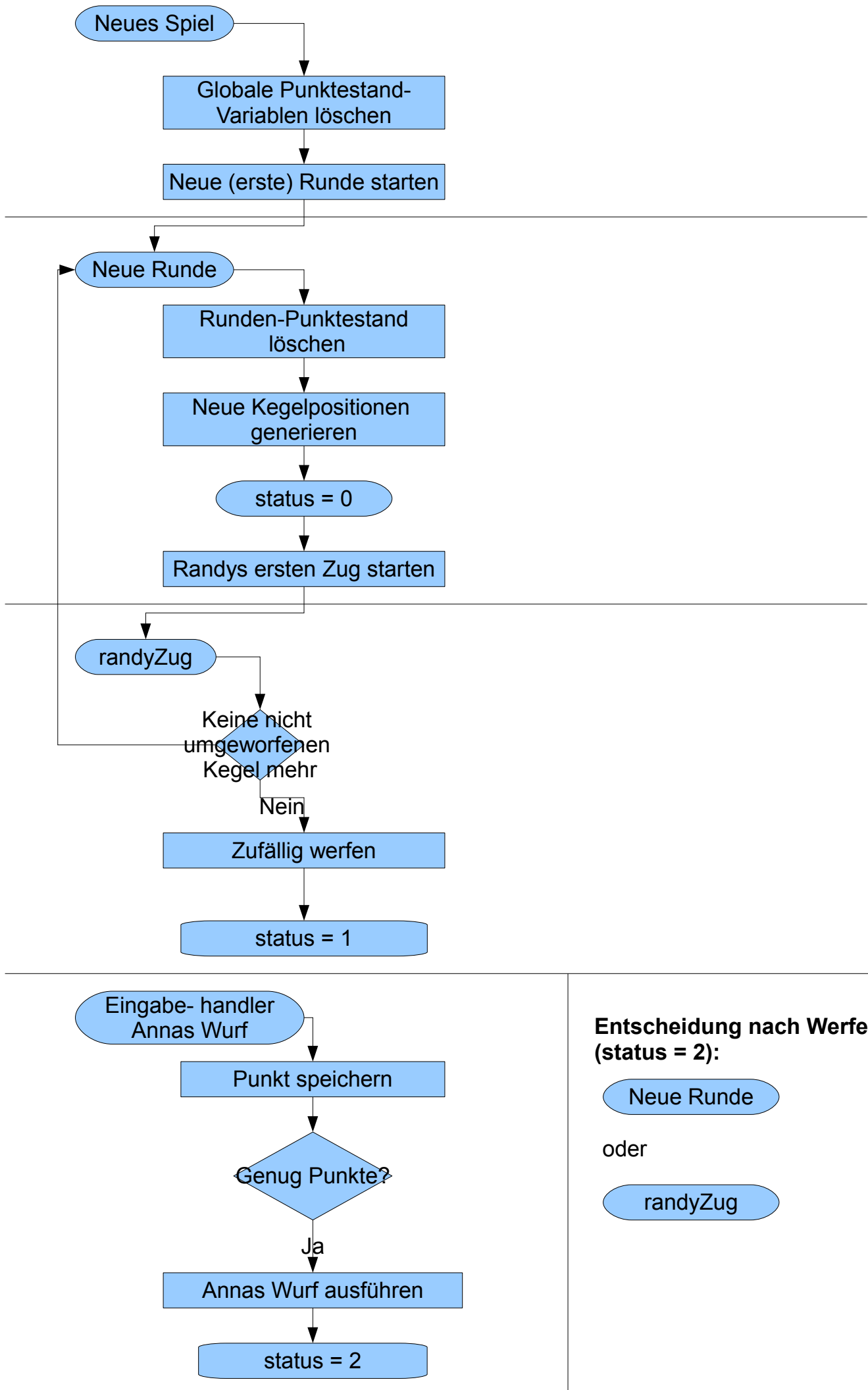
1. Verwendung globaler Variablen, da nun verschiedene Funktionen auf dieselben Variablen zugreifen müssen, wie z.B. die Kegelpositionen oder den Punktestand.
2. Aus den fehlenden Kontrollstrukturen ergibt sich die Notwendigkeit der globalen Indikator-Variable `status`, die den Spielablauf regelt. Sie kann je nach Phase des Spieles drei Werte haben, denn ich unterteile das Spiel in drei Phasen:

- a) `status = 0`: Randy zieht bzw. Anna ist nicht dran (wenn eine neue Runde oder das Spielfeld erstellt wird).
- b) `status = 1`: Anna ist dran mit Werfen, hat aber noch nicht (vollständig) geworfen.
- c) `status = 2`: Anna hat bereits geworfen und muss sich nun entscheiden, die aktuelle Runde evtl. zu beenden

Die durch ein Event (sprich: Eingabe des Benutzers) aufgerufenen Event-Handler-Funktionen überprüfen zunächst anhand dieser Indikator-Variable den Status des Spieles und damit, ob die Aktion, die durch das Event ausgeführt werden würde, in dieser Phase des Spieles überhaupt ausgeführt werden darf. Z.B. darf Anna nicht werfen, wenn sie nicht dran ist (`status = 0`), oder sie darf nicht über das Rundenende entscheiden, wenn sie noch nicht geworfen hat (`status = 1`).

Funktionen - zumindest die direkt an der Simulation des Spielablaufs beteiligten - führen also gemäß ihrer Funktionalität anhand der Informationen in den globalen Variablen Manipulationen an diesen aus - unter der Bedingung von `status`.

Kegel haben nun neben ihren x - und y -Koordinaten noch eine weitere (boolesche) Eigenschaft: `umgeworfen`.



Die Simulation beginnt mit einer Funktion, die die globalen Punktestand-Variablen zurücksetzt und eine neue Runde initialisiert. Die Runden-Funktion setzt den aktuellen Runden-Punktestand zurück und sichert davor ggf. einen bereits vorhandenen (die Funktion kann auch innerhalb des Spieles aufgerufen werden, um eine neue Runde zu beginnen). Daraufhin werden die neuen Kegel gemäß **1.1.2** generiert und global gespeichert. Da Randy immer den ersten Zug hat (den „Startvorteil“), wird seine Zug-Funktion direkt von der Funktion aufgerufen, die eine neue Runde startet.

2.2 Randys Zug

Da Randys Zug auch im Verlauf des Spieles oft aufgerufen werden kann, wird zunächst geprüft, ob überhaupt ein noch nicht umgeworfener Kegel auf dem Spielfeld steht. Ist dies nicht der Fall, wird durch Aufruf von „Neue Runde“ eine neue Runde gestartet.

2.2.1 Punkt x und Winkel v

In Randys Zug wird zunächst gemäß der Aufgabenstellung ein Winkel v erzeugt

$$0 \leq v \leq 360$$

und ein Punkt x . Für letzteren gehe ich wie in **1.1.2** vor;

Erzeugen eines weiteren Winkels α

$$0 \leq \alpha \leq 360$$

und eines zufälligen Abstandes r zum Kreismittelpunkt.

$$0 < r \leq \text{Kreisradius}$$

Koordinaten des Punktes $x = (x_x | x_y)$:

$$x_x = \cos(\alpha) \cdot r + \text{Kreisradius}$$

$$x_y = \sin(\alpha) \cdot r + \text{Kreisradius}$$

Ich errechne nun die Gleichung f einer Geraden, die durch den Punkt $x = (x_x | x_y)$ mit der Steigung v° geht.

$$y = f(\delta) = m \cdot \delta + a$$

Die Steigung m errechne ich mit der Tangens-Funktion aus v .

$$m = \tan(v)$$

Durch Umformung der normalen Geradengleichung kann man nun a errechnen und hat damit alle Werte, die Geradengleichung vollständig aufstellen zu können;

$$a = x_y - m \cdot x_x$$

2.2.2 Ausführen von Randys Wurf

Wenn Randy wirft, wird der Abstand der Geraden der Funktion, die seiner Wurfbahn entspricht, zu jedem Kegel errechnet, der sich noch auf dem Spielfeld befindet (also noch nicht umgeworfen wurde). Zur Berechnung des Abstandes verwende ich die Lotfußpunkt-Formel. Da diese eine der schwierigeren Techniken der Abstandberechnung ist und ich noch nicht einmal Vektorrechnung in der Schule hatte, haben Sie bitte Verständnis dafür, dass ich die folgende Rechnung nicht genauer erläutern kann.

Rechnen wir nun mit einem beliebigen Kegel $K = (K_x|K_y)$ und den errechneten Variablen m und a der Geradengleichung.

Die Koordinaten des Fußpunktes $F = (F_x|F_y)$ ergeben sich wie folgt:

$$F_x = \frac{m \cdot K_y + K_x - a \cdot m}{m^2 + 1}$$
$$F_y = m \cdot F_x + a$$

Mit dem Satz des Pythagoras errechne ich dann den Abstand des Kegels $K = (K_x|K_y)$ zum Fußpunkt $F = (F_x|F_y)$, der dem Abstand zwischen K und der Geraden f entspricht.

$$\text{Abstand} = \sqrt{(K_x - F_x)^2 + (K_y - F_y)^2}$$

Wenn dieser Abstand nun kleiner als 1 ist, wird die Eigenschaft `umgeworfen` des Kegels auf `true` gesetzt und Randys Punktestand wird um einen erhöht.

Diese Abstandsberechnung wird mit jedem Punkt durchgeführt, der noch nicht umgeworfen worden ist.

Nach seinem Zug wird `status` auf 1 gesetzt. Dies bewirkt, dass es dem Benutzer nun erlaubt ist, zu werfen.

2.3 Annas Wurf

Doch wie genau kann der Benutzer (bzw. Anna) einen Wurf ausführen? Die Schwierigkeit bestand hier, dem Benutzer zu ermöglichen, dem Programm auf möglichst einfache Weise mitzuteilen, wie er sich den Wurf denkt.

2.3.1 Eingabe

Hierzu werfen wir einen Blick auf das entworfene GUI (siehe Abb.11). Die Kegel sowie der Spielfeld-Kreis werden - wie in Unteraufgabe 1 - auf einer Canvas-Zeichenfläche angezeigt. Bei mir wirft der Benutzer folgendermaßen: er klickt an zwei unterschiedlichen Punkten auf die Canvas-Zeichenfläche. Das Programm merkt sich die Mauskoordinaten, verbindet sie und führt den Wurf aus. Dabei kann man auch außerhalb des Kreises klicken.

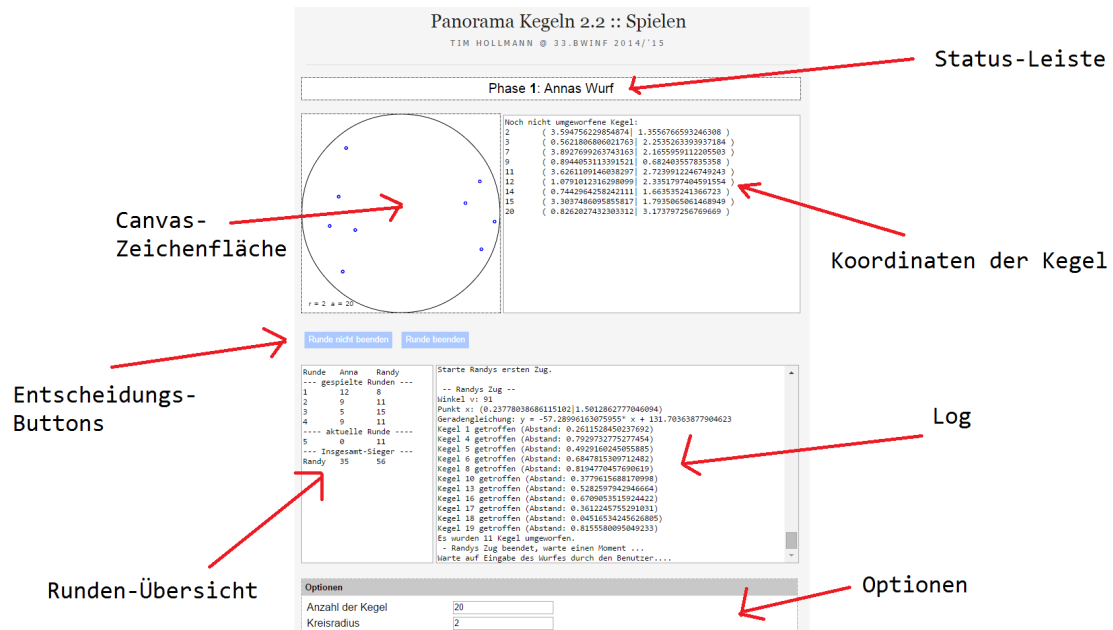


Abbildung 11: Das GUI im Browser

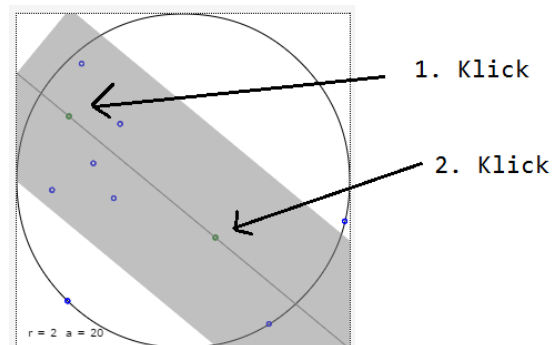


Abbildung 12: Der Benutzer klickt zwei mal auf die Zeichenfläche, das Programm verbindet diese Punkte und führt den Wurf aus

2.3.2 Errechnen der Wurf-Geraden

Man hat nun zwei Punkte $P_1 = (P_1^x | P_1^y)$ und $P_2 = (P_2^x | P_2^y)$. Man errechnet die Steigung m der Geradengleichung $y = m \cdot x + a$ durch den Differenzenquotienten:

$$\frac{\Delta y}{\Delta x} = \frac{P_1^y - P_2^y}{P_1^x - P_2^x} = m$$

Die y -Achsen-Verschiebung a kann man nun durch die umgestellte Geradengleichung mit Hilfe eines der beiden Punkte errechnen:

$$a = P_1^y - (m \cdot P_1^x)$$

Jetzt hat man die selbe Situation wie bei Randys Zug; die Gleichung der Wurf-Gerade ist bekannt und der Abstand zu jedem Kegel auf dem Spielfeld wird nun mit der Lotfußpunkt-Formel errechnet und überprüft (vgl. **2.2.2**). Da die nun folgende Vorgehensweise identisch mit der zu Randys Wurf ist, habe ich sie auch im Programm in eine eigene Funktion ausgelagert.

Nach Ausführung dieses „Wurfes“ wird `status` auf 2 gesetzt. Damit wird die erneute Eingabe von zwei Wurf-Punkten durch den Benutzer verhindert.

2.3.3 Entscheidung

Gleichzeitig wird es dem Benutzer aber dadurch ermöglicht, eine Entscheidung zu treffen:

- Das Spiel wird fortgesetzt; die Funktion „Randys Zug“ wird aufgerufen, die dann danach wiederum Annas Zug ermöglicht (\rightarrow Spiel-Kreislauf; Anstelle einer Kontrollstruktur), oder
- Die aktuelle Runde zu beenden; vor dem Aufruf von Randys Zug wird der aktuelle Rundenpunktestand zurückgesetzt (nachdem er gespeichert wurde) und neue Kegel generiert.

Dies geschieht direkt durch den Aufruf einer der Funktionen: „Randys Zug“ oder „Neue Runde“.

2.4 GUI-Funktion

Für die Aktualisierung des gesamten GUIs ist eine einzelne Funktion zuständig, die mehrere Aufgaben erledigt:

- Darstellen des Spielfeld-Kreises und der Kegel in benutzerdefinierten Farben auf der Canvas-Zeichenfläche
- Auflistung aller nicht umgeworfener Kegel und ihrer Koordinaten.

- Zeichnen der Punktestand-Übersicht: Verlauf der Runden und ihre Punktestände, Punktestand der aktuellen Runde und Insgesamt-Sieger aller Runden nach aktuellem Punktestand.
- Aktivieren bzw. Deaktivieren von Interaktionsmöglichkeiten je nach Spielphase wie z.B. die Entscheidungs-Buttons oder den event-Listener des canvas-Elementes.

Diese Funktion greift dafür ihrerseits auf die Informationen in den Globalen Variablen zu (Punktestand, Status, Kegel). Sie wird in beinahe jeder an der Simulation des Spielablaufes beteiligten Funktion aufgerufen, sodass ein Timer o.ä. für aktuelles GUI nicht nötig ist.

2.5 Umsetzung

Die Umsetzung erfolgte in JavaScript.

2.5.1 Globale Variablen

Die globalen Variablen `status`, `arrayKegel`, `punkteAktuelleRunde` und `punkteRunden` werden zu Anfang definiert und enthalten den Spiel-Status, die Kegel-Positionen, den aktuellen Runden-Punktestand und den Verlauf der Punktestände in den bereits gespielten Runden.


```
11 // Status-Variable
12 var status = 0;
13
14 // Kegel-Array
15 var arrayKegel = null;
16
17 // Punktestände: aktuelle Runde und Rundenverlauf
18 var punkteAktuelleRunde = null;
19 var punkteRunden = null;
```

2.5.2 Neues Spiel

Um ein neues Spiel zu starten, genügt es, die globale Variable `punkteRunden` zurückzusetzen und eine neue Runde mit `newRound()` zu starten, denn `status`, `punkteAktuelleRunde` und `arrayKegel` werden von dieser eh zurückgesetzt bzw. überschrieben (dazu siehe nächsten Abschnitt).

```
25 function newGame(){
26     log("\n\n\n ---- Neues Spiel gestartet ----");
27
28     /* Alte Rundenwerte löschen */
29     punkteRunden = new Array();
30
31     /* Erste Runde starten */
```

```
32     log("\n\n --- Initialisiere erste Runde ---");
33     newRound();
34 }
```

Diese Funktion wird automatisch bei Seitenaufruf ausgeführt und kann theoretisch auch innerhalb eines Spieles aufgerufen werden, um ein neues Spiel zu beginnen. Hierbei besteht jedoch die Gefahr, dass sich die Timeouts (zu denen ich später komme) überlappen und so z.B. Randy zwei mal nacheinander ziehen würde. Deshalb startet man ein neues Spiel mit Aktualisieren des Browsers per .

Ich verwende hier die Funktion `log()`, die eine aktuelle Statusmeldung der Funktion im Log-Fenster (siehe Abb. 11) ausgeben kann und so den Spielablauf für Anwender und Entwickler leichter nachvollziehbar macht.

2.5.3 Neue Runde

Um eine neue Runde zu starten, ist lediglich das Löschen des aktuellen Rundenpunktestandes und die Neu-Generierung der Kegelpositionen nötig. Diese Funktion wird während des Spieles vielfach aufgerufen (z.B. wenn sich der Benutzer für eine neue Runde entscheidet oder sich keine Kegel mehr auf dem Spielfeld befinden). Falls ein aktueller Runden-Punktestand vorhanden ist (`punkteAktuelleRunde != NULL`), wird der alte Punktestand zuvor in `punkteRunden` gesichert.

```
37 function newRound(){
38     log("\n -- Neue Runde gestartet --");
39
40     /* Alte Rundenwerte? */
41     if (punkteAktuelleRunde != null){
42         punkteRunden.push(punkteAktuelleRunde);
43         log("Alter Punktestand gefunden und gespeichert.");
44     }
45
46     /* Alten Punktestand löschen */
47     punkteAktuelleRunde = new Array();
48     punkteAktuelleRunde["anna"] = 0;
49     punkteAktuelleRunde["randy"] = 0;
50
51     /* Neue Kegel erzeugen */
52
53     var kreisRadius = ←
54         parseInt(document.getElementById("inputKreisRadius").value);
55     var anzahlKegel = ←
56         parseInt(document.getElementById("inputAnzahlKegel").value);
57
58     arrayKegel = createRandomPoints(kreisRadius, anzahlKegel);
59     log("Neue Kegelpositionen generiert.");
60
61     status = 0;
62     draw();
63 }
```

```

61     log("Starte Randys ersten Zug.");
62     setTimeout(randyZug, ←
63         parseInt(document.getElementById("inputWartezeit").value));
64 }

```

- In Zeile 53 und 54 lädt das Skript die Einstellungen `kreisRadius` und `anzahlKegel` per `document.getElementById()` direkt aus dem Eingabefeld des HTML-Formulars anhand ihrer ID; zu den IDs komme ich in Abschnitt **2.5.8**.
- Ich verwende hier in Zeile 56 die aus **1.2.2** bekannte Funktion `createRandomPoints`, deren Name etwas irreführend ist; sie generiert nicht einfach zufällige Kegelpunkte, sondern berücksichtigt den Minimalen Abstand r . Sie entspricht fast vollständig der aus **1.2.2**, nur weist sie den Kegeln eine weitere Eigenschaft zu:

```

129     temp["status"] = true;

```

Diese Eigenschaft eines Kegels gibt an, ob dieser bereits umgeworfen worden ist.

- Ich rufe in Zeile 60 hier die GUI-Funktion `draw()` auf, auf die ich später zurückkomme.
- Randys Zug (`randyZug()`) wird in Zeile 63 nicht direkt aufgerufen; es wird ein Timeout gesetzt, um die Funktion zeitverzögert zu starten; diese Zeit in Millisekunden wird direkt aus dem Eingabefeld `inputWartezeit` des HTML-Formulars geladen und kann so beliebig verändert werden. Ist diese Zeit niedrig (oder gar 0), kann man die Würfe nicht mehr nachvollziehen, da sie zu schnell ausgeführt werden. In gewisser Weise ist dies eine Erweiterung im Hinblick auf Benutzerfreundlichkeit.

Aufgrund dieser Timeouts kann es zu Überschneidungen kommen, wenn diese Funktion innerhalb einer Browsersitzung mehrmals aufgerufen würde.

2.5.4 Randys Zug

```

144 function randyZug(){
145     status = 0;
146
147     draw();
148     log("\n -- Randys Zug -- ");
149
150     /* Zufälligen Winkel v generieren */
151     var v = random(0,360);
152     var vBogen = (v * (Math.PI / 180));
153
154     log("Winkel v: " + v);
155 }

```



```

156  /* Zufälligen Punkt x generieren */
157  var a = random(0,360);
158  var aBogen = (a * (Math.PI / 180));
159
160  kreisRadius = ↵
    parseInt(document.getElementById("inputKreisRadius").value);
161
162  var abstand = random(1,100)/(100/kreisRadius);
163
164  // Punkt P : (pX | pY)
165  var pX = Math.cos(aBogen) * abstand;
166  var pY = Math.sin(aBogen) * abstand;
167
168  pX += kreisRadius;
169  pY += kreisRadius;
170
171  log("Punkt x: (" + pX + "|" + pY + ")");
172
173  /* Geradengleichung errechnen */
174
175  var m = Math.tan(vBogen);
176  var a = pY - m * pX;
177
178  log("Geradengleichung: y = " + m + "* x + " + a);
179
180  /* Wurf ausführen */
181  punkteAktuelleRunde["randy"] += wurf(m, a);
182
183  status = 1;
184
185  log(" - Randys Zug beendet, warte einen Moment ...");
186
187  setTimeout(draw, ↵
    parseInt(document.getElementById("inputWartezeit").value));
188
189  log("Warte auf Eingabe des Wurfes durch den Benutzer....");
190
191  }

```

Hier wird wie in **2.1.1** vorgegangen;

- Erzeugen von Winkel v (Zeile 151)
- Erzeugen eines zufälligen Punktes x durch einen weiteren Winkel α und Abstand zur Kreismitte r (Zeile 156-169).
- Errechnen der Geradengleichung, die durch Punkt x mit Steigung v° geht (Zeile 173-176).

2.1.2 wird hier (wie bereits erwähnt) in eine eigene Funktion (`wurf()`) ausgelagert, da sich die Vorgehensweise von hier an zu 100% mit der bei Annas Zug deckt;

```

181  punkteAktuelleRunde["randy"] += wurf(m, a, false);

```

`wurf()` zeichnet eine transparente Wurf-Linie, weshalb die GUI-Funktion `draw()` erst zeitverzögert per Timeout aufgerufen wird (Z.187), da sie diese Wurf-Linie zu schnell überschreiben (oder besser: übermalen) würde, als diese mit dem Auge wahrgenommen und nachvollzogen werden könnte.

`status` wird in Zeile 183 auf 1 gesetzt; sobald `draw()` das nächste mal aufgerufen wird, ist es dem Benutzer erlaubt, per zweimaligem Klicken einen Wurf aus zu führen.

Zunächst werfen wir aber einen Blick auf die Funktion `wurf`.

2.5.5 Wurf ausführen

Die Funktion `wurf()` führt den „eigentlichen“ Wurf aus; sie nimmt die Manipulationen an den `status`-Eigenschaften der Kegel vor und gibt die Anzahl der umgeworfenen Kegel zurück. Sie übernimmt als Parameter die Variablen m und a der Geradengleichung

$$f(\delta) = m \cdot \delta + a$$

die der Wurflinie entspricht.

```

212 function wurf(m, a, isAnna){
213
214     // Wurf entlang einer Geraden y = m * x + a
215     // boolaen isAnna := wirft Anna?
216
217     var kegelUmgeworfen = 0;
218
219     var durchmesserKugel = ←
        parseInt(document.getElementById("inputDurchmesserKugel").value);
220
221     for (var i = 0; i < arrayKegel.length; i++){
222         if (arrayKegel[i]["status"]){
223
224             /* Koordinaten des aktuellen Kegels ermitteln */
225             var px = arrayKegel[i]["x"];
226             var py = arrayKegel[i]["y"];
227
228             /* 1. Ermittle den Fusspunkt (fx | fy)
229              = Schnittpunkt der gegebenen Geraden mit der Normalen,
230              die durch den Punkt (Kegel) geht */
231             var fx = ( m * py + px - a * m) / (m * m +1);
232             var fy = m * fx + a;
233
234             /* 2. Abstand zwischen Kegel und Fusspunkt
235              = wurzel ( delta x² + delta y² ) --> Satz des Pythagoras */
236
237             var dx = px - fx;
238             var dy = py - fy;
239
240             var abstand = Math.sqrt( (dx * dx) + (dy * dy));
241
242             if (abstand <= durchmesserKugel){

```

```

243         arrayKegel[i]["status"] = false;
244         log("Kegel " + (i+1) + " getroffen (Abstand: " + abstand ←
           + ")");
245         kegelUmgeworfen++;
246     }
247
248 }
249 }
250
251 /* Grafische Ausgabe */
252
253 var canvas = document.getElementById("canvas");
254 var ctx = canvas.getContext("2d");
255
256 var kreisRadius = ←
           parseInt(document.getElementById("inputKreisRadius").value);
257 var faktor = canvas.width/(2 * kreisRadius);
258
259 var wurfFarbe = (isAnna) ? ←
           document.getElementById("inputWurfFarbeAnna").value : ←
           document.getElementById("inputWurfFarbeRandy").value;
260
261 /* Dünne Linie */
262 ctx.beginPath();
263 ctx.lineWidth = 1;
264 ctx.strokeStyle= wurfFarbe;
265 ctx.globalAlpha = 1;
266 ctx.moveTo(0, faktor * a);
267 ctx.lineTo(faktor * 100, faktor * (m * 100 + a));
268 ctx.stroke();
269
270 /* Dickere, durchsichtigere Linie */
271 ctx.beginPath();
272 ctx.lineWidth = 2 * faktor * durchmesserKugel;
273 ctx.strokeStyle=wurfFarbe;
274 ctx.globalAlpha = 0.5;
275 ctx.moveTo(0, faktor * a);
276 ctx.lineTo(faktor * 100, faktor * (m * 100 + a));
277 ctx.stroke();
278
279 log("Es wurden " + kegelUmgeworfen + " Kegel umgeworfen.");
280
281 return kegelUmgeworfen;
282 }

```

Alle Kegel werden per `for`-Schleife durchgegangen (Z.221). Wenn der Kegel dann nicht umgeworfen ist (`status != false`; Z.222), wird der Abstand zur Geraden f ermittelt, indem zunächst der Fußpunkt F ermittelt (Z.231f.) und dann der Abstand von diesem zum aktuellen Kegel ermittelt (237ff.). Ist dieser dann geringer als oder gleich dem Radius der Wurf-Kugel, der zuvor aus dem HTML-Formular geladen wurde (Z.219), wird diese Kugel per `status` als umgeworfen markiert und `kegelUmgeworfen` inkarniert (Z.242-245).

Unabhängig davon, wie viele oder überhaupt ein Kegel umgeworfen wird, folgt eine grafische Ausgabe des Wurfes durch einen intransparenten dünnen Strich f folgend (Z.261-268) und einer transparenten Linie, die den Durchmesser der Wurf-Kugel andeutet (Z.271-227). Hierzu siehe Abbildung 10. Anna und Randy haben unterschiedliche Wurf-Farben, die frei wählbar sind (Z.259).

Schließlich wird die Anzahl der umgeworfenen Kegel zurückgegeben, sodass diese Anzahl der aktuellen Runden-Punktezahl von Randy bzw. Anna hinzu addiert werden kann;

```
181 punkteAktuelleRunde["randy"] += wurf(m, a, false);
```

2.5.6 Event-Handler `canvas.onClick`

Dadurch, dass `status` am Ende von Randys Zug auf 1 gesetzt wurde, wird es dem Benutzer nach zeitverzögertem Aufruf der `draw()`-Funktion ermöglicht, auf das `canvas`-Element klicken zu können.

Das Ziel dieses Handlers ist es, zwei Punkte auf der `canvas`-Zeichenfläche zu sammeln, die der Benutzer mit der Maus anklickt. Da hier die Funktion zwei mal aufgerufen wird, ist es nötig - um die Information zu erhalten und weiter zu geben - eine globale Variable zu definieren, die Punkte speichert;

```
21 // Array zum Sammeln von eingegebenen Punkten (für Annas Wurf)
22 var punkteEingabe = null;
```

Bei jedem Aufruf des event-Handlers prüft dieser nach Eintragen des aktuellen Punktes, ob nun insgesamt zwei Punkte in `punkteEingabe` vorhanden sind. Falls nicht, tut er einfach nichts. Falls doch, fährt er wie in **2.3.2** fort.

```
427 function canvasClick(event){
428
429     var canvasTemp = document.getElementById("canvas");
430     var canvasTempContext = canvasTemp.getContext("2d");
431
432     /* Maus-Position ermitteln */
433     var xPos = event.pageX - canvasTemp.offsetLeft;
434     var yPos = event.pageY - canvasTemp.offsetTop;
435
436     draw();
437
438     if (status == 1){
439
440         if (punkteEingabe == null){
441             punkteEingabe = new Array();
442         }
443
444         var temp = new Array();
445         temp["x"] = xPos;
446         temp["y"] = yPos;
```

```
447     punkteEingabe.push(temp);
448
449     log("Eingabe am Punkt (" + xPos + "|" + yPos + ") registriert und ↵
        gespeichert als Punkt " + punkteEingabe.length);
450
451     /* Eingegebener Punkt grafisch ausgeben */
452
453     var canvas = document.getElementById("canvas");
454     var ctx = canvas.getContext("2d");
455     var kegelRadius = ↵
        parseInt(document.getElementById("inputRadiusKegel").value);
456     var faktor = canvas.width/(2 * kegelRadius);
457
458     for (var i = 0; i < punkteEingabe.length; i++){
459         ctx.beginPath();
460         ctx.strokeStyle = ↵
            document.getElementById("inputEingabeFarbe").value;
461         ctx.moveTo(punkteEingabe[i]["x"] + kegelRadius, ↵
            punkteEingabe[i]["y"]);
462         ctx.arc(punkteEingabe[i]["x"], punkteEingabe[i]["y"], ↵
            kegelRadius, 0, 2*Math.PI);
463         ctx.stroke();
464     }
465
466     if (punkteEingabe.length == 1){
467         log("Erster Punkt gesammelt.");
468
469     }else if(punkteEingabe.length == 2){
470         log("Zwei Punkte gesammelt. Führe Annas Zug aus.");
471
472         /* Annas Wurf ausführen */
473
474         /* Punkte ermitteln und in Koordinatensystem umrechnen */
475         var x1 = punkteEingabe[0]["x"] / faktor;
476         var y1 = punkteEingabe[0]["y"] / faktor;
477
478         var x2 = punkteEingabe[1]["x"] / faktor;
479         var y2 = punkteEingabe[1]["y"] / faktor;
480
481         /* Die zwei Punkte löschen */
482         punkteEingabe = null;
483
484         /* Geradengleichung errechnen */
485         // y = m*x+a
486
487         //Steigung m errechnen
488         var dX = x1 - x2;
489         var dY = y1 - y2;
490
491         var m = dY/dX;
492
493         //y-Achsen-Verschiebung a errechnen
494         var a = y1-(m * x1);
495
```

```

496         log("Geradengleichung: y = " + m + " * x + " + a);
497
498         /* Wurf ausführen */
499         punkteAktuelleRunde["anna"] += wurf(m, a, true);
500
501         status = 2;
502
503         setTimeout(draw, ↵
504             parseInt(document.getElementById("inputWartezeit").value));
505         log("Warte auf Entscheidung des Benutzers...");
506     }else{
507         /* sehr sonderbar. Alles löschen. */
508         punkteEingabe = new Array();
509         log("Fehler: Zu viele Punkte. Wiederholen Sie die Eingabe.");
510     }
511
512     }else{
513         log("Klick registriert, aber Anna ist nicht dran mit werfen.");
514     }
515 }

```

Die Koordinaten der Maus werden im event-Objekt `event` als Parameter übergeben (Z.427); sie sind allerdings noch absolut, das heißt sie gehen vom oberen linken Fens-terrand aus. Zieht man die Koordinaten des canvas-Elementes davon ab (die man per `offsetLeft` und `offsetTop` ermittelt, Z.433), hat man die Mausposition relativ zum canvas-Element. Diese Koordinaten sind aber noch in Pixeln angegeben; teilt man sie durch den Darstel-lungsfaktor (Z.475-479), kann man sie im Koordinatensystem verwenden.

Die Ermittlung des Steigungsfaktors m und der y -Achsen-Verschiebung a erfolgt dann nach **2.3.2** (Z.487-494) und der Wurf dann wie erwähnt mit der Funktion `wurf()` (Z.499).

`status` wird nach Ausführung des Wurfes in Zeile 501 auf 2 gesetzt. Nachdem die GUI-Funktion `draw` in Zeile 503 per Timeout aufgerufen wurde, ist es dem Benutzer nicht mehr möglich, weitere Punkte einzugeben (er event-Listener wird einfach entfernt). Stattdessen werden zwei Schaltflächen aktiviert (Abb.13).

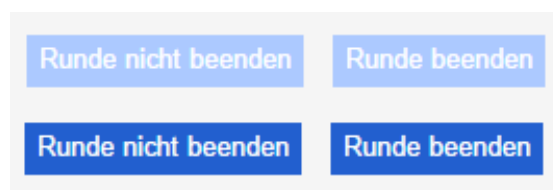


Abbildung 13: Entscheidungs-Schaltflächen: Deaktiviert (oben), aktiviert (unten, `status = 2`)

Ein Klick auf die Schaltfläche „Runde nicht beenden“ ruft `randyZug` auf und startet damit Randys Zug, der nach seiner Durchführung wieder die Eingabe der Wurf-Punkte

durch den Benutzer ermöglicht (\rightarrow Spiel-Kreis). Das Setzen von `status` auf 0 vor dem Aufruf von `randyZug` ist nicht nötig, da diese es von selbst tut (vgl. Z. 45).

Ein Klick auf die Schaltfläche „Runde beenden“ ruft die Funktion `newRound` auf und startet so eine neue Runde.

2.5.7 GUI-Funktion `draw`

Für die komplette Aktualisierung des GUIs ist eine einzige Funktion zuständig: `draw`. Sie zeichnet nicht nur den Spielfeld-Kreis mitsamt Kegeln auf das `canvas`-Element, sondern füllt auch eine Liste mit den Koordinaten der Punkte, die noch nicht umgeworfen worden sind und eine Runden-Übersicht mit Punkteverlauf und Insgesamt-Sieger. Sie ist aber nicht zuletzt auch für die Regulierung der Interaktionsmöglichkeiten je nach Spielstatus zuständig.

```
285 function draw(){
286
287     if (!(checkFeldLeer())) return false;
288
289     var canvas = document.getElementById("canvas");
290     var canvasContext = canvas.getContext("2d");
291
292     var kreisRadius = ←
293         parseInt(document.getElementById("inputKreisRadius").value);
294     var kegelRadius = ←
295         parseInt(document.getElementById("inputRadiusKegel").value);
296     var kreisFarbe = document.getElementById("inputKreisFarbe").value;
297     var kegelFarbe = document.getElementById("inputKegelFarbe").value;
298
299     /* Darstellungsfaktor errechnen */
300     var faktor = canvas.width/(2 * kreisRadius);
301
302     /* ---- Canvas-Element zeichnen ---- */
303
304     /* Zeichenfläche leeren */
305     canvasContext.clearRect(0, 0, canvas.width, canvas.height);
306
307     canvasContext.beginPath();
308
309     canvasContext.lineWidth = 1;
310     canvasContext.globalAlpha = 1;
311
312     canvasContext.strokeStyle = kreisFarbe;
313     canvasContext.arc(faktor * kreisRadius, faktor * kreisRadius, faktor * ←
314         kreisRadius, 0, 2*Math.PI);
315     canvasContext.stroke();
316
317     /* Kegel-Box ermitteln, um diese mit den Koordinaten der Punkte zu ←
318         füllen */
319     var kegelBox = document.getElementById("kegelBox");
320     kegelBox.value = "Noch nicht umgeworfene Kegel:";
```

```
318     for(var i = 0; i < arrayKegel.length; i++){
319         if (arrayKegel[i]["status"]){
320             canvasContext.beginPath();
321             canvasContext.strokeStyle = kegelFarbe;
322             canvasContext.moveTo(faktor * arrayKegel[i]["x"] + ↵
                 kegelRadius, faktor * arrayKegel[i]["y"]);
323             canvasContext.arc(faktor * arrayKegel[i]["x"], faktor * ↵
                 arrayKegel[i]["y"], kegelRadius, 0, 2*Math.PI);
324             canvasContext.stroke();
325
326             // Kegel-Box mit Koordinaten füllen
327             kegelBox.value += "\n" + (i+1) + " ( " + arrayKegel[i]["x"] + ↵
                 "| " + arrayKegel[i]["y"] + " )";
328         }
329     }
330
331     /* Unten in die Darstellungsfläche die groben Informationen schreiben: ↵
        Kreisradius und Kegelanzahl */
332
333     var textabstand = 10;
334     var fontsize = 10;
335
336     canvasContext.beginPath();
337     canvasContext.font = fontsize + "px Verdana";
338     canvasContext.fillText("r = " + kreisRadius + " a = " + ↵
        arrayKegel.length, 10, canvas.height - textabstand);
339     canvasContext.stroke();
340
341     /* ---- Runden-Übersicht zeichnen ---- */
342     var roundBox = document.getElementById("roundBox");
343     roundBox.readOnly = true;
344     roundBox.style.resize = 'none';
345     roundBox.value = "Runde Anna Randy";
346     roundBox.value += "\n-- gespielte Runden --";
347
348     var punkteRandy = 0;
349     var punkteAnna = 0;
350
351     for(var i = 0; i < punkteRunden.length; i++){
352         roundBox.value += "\n" + (i + 1) + " " + ↵
            punkteRunden[i]["anna"] + " " + punkteRunden[i]["randy"];
353         punkteRandy += punkteRunden[i]["randy"];
354         punkteAnna += punkteRunden[i]["anna"];
355     }
356
357     /* Noch keine Runden gespielt? -> Platzhalter */
358     if (punkteRunden.length == 0){
359         roundBox.value += "\n -/-";
360     }
361
362     roundBox.value += "\n-- aktuelle Runde --";
363     roundBox.value += "\n" + (punkteRunden.length + 1) + " " + ↵
        punkteAktuelleRunde["anna"] + " " + punkteAktuelleRunde["randy"];
364
```



```
365     punkteAnna += punkteAktuelleRunde["anna"];
366     punkteRandy += punkteAktuelleRunde["randy"];
367
368     roundBox.value += "\n-- Insgesamt-Sieger --";
369     roundBox.value += "\n";
370
371     if (punkteAnna > punkteRandy){
372         roundBox.value += "Anna"
373     }else if(punkteAnna < punkteRandy){
374         roundBox.value += "Randy"
375     }else{
376         roundBox.value += "Unent."
377     }
378
379     roundBox.value += " " + punkteAnna + " " + punkteRandy;
380
381     /* Sonstige Eigenschaften der Boxen */
382     document.getElementById("logBox").readOnly = true;
383     document.getElementById("logBox").style.resize = 'none';
384
385     document.getElementById("kegelBox").readOnly = true;
386     document.getElementById("kegelBox").style.resize = 'none';
387
388     /* Status-Abhängige Eigenschaften */
389
390     if (status == 0){
391         /* Weder Wurf noch Entscheidung erlaubt */
392         //Wurf deaktivieren
393         document.getElementById("canvas").removeEventListener("click", ↵
            canvasClick, false);
394         document.getElementById("canvas").style.cursor = 'no-drop';
395         //Entscheidung deaktivieren
396         document.getElementById("endRoundButton").disabled = true;
397         document.getElementById("noEndRoundButton").disabled = true;
398         //Phasen-Überschrift
399         document.getElementById("phase").innerHTML = "Phase <b>0</b>: ↵
            Randy's Wurf";
400     }else if(status == 1){
401         /* Nur Wurf erlaubt */
402         //Wurf aktivieren
403         document.getElementById("canvas").addEventListener("click", ↵
            canvasClick, false);
404         document.getElementById("canvas").style.cursor = 'crosshair';
405         //Entscheidung deaktivieren
406         document.getElementById("endRoundButton").disabled = true;
407         document.getElementById("noEndRoundButton").disabled = true;
408         //Phasen-Überschrift
409         document.getElementById("phase").innerHTML = "Phase <b>1</b>: ↵
            Annas Wurf";
410     }else{
411         /* Nur Entscheidung erlaubt */
412         // Wurf deaktivieren
413         document.getElementById("canvas").removeEventListener("click", ↵
            canvasClick, false);
```

```

414     document.getElementById("canvas").style.cursor = 'no-drop';
415     // Entscheidung aktivieren
416     document.getElementById("endRoundButton").disabled = false;
417     document.getElementById("noEndRoundButton").disabled = false;
418     //Phasen-Überschrift
419     document.getElementById("phase").innerHTML = "Phase <b>2</b>: ↔
        Annas Entscheidung";
420 }
421
422 return false;
423
424 }

```

Bei jedem Aufruf wird zunächst durch die Funktion `checkFeldLeer()` (auf die ich hier nicht genauer eingehe, da sie sehr trivial arbeitet) geprüft, ob überhaupt noch nicht umgeworfene Kegel auf dem Spielfeld stehen. (Z.287) und ggf. eine neue Runde per `newRound` gestartet.

Nach Laden von einigen Einstellungen (Z.292-295) zeichnet sie den Spielfeld-Kreis (Z.331) und die noch nicht umgeworfenen Kegel-Punkte auf die canvas-Zeichenfläche (Z.318-329). Im gleichen Rutsch füllt sie ein Textarea-Feld mit den Koordinaten der noch nicht umgeworfenen Kegel (Z.314-316 u. 327) gleich mit.

Eine Übersicht der gespielten Runden und ihrer Punktestände, sowie der aktuelle Runden-Punktestand und der Ingesamt-Sieger nach aktuellem Punktestand werden in Z. 341-379 in ein weiteres Textarea-Feld geschrieben.

Etwas besonders wichtiges für den Spielablauf kommt jetzt: Je nach Spielphase werden Interaktionsmöglichkeiten verhindert oder erlaubt, wie z.B. die Entscheidungs-Buttons deaktiviert oder der Event-Handler des `canvas`-Elementes entfernt oder aktiviert (Z.390-420). Auch der Cursor über der Canvas-Zeichenfläche passt sich dem Spielstatus an. Der Phasen-Indikator oberhalb des Canvas-Elementes wird ebenfalls je nach Status angepasst.

2.5.8 Das HTML-Gerüst

Das Skript stützt sich - wie eigentlich jedes JavaScript - auf eine existierende HTML-Seite und greift auf deren Elemente zu. Konkret in meinem Skript natürlich auf das `canvas`-Element

```
<canvas id="canvas" width="300" height="300">...</canvas>
```

die Ausgabeboxen für Log, Kegel-Koordinaten und Runden-Übersicht

```
<textarea id="logBox" style="width: 550px; height: 300px;"></textarea>
```

```
<textarea id="kegelBox" style="width: 450px; height: 300px;"></textarea>
```

```
<textarea id="roundBox" style="width: 200px; height: 300px;"></textarea>
```

sowie einige Eingabefelder für diverse benutzerdefinierte Einstellungen.

```
Anzahl der Kegel
<input id="inputAnzahlKegel" value="20" />
Kreisradius
<input id="inputKreisRadius" value="2" />
Durchmesser der Wurfkugel
<input id="inputDurchmesserKugel" value="1" />
Wartezeit
<input id="inputWartezeit" value="2000" />ms
Radius der Kegel-Kreise
<input id="inputRadiusKegel" value="2" />px
Farbe des Kreises
<input id="inputKreisFarbe" value="black" />
Farbe der Kegel
<input id="inputKegelFarbe" value="blue" />
Farbe der Eingabepunkte
<input id="inputEingabeFarbe" value="green" />
Farbe der Wurflinie von Anna
<input id="inputWurfFarbeAnna" value="red" />
Farbe der Wurflinie von Randy
<input id="inputWurfFarbeRandy" value="purple" />
```

Hinzu kommen die beiden Buttons für die Entscheidung(Runden-Ende oder nicht)

```
<input class="button" type="button" id="noEndRoundButton" value="Runde ↔
nicht beenden" onClick="javascript:randyZug();" />
<input class="button" type="button" id="endRoundButton" value="Runde ↔
beenden" onClick="javascript:newRound();" />
```

Bei Seitenaufruf wird sofort ein neues Spiel gestartet:

```
<body onLoad="javascript:newGame();" >
```

2.6 Ausführung

Das Script starten sie deshalb durch Öffnen des Dokumentes „2.html“ mit einem HTML5-fähigen Browser Ihrer Wahl.

3 Aufgabe 3

3.1 Theorie - Strategie

Für Annas Strategie können Wurf und Entscheidung herangezogen werden. Gemäß dem Ziel, insgesamt mehr Kegel zu erreichen als Randy, ist es naheliegend, Annas Würfe

zunächst einmal so effizient wie möglich zu gestalten; die von Anna geworfenen Würfe müssen stets die meist mögliche Anzahl an Kegeln umwerfen. Auf der Suche nach einem passenden Algorithmus hierfür bin ich auf den RANSAC-Algorithmus gestoßen, der allerdings leider nicht in oder für JavaScript Implementiert wurde, sodass mir folgende Möglichkeiten blieben:

1. Verbinden von zwei beliebigen Kegeln und denjenigen Wurf ausführen, der am meisten Kegel umwirft
2. 360° von jedem Kegel ausprobieren und -

Möglichkeit 2 erschien mir allerdings als zu ineffizient ($360 * 20$ Tests pro Zug). Ich verbinde also bei jedem Zug der Strategie-KI jeden Kegel einmal mit jedem anderen (unter der Bedingung, dass beide noch auf dem Spielfeld stehen) und führe denjenigen Wurf aus, der die meisten Kegel umstößt. Es hat sich im Folgenden herausgestellt, dass es nur wenige Situationen gibt, bei denen diese Strategie nicht die meist mögliche Anzahl an Kegeln umstößt, obgleich sie an die Verbindung von zwei Punkten gebunden ist. Im Gegensatz zu Randy trifft sie so auch pro Wurf mindestens 2 Kegel.

Die Länge eines Spieles wird von der Angabe nicht genau definiert; dort ist lediglich die Rede davon, „über viele Spielrunden hinweg mehr Punkte als Randy zu sammeln“. Wenn der Spielablauf und dessen Darstellung zum Zeitpunkt der Interaktion wie in **2.2** erfolgt, ist es auch kein Problem, das Spiel theoretisch unendlich lange fortlaufen zu lassen; aus Gründen, auf die ich in **3.2** genauer eingehe, ist es notwendig, die Simulation des Spielablaufes zum Zeitpunkt deren Anzeige für den Benutzer bereits abgeschlossen zu haben; die Simulation erfolgt vor der Anzeige. Deshalb ist auch eine maximale Anzahl an Runden notwendig; das Spiel besteht dann aus einer gegebenen Anzahl an Runden.

Durch die Entscheidung am Ende ihres Zuges für eine neue Runde kann Anna das Spiel seinem Ende näher bringen; oder sie kann das Spiel fortführen und hat dann eine Chance auf die evtl. noch auf dem Spielfeld verbliebenen Kegel. Allerdings kann Randy einige dieser Kegel zufälliger Weise auch selbst umwerfen; Anna muss also dazwischen abwägen, ob die noch auf dem Spielfeld befindlichen Kegel von Randy getroffen werden, oder, diese selbst umzuwerfen, um ihren Punkteabstand zu Randy auszubauen.

Meine Strategie-KI geht das Risiko einer Runden-Fortsetzung nur dann ein, wenn sie so weit im Punkte-Vorsprung ist, dass Randy, selbst wenn er die Hälfte der auf dem Spielfeld verbliebenen Kegel zufälliger Weise umwerfen würde, immer noch weniger oder gleich viele Punkte wie Anna hätte. Bei einem Rückstand hat Anna die Punkte zwar noch nötiger, jedoch überwiegt hier die Gefahr, dass Randy noch weiter in Vorsprung kommt.

3.2 Umsetzung

In dieser Unteraufgabe ist keinerlei Interaktion mit dem Benutzer mehr nötig; der Spielablauf kann deshalb innerhalb einer Kontrollstruktur stattfinden, womit `status` nicht mehr benötigt wird. Es ist naheliegend, Annas Strategie ähnlich wie Randys Zug `randyZug()` in eine eigene Funktion zu legen: `annaZug()`. Es könnte folgendermaßen aussehen:

```
function game(){
  for(i = 1; i <= anzahlRunden; i++){
    round();
  }
}

function round(){
  createRadomPoints(); // Kegelpositionen generieren
  do{
    randyZug(); // Randys Zug
  }while(annaZug()); // Annas Zug (Rückgabewert ^= Entscheidung)
}
```

Da Randys Zug in einer Runde immer zuerst ausgeführt wird und dann Annas Zug und ihre Entscheidung zur Weiterführung der Runde ausschlaggebend sind, bietet sich hier eine fußgesteuerte Schleife an. Bei der Umsetzung in JavaScript stellte sich dann leider heraus, dass das `canvas`-Element innerhalb einer Schleife nicht neu gezeichnet wird. Ich habe es auch mit Timeouts versucht, aber bei denen besteht die Gefahr der gegenseitigen Überschneidung. Deshalb war es notwendig, die Simulation des Spielablaufes zeitlich von dessen Darstellung zu trennen und den Quelltext in eine andere Programmiersprache zu portieren, die in der Lage ist, möglichst einfach und schnell die Darstellungen in Bild Form während des Programmablaufes zu erzeugen. Ich habe mich am Turniersystem orientiert und den Quelltext in PHP portiert. Das PHP-Script soll die Spielfeld-Zustände und Würfe in Bild-Form abspeichern und später zum Ansehen bereitstellen; hierfür muss aber das Spiel aber zuerst in einzelne Abschnitte eingeteilt werden, die dann gespeichert werden (im Turniersystem wären dies die „Runden“). Z.B. Abb.14

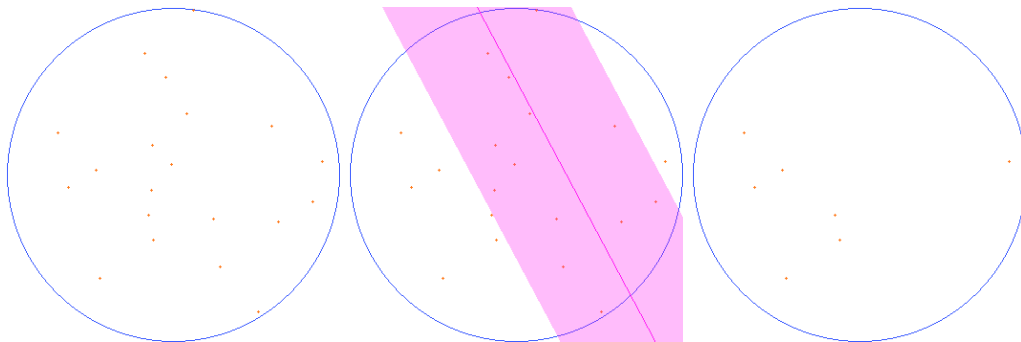


Abbildung 14: Zustände: Zustand 1, Zustand 2 (Wurf), Zustand 3

Die Bilder alleine wären als Darstellung aber ein bisschen mager; zu einem Spielzustand gehören noch ein Log und der aktuelle Punktestand; ich nenne sie zusammen „Frames“. Ein „Frame“ besteht also aus mehreren Dateien:

- dem Bild des Spielfeldes (*.png)
- dem Punktestand zum Zeitpunkt des Frames (*_points.txt)













 1.png	06.02.2015 21:20	PNG-Bild
 1_log.txt	06.02.2015 21:20	Textdokument
 1_points.txt	06.02.2015 21:20	Textdokument
 2.png	06.02.2015 21:20	PNG-Bild
 2_log.txt	06.02.2015 21:20	Textdokument
 2_points.txt	06.02.2015 21:20	Textdokument
 3.png	06.02.2015 21:20	PNG-Bild
 3_log.txt	06.02.2015 21:20	Textdokument
 3_points.txt	06.02.2015 21:20	Textdokument
 4.png	06.02.2015 21:20	PNG-Bild
 4_log.txt	06.02.2015 21:20	Textdokument
 4_points.txt	06.02.2015 21:20	Textdokument

Abbildung 15: Frame-Dateien

- dem Log: Alles was in der Log-Box seit Beenden des letzten Frames ge-logt wurde (*_log.txt)

Diese Dateien werden dann in einem eigenen Ordner abgelegt (sein Name ist der aktuelle UNIX-Zeitstempel → keine doppelten Ordner-Namen). Ein JavaScript-Player greift dann per *IMG*-Tag bzw. JavaScript-AJAX-Request auf diese Dateien zu. (Hierfür müssen diese auf einem Server liegen, was PHP aber ohnehin schon erfordert.). Den Player habe ich an den Player des Turniersystems angelehnt (**3.2.1**).

3.2.1 Das GUI

Eigentlich müsste ich eher schreiben: Der Player. Denn das ist es, was der Benutzer (neben der Eingabemaske) ausschließlich zu Gesicht bekommt; das Backend in Form von PHP tritt logischerweise nicht in Erscheinung sondern kümmert sich lediglich um die Erstellung der Frames und den Spielablauf. Der Browser ist auch hier wieder die (grafische) Schnittstelle zum Benutzer; dieser ruft mit diesem zunächst die Datei `index.html` auf, in der sich die Eingabemaske befindet (Abb. 16 links). Nach Eingabe der Einstellungen sendet er diese mit Klick auf "Spiel starten!" per `GET` aus dem HTML-Formular an das PHP-Skript `game.php`. Dieses simuliert daraufhin den Spielablauf und generiert dabei die Frames. Anschließend gibt es den JavaScript-Player (siehe Abb. 16 rechts) zurück, das durch Zugriff auf die erstellten Frame-Dateien dem Benutzer die Betrachtung des soeben ausgetragenen Duells ermöglicht.

3.2.2 `index.html` - Eingabemaske für Spieleinstellungen

Wir beginnen chronologisch mit der Eingabemaske. Diese erreicht der Benutzer durch Öffnen der Datei `index.html` mit einem Browser. Da der Sinn dieser Webseite in der

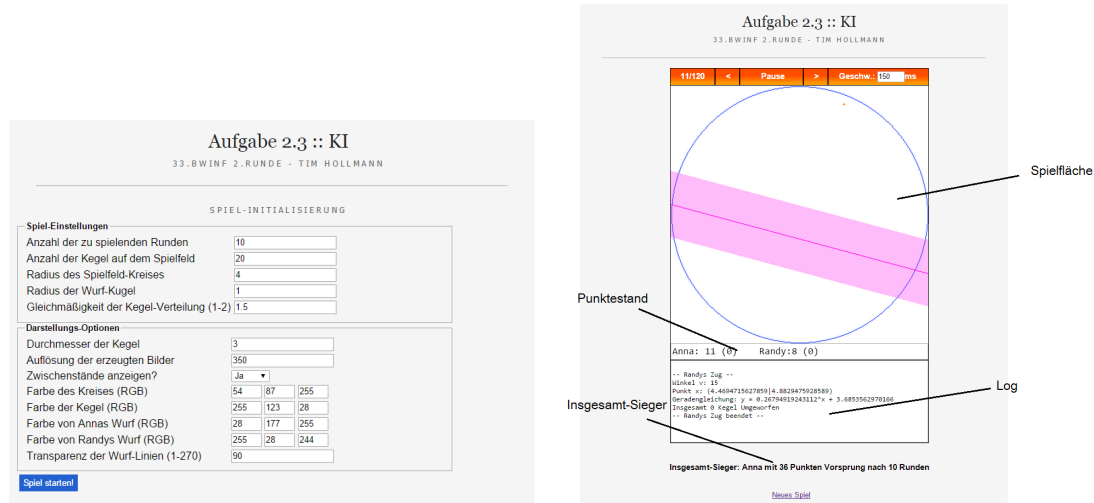


Abbildung 16: **Links:** Die Eingabemaske für die Spiel-Konfiguration; neben Einstellungen des Spiel betreffend können hier auch Einstellungen bezüglich der grafischen Darstellung in den Frames getroffen werden. **Rechts:** Der Frame-Player: Spielfläche, Punktestand, Insgesamt-Sieger und Log sowie Kontrollelemente

Informationsübermittlung zur simulierenden Skript-Datei `game.php` gedacht ist, besteht sie auch hauptsächlich aus einem HTML-Formular und Eingabeboxen

```
<html>
  <head>
    [...]
    <title>Aufgabe 2.3 :: KI - Spiel-Initialisierung | Tim Hollmann @ <-
      33.BwInf 2014/'15</title>
  </head>
  <body>
    [...]
    <Form action="game.php" method="get">
      <fieldset [...]>
        <legend><strong>Spiel-Einstellungen</strong></legend>
        <input type="number" name="anzahlRunden" value="10" />
        <input type="number" name="anzahlKegel" value="20" />
        <input type="number" name="kreisRadius" value="4" />
        <input type="number" name="durchmesserKugel" value="1" />
        <input name="verteilungsfaktor" value="1.5">

        [...]

      </fieldset>

      <fieldset [...]>
        <legend><strong>Darstellungs-Optionen</strong></legend>
```

```
<input type="number" name="kegelDurchmesser" value="3" />
<input type="number" name="bildAufloesung" value="350" />

[...]
```

```
</fieldset>

<input class="button" type="submit" value="Spiel starten!" />
</Form>
[...]
```

```
</body>

</html>
```

Veränderbare Spiel- und Darstellungsoptionen

- Anzahl der Runden, die gespielt werden sollen
- Die Anzahl der Kegel auf dem Spielfeld (20 Standard)
- Radius des Spielfeld-Kreises (Standard 4, nicht 2)
- Radius der Wurf-Kugel; maximaler Abstand zur Wurf-Linie (Standard: 1)
- Gleichmäßigkeit: In diesem Skript ist es möglich, die Gleichmäßigkeit der Kegel-Verteilung noch weiter zu erhöhen; in **1** und **2** habe ich sie auf $1r$ festgesetzt, hier kann sie nun auf das Maximum $2r$ erhöht werden.
- Größe der anzuzeigenden Kegel; Kreise oder Pixel?
- Die Auflösung der vom Skript erzeugten Frame-Spielzustand-Bilder; evtl. muss 6 dementsprechend abgepasst werden
- Sollen Zwischen zwei Würfen Zwischenstände angezeigt werden (evtl. übersichtlicher). Hierzu siehe z.B. Abb. 14; das rechte wäre Bild wäre ein Zwischenstand vor einem darauf folgenden Wurf.
- Die Farben des Kreises
- der Kegel
- des Wurfes von Anna
- und des Wurfes von Randy
- Transparenz der dickeren Linie, die der Wurf-Linie folgend die Fläche kennzeichnet, auf der die Kegel umgeworfen wurden; die darunterliegenden (und folglich umgeworfenen) Punkte sind dementsprechend mehr oder weniger sichtbar.

Nach Klick auf „Spiel starten“ werden diese Einstellungen per URL-Parameter (\rightarrow GET) an `game.php` übermittelt.

Spiel-Einstellungen	
Anzahl der zu spielenden Runden	<input type="text" value="10"/>
Anzahl der Kegel auf dem Spielfeld	<input type="text" value="20"/>
Radius des Spielfeld-Kreises	<input type="text" value="4"/>
Radius der Wurf-Kugel	<input type="text" value="1"/>
Gleichmäßigkeit der Kegel-Verteilung (1-2)	<input type="text" value="1.5"/>

Darstellungs-Optionen	
Durchmesser der Kegel	<input type="text" value="3"/>
Auflösung der erzeugten Bilder	<input type="text" value="350"/>
Zwischenstände anzeigen?	<input type="button" value="Ja"/>
Farbe des Kreises (RGB)	<input type="text" value="54"/> <input type="text" value="87"/> <input type="text" value="255"/>
Farbe der Kegel (RGB)	<input type="text" value="255"/> <input type="text" value="123"/> <input type="text" value="28"/>
Farbe von Annas Wurf (RGB)	<input type="text" value="28"/> <input type="text" value="177"/> <input type="text" value="255"/>
Farbe von Randys Wurf (RGB)	<input type="text" value="255"/> <input type="text" value="28"/> <input type="text" value="244"/>
Transparenz der Wurf-Linien (1-270)	<input type="text" value="90"/>

Abbildung 17: Eingabemaske index.html

localhost/aufgabe2_3/game.php?anzahlRunden=10&anzahlKegel=20&kreisRadius=4&durchmesserKugel=1&verteilungsfaktor=1.5&kegelDurchr

Abbildung 18: URL-Parameter beim Aufruf von game.php nach index.html

3.2.3 game.php - Spielsimulation

Laden der Einstellungen

Beim Skript angekommen werden die Einstellungen zunächst in Konstanten definiert

```

12 define("anzahlRunden", $_REQUEST["anzahlRunden"]); //Anzahl der Runden
13 define("anzahlKegel", $_REQUEST["anzahlKegel"]); //Anzahl der Kegel
14 define("kreisRadius", $_REQUEST["kreisRadius"]); //Kreis-Radius
15 define("durchmesserKugel", $_REQUEST["durchmesserKugel"]); //Durchmesser ↵
    der Wurf-Kugel
16 define("verteilungsfaktor", $_REQUEST["verteilungsfaktor"]); ↵
    //Gleichmäßigkeits-Faktor
17
18 define("kegelDurchmesser", $_REQUEST["kegelDurchmesser"]); //Durchmesser ↵
    der Kegel auf dem erzeugten Bild
19 define("bildAufloesung", $_REQUEST["bildAufloesung"]); //Auflösung der ↵
    erzeugten Bilder (px)
20
21 define("zwischenstand", (($_REQUEST["zwischenstand"] == 1) ? true: ↵
    false)); //Zwischenstände anzeigen?
22

```

```

23 define("transparency", $_REQUEST["transparency"]); //Transparenz der ↔
    Wurf-Linien
24
25 $kreisFarbe = Array();
26 $kreisFarbe["r"] = $_REQUEST["kreisFarbeR"];
27 $kreisFarbe["g"] = $_REQUEST["kreisFarbeG"];
28 $kreisFarbe["b"] = $_REQUEST["kreisFarbeB"];
29
30 $kegelFarbe = Array();
31 $kegelFarbe["r"] = $_REQUEST["kegelFarbeR"];
32 $kegelFarbe["g"] = $_REQUEST["kegelFarbeG"];
33 $kegelFarbe["b"] = $_REQUEST["kegelFarbeB"];
34
35 $wurfFarbeAnna = Array();
36 $wurfFarbeAnna["r"] = $_REQUEST["wurfFarbeAnnaR"];
37 $wurfFarbeAnna["g"] = $_REQUEST["wurfFarbeAnnaG"];
38 $wurfFarbeAnna["b"] = $_REQUEST["wurfFarbeAnnaB"];
39
40 $wurfFarbeRandy = Array();
41 $wurfFarbeRandy["r"] = $_REQUEST["wurfFarbeRandyR"];
42 $wurfFarbeRandy["g"] = $_REQUEST["wurfFarbeRandyG"];
43 $wurfFarbeRandy["b"] = $_REQUEST["wurfFarbeRandyB"];

```

Die Farben \$kreisFarbe, \$kegelFarbe, \$wurfFarbeAnna und \$wurfFarbeRandy habe ich hier als Variablen und nicht als Konstanten definiert, da die per define definierten Konstanten in PHP nur Numerische oder Boolesche Werte besitzen dürfen und die Farben ein Array sind (bzw. ein pseudo-RGB-Objekt). Falls sie sich nicht mit PHP auskennen: die per GET oder POST übermittelten Werte können im Skript in den entsprechenden assoziativen Arrays \$_POST, \$_GET oder übergeordnet \$_REQUEST angesprochen werden; der Index entspricht dann dem Wert, der im HTML-Formular im Attribut name angegeben wurde. (Die Verwendung von \$_REQUEST hat den Vorteil, dass man jetzt z.B. auch Informationen über POST übergeben könnte bzw. den Übertragungstyp auch problemlos auf POST ändern kann, falls einem die GET-Parameter in der URL missfallen.)

Frames und -Verwaltung

Herzstück der Verwaltung der Frames ist die globale Variable \$currentFrame, in der die Nummer des aktuellen Frames gespeichert ist; der Name der erzeugten Dateien, die zu diesem Frame gehören, hängt von dieser Nummer ab (siehe Abb.15).

Die Funktion nextFrame() wechselt zum nächsten Frame, indem sie \$currentFrame inkrementiert, die dann von der die Grafik erzeugenden Funktion sowie der Log- und Punktestands- Funktion verwendet wird.

Die Funktion draw() erzeugt die Grafiken und die Funktion savePoints() speichert den aktuellen Punktestand. Per logText(\$text) kann eine Log-Nachricht gespeichert werden (wie in Unteraufgabe 2, nur dass hier der Log an den aktuellen Frame gebunden ist; es wird eine eigene Log-Textdatei pro Frame erstellt, Abb. 15).

Weitere Funktionen

Einen Großteil der an der Simulation des Spielablaufes beteiligten Funktionen kann ich (neben Portierungsbedingten Änderungen) unverändert aus Unteraufgabe 2 übernehmen, wie z.B. die Funktionen `createRandomPoints()`, `checkFeldLeer()`, `randyZug()` und `wurf`.

Funktionsname	Aufgabe
<code>nextFrame</code>	Wechselt zum nächsten Frame
<code>savePoints</code>	Speichert den aktuellen Punktestand in einer Frame-Zugehörigen Datei
<code>draw</code>	Erzeugt die Grafik des Spielfeldzustandes
<code>createRandomPoints</code>	Zufällige und gleichmäßige Erzeugung von Kegelpositionen
<code>wurf</code>	„Wirft“ eine Kugel anhand einer per Parameter übergebenen Linie
<code>randyZug</code>	Randys Zug
<code>annaZug</code>	Annas Zug

Globale Variablen

Alle wichtigen Spieleigenschaften sind bereits am Anfang des Skriptes als Konstanten definiert werden. Globale Variablen sind deshalb nur noch dann nötig, wenn es sich dabei um Datentypen handelt, die von der `define`-Funktion nicht unterstützt werden, wie z.B. Arrays. So z.B. das Punktestand-Array `$punkte` oder der Kegel-Array `$kegelArray`. (Abgesehen davon, dass es zudem nicht intelligent wäre, den Punktestand als Konstante zu definieren).

Der Spielablauf

Pro angefordertem Spiel wird ein Unterordner erstellt, in dem die Frame-Dateien abgelegt werden. Der Name des Ordners entspricht (damit keine Überschneidungen entstehen) dem UNIX-Zeitstempel zum Zeitpunkt der Ausführung des Skriptes

```

384 //Ordner anlegen
385 $timestamp = time();
386 if (!(mkdir($timestamp))) Die("Fehler - Verzeichnis zum Speichern der ↵
    Frames konnte nicht erstellt werden.");
387 define("path", $timestamp."/");

```

Ist der Ordner erfolgreich erstellt, wird der Spielablauf gestartet;

```

397 //Spiel starten
398 for($a = 0; $a < anzahlRunden; $a++){
399
400     nextFrame(); //Nächsten Frame
401     logFrame("Neue Runde gestartet (Nr. " . ($a+1) . ".)");
402
403     //Punktestand-Variable; neue Runde
404     $temp = Array();
405     $temp["anna"] = 0;
406     $temp["randy"] = 0;

```

```

407     $punkte[] = $temp;
408
409     //Neues Spielfeld generieren
410     $kegelArray = createRandomPoints();
411     logFrame("Kegel generiert.");
412
413     //Die erzeugten Kegel anzeigen
414     draw();
415     savePoints();
416     logFrame("Starte den Spielablauf mit Initialisierung von Randys erstem ↵
        Zug.");
417
418     do{
419         if (zwischenstand){ nextFrame(); logFrame(" "); draw(); ↵
            savePoints(); }
420         randyZug(); //Randys Zug
421         if (checkFeldLeer()) break;
422         if (zwischenstand){ nextFrame(); logFrame(" "); draw(); ↵
            savePoints(); }
423     }while(annaZug() && !checkFeldLeer()); //Annas Zug
424
425 }

```

Per `for`-Schleife werden `anzahlRunden` viele Runde gestartet (Z.398). Bei jeder neuen Runde wird ein neuer Frame gestartet (Z.400) und ein neuer Punktestand gesetzt (Z.404-407). Beim Punktestand-Array `$punkte` ist der letzte Index die aktuelle Runde (in der letzten Unteraufgabe hatte ich die aktuellen und abgeschlossenen Werte noch voneinander getrennt). Durch Hinzufügen eines leeren Arrays an dieses Punktestand-Array wird eine neue Punktestand-Zählung angefangen, aber der alte Punktestand der letzten Runde (falls vorhanden) bleibt erhalten. Per `createRandomPoints()` werden dann die Kegelpositionen erzeugt und gespeichert (Z.410). Die neuen Kegel werden gezeichnet (Z.414). In einer fußgesteuerten `while`-Schleife beginnt dann der „Schlagabtausch“; Randy zieht zuerst (Z.420), dann zieht Anna in der Bedingung der `while`-Schleife (Z.423). Der Rückgabewert der Zugfunktion entspricht damit direkt ihrer Entscheidung; bei `true` wird die Schleife fortgesetzt und bei `false` abgebrochen. Wenn ein Zwischenstand angezeigt werden soll, wird zwischen Randys und Annas Zug immer einer erzeugt (Z.419+422). Die Fortsetzung der Runden-Schleife ist ebenfalls an die Bedingung geknüpft, dass sich überhaupt noch nicht umgeworfene Kegel auf dem Spielfeld befinden (Z.423).

Randys Zugfunktion `randyZug` hat sich seit Unteraufgabe 2 nicht sonderlich verändert; lediglich die Abspeicherung der erreichten Punkte ist anders und es gibt keinen `status` mehr.

`annaZug()` - Annas Zug

Neu ist die Funktion `annaZug`; die Funktion, die nach Annas Strategie zieht.

Nach 3.1 ermittelt sie zuerst diejenige Wurflinie, die die größtmögliche Anzahl an Kegeln umwirft (Z.317-356). Dafür verbindet sie mit zwei `for`-Schleifen alle noch auf dem Spielfeld vorhandenen Kegel einmal mit einem anderen und berechnet die Anzahl der

umgeworfenen Kegel durch Errechnung der Geradenfunktion und Aufruf der Funktion `wurf()` - mit einem optionalen dritten Parameter: `$count = false`. Dies verhindert, dass die Funktion selbst die durch sie „umgeworfenen“ Kegel als umgeworfen markiert; sie läuft also praktisch im „Sandkastenmodus“ und gibt lediglich die Anzahl der theoretisch umgeworfenen Kegel zurück; „Was wäre wenn..“ (ich so geworfen hätte). Bei der Geradengleichung, bei der am meisten zurückgeworfen wird, wird dann „geworfen“; sind mehrere gleich gut, wird die erste gefundene genommen (sie wird dann nicht in Zeile 344ff. überschrieben).

```

302 function annaZug(){
303     global $kegelArray;
304     global $punkte;
305
306     nextFrame();
307     logFrame("\n--- Annas Zug ---");
308     logFrame("- Wurf -");
309
310     $besteWurfLinie = Array();
311     $besteWurfLinie["m"] = 0;
312     $besteWurfLinie["a"] = 0;
313     $besteWurfLinie["getroffen"] = 0;
314
315     logFrame("Ermittle beste Wurf-Linie");
316
317     for($p1 = 0; $p1 < sizeof($kegelArray); $p1++){
318         if ($kegelArray[$p1]["status"]){
319             if (anzahlKegelAufDemSpielfeld() != 1){
320                 for($p2 = 0; $p2 < sizeof($kegelArray); $p2++){
321                     if ($kegelArray[$p2]["status"] && $p1 != $p2){
322
323                         //Koordinaten übersichtlicher
324                         $p1X = $kegelArray[$p1]["x"];
325                         $p1Y = $kegelArray[$p1]["y"];
326                         $p2X = $kegelArray[$p2]["x"];
327                         $p2Y = $kegelArray[$p2]["y"];
328
329                         //m errechnen
330                         $dX = $p1X - $p2X;
331                         $dY = $p1Y - $p2Y;
332
333                         if ($dX == 0){ $dX = 1; } //Division durch Null ←
                                    verhindern
334
335                         $m = $dY / $dX;
336
337                         //a errechnen
338                         $a = $p1Y - ($m * $p1X);
339
340                         $getroffen = wurf($m, $a, false);
341
342                         logFrame("Es könnten ".$getroffen." Kegel ←
                                    getroffen werden.");

```

```

343         if ($getroffen > $besteWurfLinie["getroffen"]){
344             $besteWurfLinie["m"] = $m;
345             $besteWurfLinie["a"] = $a;
346             $besteWurfLinie["getroffen"] = $getroffen;
347         }
348     }
349 }
350 }
351 }else{
352     $besteWurfLinie["m"] = 0;
353     $besteWurfLinie["a"] = $kegelArray[$p1]["y"];
354 }
355 }
356 }
357
358 logFrame("Beste wurf-Linie ermittelt: y = ".$besteWurfLinie["m"]." * x ↵
359 + ".$besteWurfLinie["a"]);
360 draw($besteWurfLinie["m"], $besteWurfLinie["a"], true); //Besten wurf ↵
361 zeichnen
362
363 $anzahlUmgeworfen = wurf($besteWurfLinie["m"], $besteWurfLinie["a"], ↵
364 true); //Besten Wurf ausführen
365 logFrame("Insgesamt ".$anzahlUmgeworfen." Kegel umgeworfen.");
366 $punkte[sizeof($punkte) - 1]["anna"] += $anzahlUmgeworfen;
367
368 savePoints(); //Punktestand speichern
369
370 //Entscheidung: Runde fortsetzen?
371
372 if (punkteInsgesamt()["randy"] + (anzahlKegelAufDemSpielfeld() * 0.5) ↵
373 <= punkteInsgesamt()["anna"]){
374     logFrame("\n---- Anna hat sich für die Fortsetzung der Runde ↵
375 entschieden. ----");
376     return true;
377 }else{
378     logFrame("\n---- Anna beendet die aktuelle Runde. ----");
379     return false;
380 }
381 }

```

Ist Anna mindestens halb so viele Punkte im Vorsprung, als nach ihrem Wurf noch auf dem Spielfeld verbliebenen, riskiert sie die Fortsetzung der Runde (Z.371) oder startet eine neue Runde (Z.374); nach **3.1**.

Die Entscheidung wird per Rückgabewert der `while`-Schleife in Zeile 423 übermittelt.

`draw()`-Funktion

Von den drei Frame-Funktionen (`logFrame`, `savePoints` und `draw`) hat `draw` die Aufgabe, eine Grafik des Spielfeldes zu erstellen. Die hierfür notwendigen Informationen liegen in den globalen Variablen vor; Kegel-Array `arrayKegel`, Punktestand `punkte`, Kreisradius

kreisRadius, Auflösung des zu erzeugenden Bildes bildAufloesung und - bei einem Wurf - die Wurf-Farben wurfFarbeAnna und wurfFarbeRandy sowie die Transparenz der Wurf-Linie transparency sowie nicht zuletzt der aktuelle Frame \$currentFrame, aus dem sich der spätere Dateiname der zu erzeugenden Grafik ableitet.

```

81 function draw($m = false, $a = false, $isAnna = false){
82     global $kegelArray;
83     global $currentFrame;
84
85     global $kreisFarbe;
86     global $kegelFarbe;
87     global $wurfFarbeAnna;
88     global $wurfFarbeRandy;
89
90
91     // Bild mit Auflösung erstellen (Verhältnis: 1:1; Quadrat)
92     $bild = imagecreatetruecolor(bildAufloesung, bildAufloesung);
93
94     //Hintergrundfarbe
95     $farbe_weiss = imagecolorallocate($bild, 255, 255, 255);
96     imagefilledrectangle($bild, 0, 0, bildAufloesung, bildAufloesung, ↵
97         $farbe_weiss);
98
99     //Kreis zeichnen
100    $farbe_kreis = imagecolorallocate($bild, $kreisFarbe["r"], ↵
101        $kreisFarbe["g"], $kreisFarbe["b"]);
102    imageellipse($bild, faktor * kreisRadius, faktor * kreisRadius, 2 * ↵
103        faktor * kreisRadius - 1, 2 * faktor * kreisRadius - 1, ↵
104        $farbe_kreis);
105
106    //Kegel zeichnen
107    $farbe_kegel = imagecolorallocate($bild, $kegelFarbe["r"], ↵
108        $kegelFarbe["g"], $kegelFarbe["b"]);
109
110    foreach($kegelArray as $kegel){
111        if ($kegel["status"]){
112            imagefilledellipse($bild, faktor * $kegel["x"], faktor * ↵
113                $kegel["y"], kegelDurchmesser, kegelDurchmesser, ↵
114                $farbe_kegel);
115        }
116    }
117
118    //Evtl. Wurf einzeichnen
119    if ($m !== false || $a !== false){
120
121        $farbe = ($isAnna) ? $wurfFarbeAnna : $wurfFarbeRandy;
122
123        //Dünne Linie
124        $farbe_wurf = imagecolorallocate($bild, $farbe["r"], $farbe["g"], ↵
125            $farbe["b"]);
126        imageline($bild, 0, faktor * $a, faktor * 100, faktor * ($m * 100 ↵
127            + $a), $farbe_wurf);
128        //Dickere, transparente Linie

```

```

120     imagesetthickness($bild, 2 * faktor * durchmesserKugel);
121
122     $farbe_wurf_transparent = imagecolorallocatealpha($bild, ↵
        $farbe["r"], $farbe["g"], $farbe["b"], transparency);
123     imageline($bild, 0, faktor * $a, faktor * 100, faktor * ($m * 100 ↵
        + $a), $farbe_wurf_transparent);
124
125     imagesetthickness($bild, 1);
126 }
127
128 //Bild speichern
129 imagepng($bild, path.$currentFrame.".png");
130
131 }

```

Der dritte Parameter `isAnna` gibt bei einem Wurf an, ob Anna diesen wirft, um die entsprechende Wurf-Farbe zu verwenden. Mit dem Letzten Befehl (in Zeile 129) wird das Bild im Frame-Ordner in `path` als PNG abgespeichert: `imagepng($bild, path.$currentFrame.".png");`

Rückgabe

Wurde der Spielablauf durch die `for`- und `while`-Schleife in Z.398 und 423 zu Ende simuliert, gibt das Skript eine HTML-Webseite zurück, in dem auf das Skript des Players - `player.js` - verwiesen wird.

```

445 <html>
446   <head>
447     <!-- Stylesheet einbinden -->
448     <link href="style_3_player.css" type="text/css" rel="stylesheet" />
449
450     <!-- Titel -->
451     <title>Aufgabe 2.3 :: KI | Tim Hollmann @ 33.BwInf 2014/'15</title>
452   </head>
453   <body>
454     <div id="wrapper">
455       <div class="row">
456         <h1>Aufgabe 2.3 :: KI</h1>
457         <h2>33.BwInf 2.Runde - Tim Hollmann</h2>
458       </div>
459       <hr>
460       <div class="row">
461         <!-- Player -->
462         <div id="playerWrapper">
463           <input type="hidden" name="gameID" id="gameID" ↵
              value="<?php Echo $timestamp; ?>" />
464           <input type="hidden" name="anzahlFrames" ↵
              id="anzahlFrames" value="<?php Echo $currentFrame; ↵
              ?>" />
465           <div id="controlsWrapper">
466             <div class="controlsItem" id="imageIndicator" ↵
              style="width: 50px; text-align: ↵
              center;">0/0</div>

```



```

467         <div class="controlsItem" ↵
468             onClick="javascript:previousFrame()"><</div>
469         <div class="controlsItem" style="width: 80px;" ↵
470             id="playPauseButton" ↵
471             onClick="javascript:isPlaying = ↵
472                 !isPlaying;">-/-</div>
473         <div class="controlsItem" ↵
474             onClick="javascript:nextFrame()">>>/div>
475         <div class="controlsItem" style="border-right: ↵
476             none;">Geschw.: <input type="number" ↵
477             style="width: 50px;" id="intervalInput" ↵
478             value="150"/>ms</div>
479     </div>
480     <div id="imageBoxWrapper">
481         <img id="imageBox" src="" alt="LOADING"></img>
482     </div>
483     <div id="pointsBoxContainer">
484         <textarea readonly id="pointsBox"></textarea>
485     </div>
486     <div id="logBoxContainer">
487         <textarea readonly id="logBox"></textarea>
488     </div>
489
490     <!-- Player-Skript einbinden -->
491     <script type="text/javascript" src="player.js"></script>
492 </div>
493 </div>
494 <div class="row">
495     <div style="" align="center">
496         <h3>Insgesamt-Sieger: <?php Echo $sieger." mit ↵
497             ".$differenz." Punkten Vorsprung nach ↵
498             ".anzahlRunden." Runden"; ?></h3>
499     </div>
500 </div>
501 <div class="row">
502     <div style="width: 100px; margin: 0 auto;"><a ↵
503         href="index.html">Neues Spiel</a></div>
504 </div>
505 </div>
506 </body>
507 </html>

```

3.2.4 player.js - Anzeigen der erzeugten Spielzustände ($\hat{=}$ Frames)

Die Aufgabe des Players ist es, die erzeugten Bilder, Punktestände und Logs (\rightarrow alle Frame-Daten) nacheinander zu laden und darzustellen. Der Benutzer kann den Player auch Anhalten sowie manuell zum nächsten und vorherigen Frame springen.

```

7  var isPlaying = true;
8
9  var interval = 0;

```

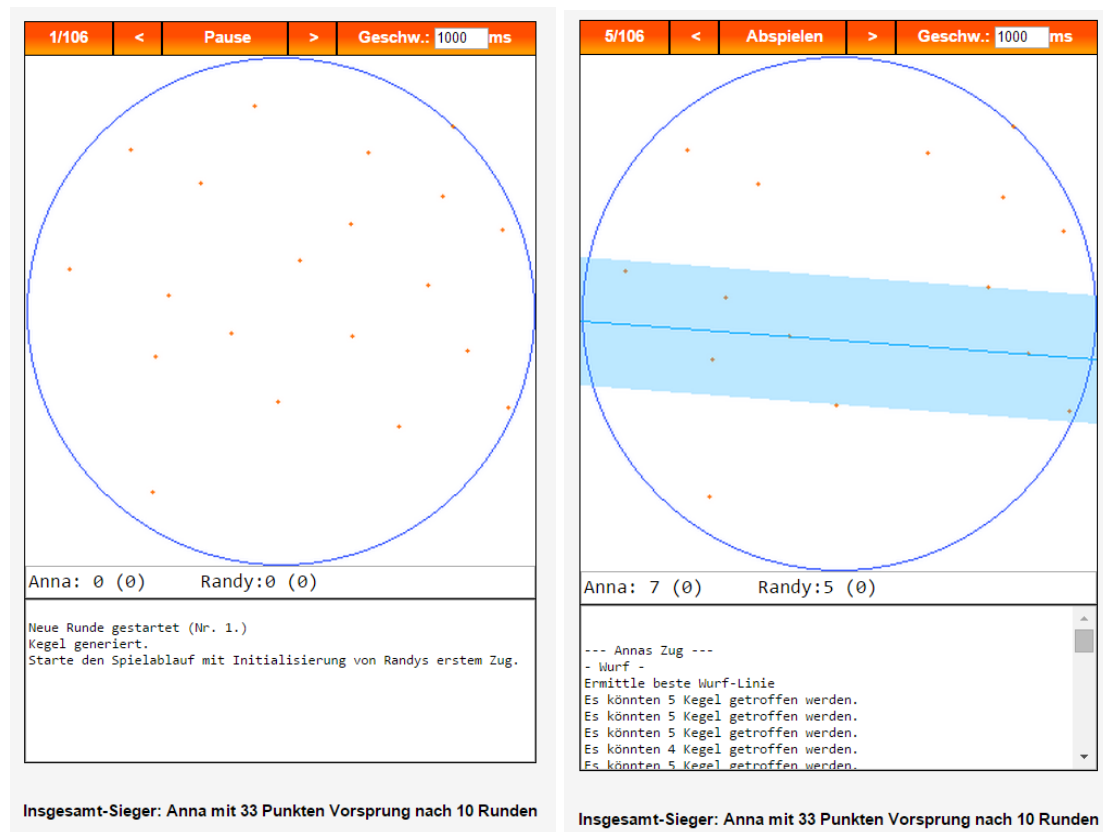


Abbildung 19: Player

```

10
11 var currentFrame = 0;
12 var playerCounter = interval;
13
14 var imageBox = document.getElementById("imageBox");
15
16 var gameId = parseInt(document.getElementById("gameID").value);
17 var anzahlFrames = parseInt(document.getElementById("anzahlFrames").value);
18
19 setInterval("playerTick()", 10);
20
21 function playerTick(){
22     playerCounter++;
23
24     document.getElementById("playPauseButton").innerHTML = (isPlaying) ? ↵
25         "Pause" : "Abspielen";
26
27     interval = parseInt(document.getElementById("intervalInput").value);
28
29     if (isPlaying){
30         if (playerCounter % interval == 0){

```

```
30         currentFrame++;
31         if (currentFrame > anzahlFrames){ currentFrame = 1; }
32         loadImage();
33     }
34 }
35 }
36
37 function loadImage(){
38     imageBox.src = gameId + "/" + currentFrame + ".png";
39     document.getElementById("imageIndicator").innerHTML = currentFrame + " von " + anzahlFrames;
40
41     //Log-Box
42     xmlhttpObjectLog.open('get', gameId + "/" + currentFrame + "_log.txt");
43     xmlhttpObjectLog.onreadystatechange = handleContentLog;
44     xmlhttpObjectLog.send(null);
45
46     //Punkte-Box
47     xmlhttpObjectPoints.open('get', gameId + "/" + currentFrame + "_points.txt");
48     xmlhttpObjectPoints.onreadystatechange = handleContentPoints;
49     xmlhttpObjectPoints.send(null);
50 }
51
52 function handleContentLog(){
53     if (xmlhttpObjectLog.readyState == 4){
54         document.getElementById('logBox').innerHTML = xmlhttpObjectLog.responseText;
55     }
56 }
57
58 function handleContentPoints(){
59     if (xmlhttpObjectPoints.readyState == 4){
60         document.getElementById('pointsBox').innerHTML = xmlhttpObjectPoints.responseText;
61     }
62 }
63 function previousFrame(){
64
65     currentFrame = (currentFrame == 1) ? anzahlFrames : currentFrame - 1;
66     loadImage();
67 }
68
69 function nextFrame(){
70     currentFrame = (currentFrame == anzahlFrames) ? 1 : currentFrame + 1;
71     loadImage();
72 }
73
74 //Ajax für Log-Box
75
76 var xmlhttpObjectLog = false;
77
78 if (typeof XMLHttpRequest != 'undefined')
79 {
```

```
80     xmlHttpRequestLog = new XMLHttpRequest();
81 }
82
83 if (!xmlHttpRequestLog)
84 {
85     try
86     {
87         xmlHttpRequestLog = new ActiveXObject("Msxml2.XMLHTTP");
88     }
89     catch(e)
90     {
91         try
92         {
93             xmlHttpRequestLog = new ActiveXObject("Microsoft.XMLHTTP");
94         }
95         catch(e)
96         {
97             xmlHttpRequestLog = null;
98         }
99     }
100 }
101
102 var xmlHttpRequestPoints = false;
103
104 if (typeof XMLHttpRequest != 'undefined')
105 {
106     xmlHttpRequestPoints = new XMLHttpRequest();
107 }
108
109 if (!xmlHttpRequestPoints)
110 {
111     try
112     {
113         xmlHttpRequestPoints = new ActiveXObject("Msxml2.XMLHTTP");
114     }
115     catch(e)
116     {
117         try
118         {
119             xmlHttpRequestPoints = new ActiveXObject("Microsoft.XMLHTTP");
120         }
121         catch(e)
122         {
123             xmlHttpRequestPoints = null;
124         }
125     }
126 }
```

Das PHP-Skript hat dem Player die Zahl der erzeugte Frames in einem versteckten Eingabefeld übermittelt (vgl. Z.464), da diese für JavaScript sonst nur sehr schwer zu ermitteln wäre (JavaScript hat fast keine Rechte, auf Dateisysteme zuzugreifen, erst recht nicht auf das eines Servers).

Per `setInterval` wird eine Ticker-Funktion `playerTick()` gestartet (Z.19), die alle 10 Millisekunden tickt. Immer dann, wenn der bei jedem Tick inkrementierte `playerCounter` ganzzahlig durch den vom Benutzer eingegebenen Player-Intervall `interval` teilbar ist

$$\text{playerCounter} \bmod \text{interval} = 0$$

(Z.29), wird `currentFrame` inkrementiert und per Funktion `loadImage` der nächste Spielzustand geladen; der Name ist hier irreführend, da nicht nur das Bild, sondern auch alle anderen Framedaten per *HTML*- und *AJAX*-Request geladen werden. Per `nextFrame()` und `previousFrame()` wird `currentFrame` inkrementiert bzw. dekrementiert und dann wieder per `loadImage` der Spielzustand geladen und dargestellt; diese Funktionen sind mit den Vor- und Zurück-Buttons verbunden. Wenn man auf den Pause-Button klickt, wird `isPlaying` auf `false` gesetzt und daraufhin wird `playerCounter` in `playerTick()` nicht mehr inkrementiert. Bei einem erneuten Klick ist `isPlaying` wieder `true`.

3.3 Beispiele

(\Rightarrow Abb. 20)

3.4 Ausführen des Programmes

Rufen Sie die Datei `index.html` in einem Ordner mit der Datei `game.php` und den dazugehörigen Skripts und Stylesheets mit einem beliebigen Browser von einem Webserver ab, der PHP unterstützt und diesem den Schreibzugriff im aktuellen Verzeichnis gewährt.

Falls Sie gerade keinen Webserver zur Hand haben, finden Sie meine Einsendung auch unter <http://www.tim-hollmann.de/BwInf/33/2/Einsendung/>. Dort können Sie auch das PHP-Skript ausführen.

Wenn sie eine sehr hohe Anzahl an Runden simulieren wollen (>40), müssen Sie ggf. die Option `max_execution_time` in der `php.ini` erhöhen (die maximale Laufzeit), da dann eventuell eine längere Laufzeit als 30 Sekunden (der Standardwert) benötigt wird.

4 Aufgabe 4

(\Rightarrow Abb.21) Man sieht hier sehr deutlich, dass meine Strategie ab einem Spielfeld-Radius > 2 immer erfolgreich ist. Für $r \leq 2$, $r = 20$ eine garantierte Gewinnstrategie zu finden, ist schwierig und war auch nicht zwingend notwendig¹

¹Siehe EI-Community-Forum und www.bundeswettbewerb-informatik.de/neuigkeiten/artikel/2-runde-aufgabe-panoramakegeln

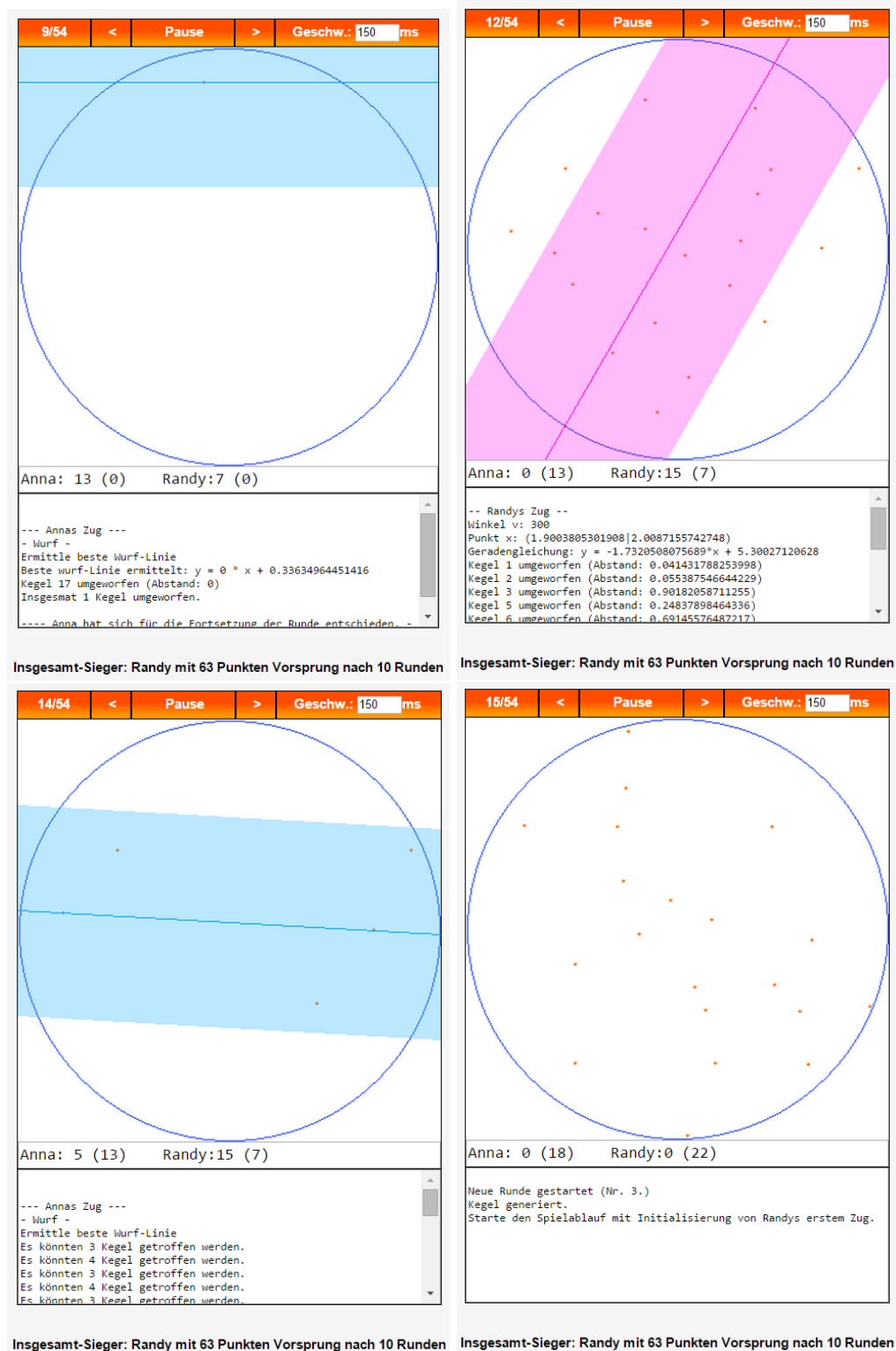


Abbildung 20: Beispiele

Spielfeld-Radius	Kegel-Anzahl	DF1	DF2	DF3
2	20	-75	-5	-40
	30	-99	-107	-121
	40	-37	-125	-126
	50	-225	-88	-91
3	20	7	60	58
	30	23	76	33
	40	110	43	68
	50	70	53	87
4	20	54	83	85
	30	162	116	142
	40	124	196	79
	50	202	94	116
5	20	112	89	104
	30	155	185	166
	40	205	202	222
	50	223	219	252
6	20	124	146	110
	30	150	173	143
	40	191	265	226
	50	235	275	262

Abbildung 21: Punkt-Vorsprung bei anderen Kegelanzahlen und Radien; es wurden pro Durchführung 20 Spielrunden gespielt. Negativer Vorsprung = Randy gewinnt