Dokumentation Projektgruppe 065

Gruppenmitglieder:

- *Yusuf Emre Berber*
- *Christoph Lukas Berst*
- *Jan Kai Braun*
- *Julian Imhof*

Inhaltsverzeichnis:

3.1.2 Kanten bilden (Burgen verbinden)

Für jede Burg wird zuerst eine Liste erstellt, die in aufsteigender Reihenfolge sortiert, nach ihrem Abstand zur der aktuell betrachteten Burg, alle anderen Burgen enthält. Damit die Linien zwischen den Burgen nicht irgendwie kreuz und quer entstehen, wird die Burg mit bis zu 4 ihrer nächsten Burgen (meistens zwischen 2 und 4 Nachbarburgen) verbunden. Dies geschieht mit ein paar Einschränkungen:
Wenn die Burg schon mit einer Nachbarburg verbunden ist, wird stattdessen eine andere Burg als Partner gesucht, damit die Burg immer genügend Verbindungen zu anderen Burgen hat. Dies soll das Spiel ausgeglichen halten. Rein lineare Kämpfe zwischen zwei Gebieten haben schon in Risiko dafür gesorgt, dass es Regionen gab, die man nur über einen einzigen Punkt mit Truppen zu stopfen konnte, um eine allumfassende Verteidigung herzustellen, ähnlich wie eine Flasche an ihrem Hals mit einem einfachen Korken. Damit jedoch Gebiete nicht zu viele Angriffsvektoren besitzen, wird einer Burg keine weitere Angriffsmöglichkeit hinzugefügt, sobald sie das vorgesehene Maximum (4 andere Burgen) an Nachbarn besitzt.
Es gibt zwar in Ausnahmefällen überkreuzende Linien, jedoch halten sich diese in Grenzen aufgrund der vorher genannten Einschränkungen. Besonders mit der Einschränkung auf nur ein paar Nachbarburgen gibt es im Prinzip keine Linien quer über die Karte.

3.1.3 Iteratoren und Wege finden I

getSmallestNode(): Es wird durch alle *availableNodes* gegangen und so in einem durchlauf die im Werk kleinste, nicht negative korrespondierende *AlgorithmNode* gefunden und am Ende zurückgegeben. Wurde keine gefunden, wird wie in der Aufgabenstellung verlangt *null* geliefert.
run(): Es wird jeden Knoten der *availableNodes* verwendet, um den neuen Wert aller Nachbarknoten zu bestimmen. Der neue Wert wird allerdings nur übernommen, sollte der Weg zum benachbarten Knoten passierbar und der Knoten selbst entweder noch keinen Wert oder einen größeren als den Neuen haben. Sind diese Bedingungen erfüllt, wird auch der *Vorgängerknoten* des benachbarten Knoten gleich dem aktuellen Knoten gesetzt.
getPath(Node): Es wird ein vier Elementiger Iterator gebaut, der aus dem Hauptknoten, dessen p*revious Node* und den jeweils korrespondierenden *AlogorithmNodes* besteht. In jedem Iterationsschritt, wird der Hauptknoten auf die entsprechende *previous Node* gesetzt und alle anderen Elemente angepasst. Des Weiteren wird jeweils die Kante zwischen den Knoten einer Liste hinzugefügt, welche am Ende umgedreht und zurückgegeben wird.

3.1.4 Wege finden II

A. $f \in O(n^2 * n)$

Man gehe von dem Fall aus, dass in einem Graphen jeder Knoten mit jedem anderen Knoten und sich selbst verbunden sei:
Die äußere Schleife wird im wesentlichen durch “getSmallestNode()” bestimmt, welches sich durch das Entfernen der Elemente nach dem Suchen wie Selectionsort verhält, welches quadratisch wächst. Da nun jeder Knoten mit jedem Knoten verbunden ist, durchläuft die innere Schleife auch n Kanten. Die Bedingungen, ob etwas passierbar ist, haben nur eine geringe Auswirkung auf die Laufzeit und können daher vernachlässigt werden.

B. $f \in O(T(n) * n)$

$T(n)$ hängt von der List, welche in “getSmallestNode()” verwendet wird ab.

Wenn man nun eine sortierte Liste wie eine “SortedList” oder eine “PriorityQueue” verwenden würde, bei der sogar ein geschickt gewählter Comparator verwendet wird, welcher die negativen Zahlen direkt ans Ende der Liste einsortiert, wäre der Worst-Case nur noch $O(n)$, da man am ersten Element schon erkennen kann, ob das gesuchte Element vorhanden ist.

C. Invariante: Nach $h \geq 0$ Durchläufen und der Datenmenge n gilt:

a. Liste von Knoten, deren Wert kleiner 0 ist, und maximal $(n-h)$ Knoten, welche in den vorherigen h Schritten nicht den kleinsten Wert hatten.

b. Knoten, deren Wert kleiner 0 ist oder Knoten, welche mindestens den $h+1$ niedrigstens Wert besitzen, der nicht negativ ist.

c. h Knoten, deren Wert definitiv positiv ist und welche nicht mehr in der Liste “availableNodes” vorhanden sind.

3.1.5 Kürzester Pfad zu allen Knoten

A. Ein negativer Zykel setzt voraus, dass ein negatives Kantengewicht im Graphen vorhanden ist. Dies hätte zur Folge, dass Kantengewichte die Distanz zwischen zwei Knoten nicht korrekt widerspiegeln. Der Distanzbegriff aus der Mathematik ist der nicht-negative Abstand zwischen zwei Objekten. Die Distanz zwischen verschiedenen Knoten oder die Länge der Kanten zwischen diesen Knoten würde demnach nicht korrekt bestimmt werden können. Weiterhin ist es durch den negativen Zykel möglich wieder und wieder die Summe der Knoten zu verringern und dadurch einen kürzeren Pfad zu bilden. Damit würde nach *n-1* Durchläufen nicht die Länge des kürzesten Pfades von v nach w in $M^h(v,w)$ enthalten sein, wobei $v,w \in V$.

B. Best-Case: $\Theta(n-1 * T(n))$

Worst-Case: $\Theta(n-1 * T(n))$

$T(n)$ beschreibt die Komplexität des Algorithmus der die Länge des kürzesten Pfades mit maximal h + 1 Kanten für jeden Eintrag der Matrix berechnet.

Die äußere Schleife bricht in jedem Fall erst nach $n-1$ Durchläufen ab.

3.3.1 Computergegner

Quelle: https://www.youtube.com/watch?v=6VBCXvfNICM

Die Grundlage der AdvancedAI ist ein Baumdiagramm, die aus 5 Komponenten besteht : Root,Selector,Sequence,Leaf und Blackboard

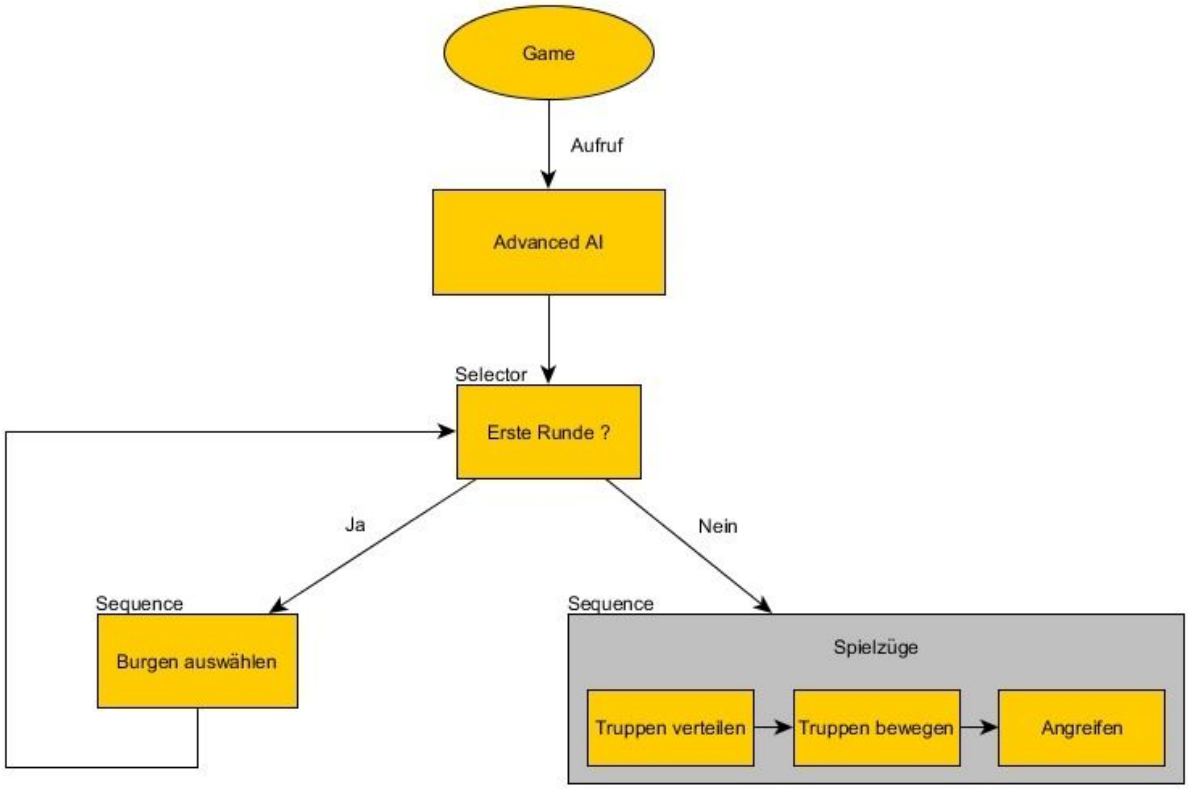
Blackboard : Das Blackboard beinhaltet Daten, die die AI braucht um entscheidungen zu treffen(Bspw. Eigenschaften von Objekten)

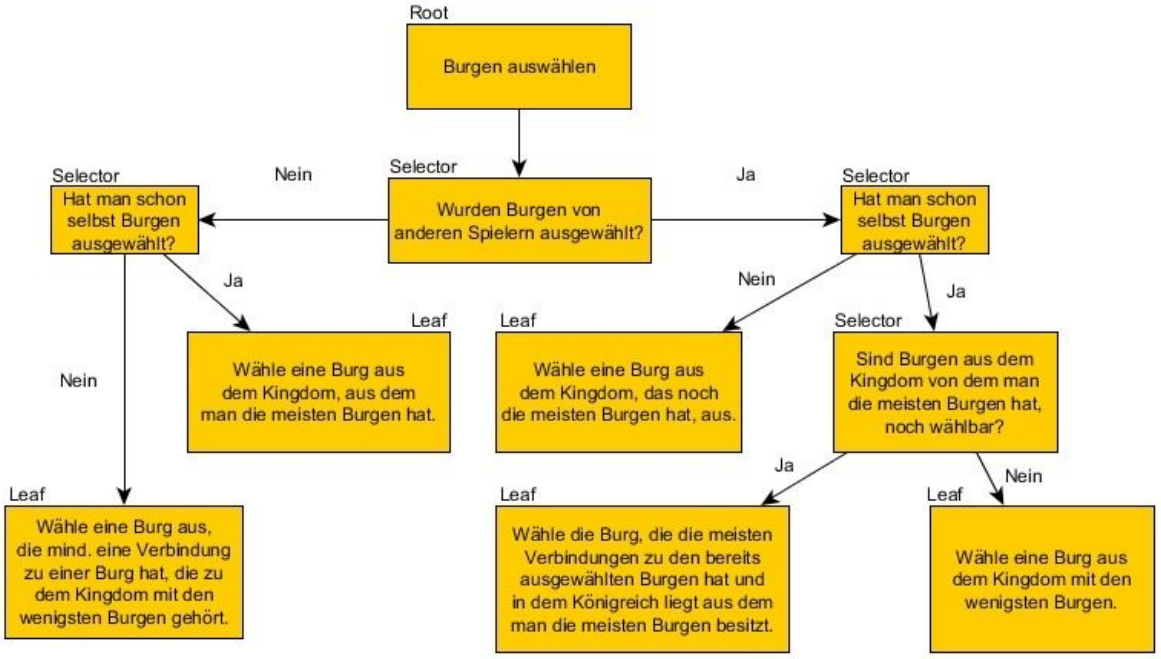
Root : Root stellt den Start der AI da, bzw. die Aktivierung

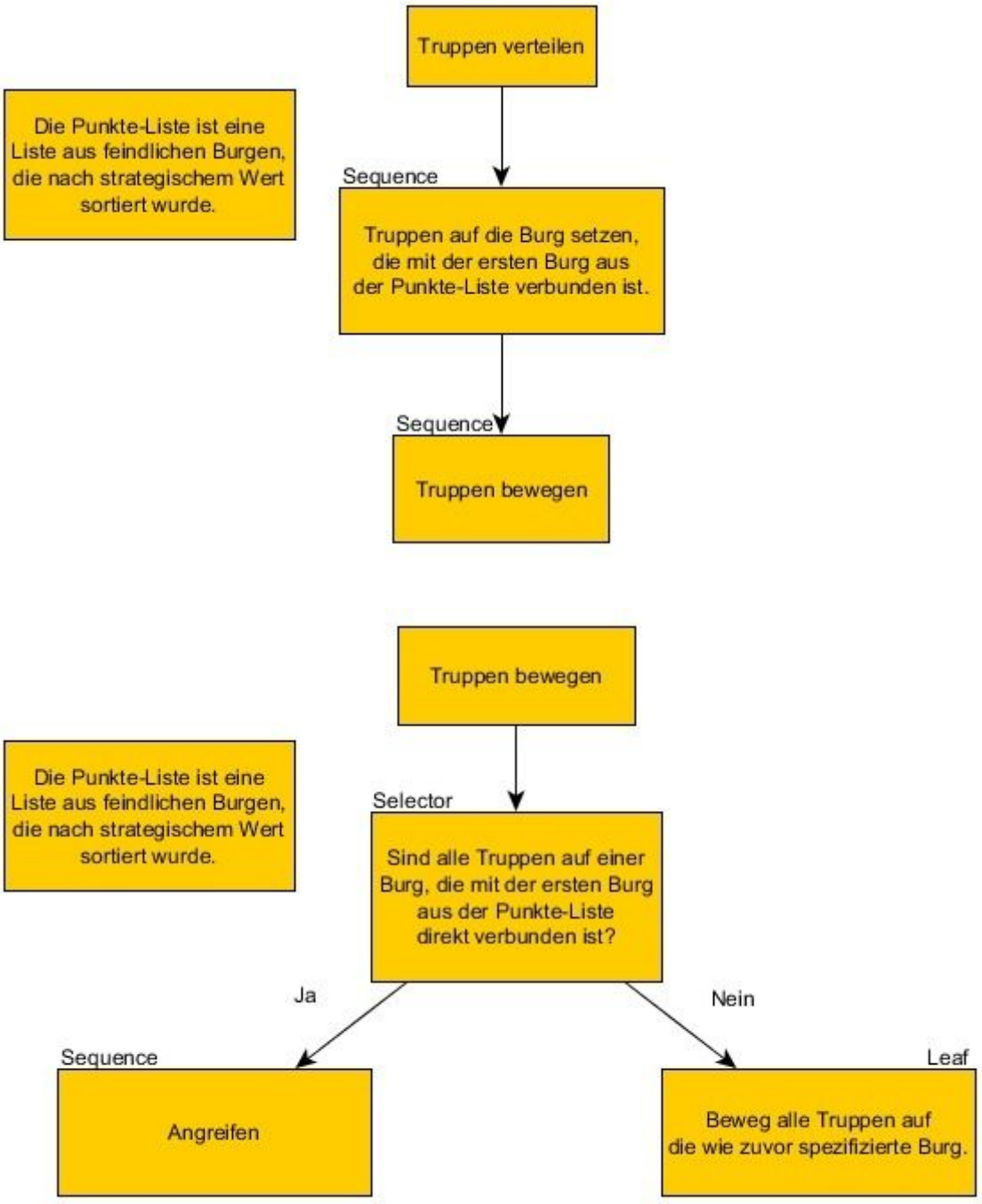
Selector : Ein Selector stellt eine Ja/Nein-Frage da, die (wenn benötigt) anhand der Daten aus dem Blackboard beantwortet wird. Nach der Beantwortung wir ein Selector,eine Sequence oder ein Leaf aufgerufen.

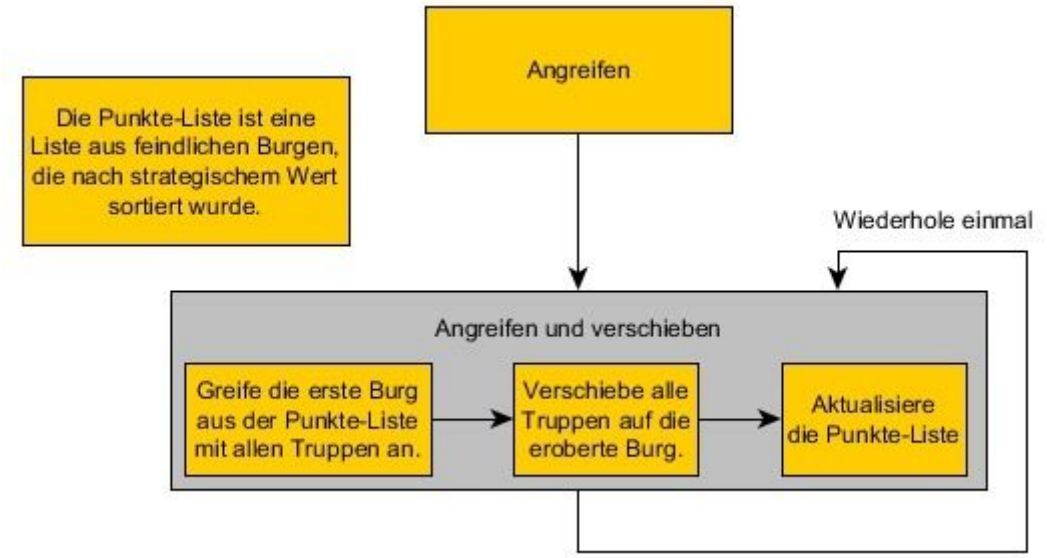
Sequence : Eine Sequence kann mehrere Selectors,Sequences oder Leafs aufrufen kann ohne eine Ja/Nein-Frage zu beantworten.Die Komponente werden von Links nach Rechts aufgerufen.
Leaf : Ein Leaf ist die eigentliche Handlung, die die AI ausführt.

Die AI besteht also aus Entscheidungsfragen,die durch Informationen die ihm zur verfügung gestellt werden,beantwortet.






Truppen verteilen
Die Punkte-Liste ist eine Liste aus feindlichen Burgen, die nach strategischem Wert sortiert wurde.
Sequence
Truppen auf die Burg setzen, die mit der ersten Burg aus der Punkte-Liste verbunden ist.
Sequence
Truppen bewegen
Truppen bewegen
Die Punkte-Liste ist eine Liste aus feindlichen Burgen, die nach strategischem Wert sortiert wurde.
Selector
Sind alle Truppen auf einer Burg, die mit der ersten Burg aus der Punkte-Liste direkt verbunden ist?
Ja
Nein
Sequence
Angreifen
Leaf
Beweg alle Truppen auf die wie zuvor spezifizierte Burg.



P.S : Manchmal kann die BasicAI mehr Burge auswählen(durch die Anzahl der Burge, die Random ist), wodurch er mehr Truppen hat,womit die Chance sich erhöht, dass sie gegen die AdvancedAI gewinnt.

3.3.4 Anpassung für Studierende im Studiengang CE

Würfel selbst:

Ein Würfel besteht wie in der Vorlage aus 8 Punkten und 22 Kanten. Die Punkte dienen der Kollisionserkennung, Bewegung und dem Rendering. Die Kanten sind dafür verantwortlich, dass der Würfel seine Form behält. Die Welt

dice3d.main.World

in der sich alle Würfel, sowie der Boden, befinden, tickt alle 15 ms. Pro Tick wird folgendes ausgeführt:

- Die Position eines jeden Objektes wird aktualisiert, sollte er sich im vorherigen Tick bewegt haben. Dies dient dazu nur noch fallende Würfel zu beachten und Leistung zu sparen.
    - Die Bewegung des vorherigen Ticks wird komplett übernommen und die Gravitation der Welt wird addiert, um eine realistische Fallbewegung mit konstanter Beschleunigung zu simulieren.
- der Würfel wird ein paar (4) mal repariert
    - alle Kanten werden auf ihre Originallänge gebracht, indem die Punkte entsprechend auf der die Kante beinhaltenden Gerade verschoben werden.
- Zu jeder Reparatur wird überprüft, ob aus der Bewegung eine Kollision mit einem ANDEREN Objekt in der Welt hervorging. (s.u.)
    - Dies wird überprüft, indem berechnet wird, ob einer der eigenen Punkte innerhalb eines anderen Objektes liegt.
- anschließend wird aus der Summe aller y Werte einer Ebene die oben liegende Ebene ermittelt.

Asynchron dazu muss regelmäßig die draw jedes Elements der Welt aufgerufen werden, um ihn auch anzuzeigen. Tritt in einem Update eine Kollision ein, so wird als erstes entschieden, ob es sich um eine Kollision mit dem Boden oder mit einem anderen Würfel handelt. Ist es der Boden, so wird einfach der Kollidierte Punkt in seiner Höhe wieder auf Bodenhöhe

gesetzt und die Position des vorherigen Ticks wird so manipuliert, dass im nächsten Tick eine Bewegung in die entgegengesetzte Richtung mit besonders starker y Richtung, damit der Würfel auch tatsächlich nach oben fliegt. Aus der “Kantenreparation” folgt somit ein Würfel mit realistischer Physik.
Handelt es sich um eine Kollision mit einem anderen Würfel, wird dieses leider aus Performancegründen ignoriert. Um dies auszugleichen erhalten die Würfel, sollten mehrere geworfen werden, ihre zufällige Startbewegung so, dass eine Kollision möglichst ausgeschlossen ist.
Jeder Würfel kann zu jeder Zeit asynchron zurückgesetzt oder versteckt werden. Beim zurücksetzen erhält der Würfel seine ursprüngliche Position und jeder Punkt einen zufälligen Startvektor, damit der Würfel möglichst zufällig in die für ihn vorgesehene Richtung fällt. Um den Würfel zu verstecken, wird diese auf eine Position außerhalb des Sichtfeldes gesetzt und die Position des vorherigen Ticks auf die Selbe gesetzt, um keine weiteren Updates des Würfels zu triggern.
Um möglichst Clippingfehler zu vermeiden, wird bei jedem draw die Position aller Punkte gespiegelt und die Kopie gezeichnet, um zu verhindern, dass sich die Punkte verändern, während sie gemalt werden.
Der Würfel wird koloriert, in dem jeder Pixel einer jeden sichtbaren Textur in einem 1x1 Polygon genau wie in der Vorlage auf seine entsprechende 2D Position projiziert wird.
Zur besseren Sichtbarkeit, wird der Boden nicht gezeichnet, sondern nur die auf diesen projizierten Schatten.

```
43 ~ " ~ 782865  \ -:- /  1: 129717, 2: 111520, 3: 141714, 4: 144450, 5: 137942, 6: 117522
43 ~ " ~ 782866  \ -:- /  1: 129717, 2: 111520, 3: 141715, 4: 144450, 5: 137942, 6: 117522
43 ~ " ~ 782867  \ -:- /  1: 129717, 2: 111521, 3: 141715, 4: 144450, 5: 137942, 6: 117522
43 ~ " ~ 782868  \ -:- /  1: 129717, 2: 111522, 3: 141715, 4: 144450, 5: 137942, 6: 117522
```

15 stündiger Test dreier Würfel. v.l.n.r. Kollisionen, Anzahl an Würfen mit 3 Würfeln, Anzahl der Augenzahlen

Würfel Integration:

Die Idee war auf Grund der eher großen Animation, den Wurf direkt auf dem Spielfeld durchzuführen, aber die zur Zeit oben liegende Seite auch weiterhin in dem bereits vorhandenen Feld anzuzeigen. Dazu verfügen nun

gui.views.GameView,
gui.components.DicePanel und
gui.components.MapPanel

über eine einheitliche

dice3d.main.World

Instance, die durch die GameView Initialisierung instanziert wird. Wird nun

gui.components.DicePanel#generateRandom(int, boolean)[1]

aufgerufen und ist die boolean WAHR, so werden über

dice3d.main.World#roll(int)

die Würfel geworfen und entsprechend animiert. Solange noch nicht alle Würfel gelandet sind, wird die alte Ansicht am rechten oberen Rand entsprechend aktualisiert. Sind alle Würfel gelandet, werden alle Werte übernommen und nach kurzer Wartezeit die Würfel wieder entfernt.

[1]gui.components.DicePanel#generateRandom(int, boolean) ist die bereits in der Vorlage verwendete Funktion um mit einer Animation zu würfeln und liefert eine Liste an Integern der gewürfelten Zahlen.

Quellen:
https://github.com/leonardo-ono/JavaVerletIntegration3DTest
http://codeflow.org/entries/2010/nov/29/verlet-collision-with-impulse-preservation/
https://math.stackexchange.com/questions/1472049/check-if-a-point-is-inside-a-rectangular-shaped-area-3d
https://stackoverflow.com/questions/5666222/3d-line-plane-intersection

Änderungen der Vorlage (außerhalb der Aufgaben)

- Würfel werden im Chat in absteigender Reihenfolge sortiert
    - gui.views.GameView#onAddScore(Player, int)
- Wenn ein Angriff fehlschlägt und man die nächste Runde starten will, wurde eine NullPointerException geworfen
    - gui.components.MapPanel#clearSelection()
- Wenn man nicht mit allen Truppen angreifen wollte, wurde der Kampf trotzdem so lange geführt, als hätte man mit allen Truppen angreifen wollen
    - gui.AttackThread#run()
- Angreifen sollte nur möglich sein, wenn mindestens eine Truppe weniger ausgewählt wurde, als auf der angreifenden Burg vorhanden ist
    - game.Game#startAttack(Castle, Castle, int)
    - gui.components.MapPanel#mouseClicked(MouseEvent)
    - game/players/BasicAI#actions(Game)
- Balancing:
    - KI konnte eigene Truppen durch Feindburgen verschieben
        - game/players/BasicAI#actions(Game)
        - game/Game#isPath(Castle, Castle, ACtion)
    - KI welche einen zufällige Burg am Feindgebiet wählt und mit voller Wucht (allen Truppen) dort angreift
        - game/players/RandomAI
    - ~~Der Spieler mit der wenigstens Burgen beginnt die zweite Runde (also darf als Erster ziehen und angreifen)~~
- nützliche Methoden
    - base.Graph#getEdge(Node, Node)
    - base.Graph#getNodes(Node)