Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

AI for Games and Interactive Systems
Kipifub

Sebastian Kindt
s0555665

Prof. Dr. Tobias Lenz

Sommersemester 2017

Contents

Chapter 1

Kipifub - 9 Pinsel - 3 Spieler - 1 Kunstwerk

1.1 Einleitung

Kibifub ist ein Strategiespiel mit 3 Spielern, jeder Spieler hat 3 Pinsel und das Ziel ist es die meiste Fläche mit der eigenen Farbe einzufärben.

Figure 1.1: initial abgezeichnetes Spielfeld

1.2 Repräsentation

Als Repräsentation des Spielfeldes habe ich ein zweidimensionales Array gewählt. Die erste Dimension sind die X Reihen und die zweite Dimension der Y Wert der X Reihe. Also ein Punkt mit den Koordinaten (x: 30, y: 30) kann über "feld[x: 30][y: 30]" abgefragt werden. Es ist keine eins zu eins Repräsentation des original Spielfeldes, sondern eine skalierte. Ich habe jeweils 16x16 Pixel zu einem großen "pixel" als eigenes Model zusammengefasst. Somit sieht die Repräsentation wie folgt aus:

```
//Das Spielfeld wird als 2dimensionales Array dargestellt
private ScaledField[][] game = new ScaledField[1024/16][1024/16];
```

Um die Information zu erhalten wie das Spielfeld aussieht, muss mit Hilfe des Networkclients Pixel für Pixel abgefragt werden, ob ein Feld begehbar ist oder nicht. Dies passiert allerdings nur einmal zu Anfang des Spiels. Um das Feld in die skalierte Variante zu übersetzen ohne später fehlerhafte Felder

zu haben, muss die Begehbarkeit eines Feldes genauer betrachtet werden. Da ein skaliertes Feld 256 Pixel zusammenfasst und ein Pixel von diesen als nicht begehbar markiert sein kann, muss sobald nur ein Pixel so markiert ist, das gesamte skalierte Feld als nicht begehbar markiert werden. Alle anderen Felder werden mit einem Ausgangswert besetzt. Als Resultat ergibt sich eine verpixelte Variante des eigentlichen Spielfeldes.

1.2.1 ScaledField

ScaledField ist die Darstellung des Skalierten Feldes. X&Y sind die Pixel der linken oberen ecke des Quadrats, die echten Koordinaten. Zudem gibt es ein Feld "isWalkable" und einen Score. Um das richtige Feld im skalierten Spielfeld zu finden muss jeweils der X und Y wert durch den skalier Faktor geteilt werden:

```
    private ScaledField getField(int x, int y) {
        return game[x / SCALE][y / SCALE];
     }
```

1.3 Updaten des Spielfeldes

1.3.1 Wahl der Darstellung

Die Darstellung verpixelt das eigentliche Spielfeld. Somit sind Änderungen die über ColorChange kommen nicht eins zu eins übertragbar. Es muss überlegt werden, wie die Informationen verarbeitet werden. Die Darstellung die ich gewählt habe ist eine doch sehr andere als der Server es darstellt. Der Hauptunterschied ist, dass es nur zwei Arten von Spieler gibt. Einmal die Pinsel von meinem Spieler und die Pinsel von den anderen Spielern. Es wird also nicht für jeden Spieler, für jede Farbe unterschieden, sondern nach eigenen Spieler und Gegner.

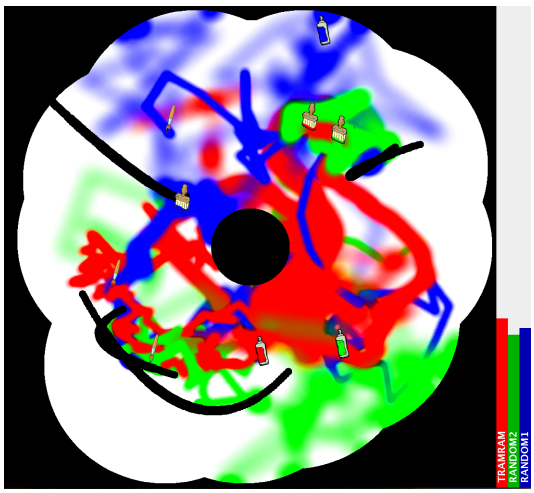


Figure 1.2: Darstellung nach einem Spieldurchlauf

Das zufällig generierte Spielfeld mit Hindernissen ergibt ein weiteres Problem. Die Pfade die, die Pinsel gehen können sind nicht immer gleich. Sie müssen zur Laufzeit ermittelt werden. Daher wird es eine Wegfindung geben. Ich habe dafür den A* mit Heuristik ausgewählt. A* bietet eine sehr schnelle Berechnung des Pfades und kann somit zu jedem Zeitpunkt für jede Spielfeldkonfiguration angewendet werden. A* hat die Eigenschaft, dass die Kanten mit den geringsten kosten abgelaufen werden. Geringe Kosten heißt kleine Werte. Und Kanten über die der Pfad nicht verlaufen soll werden mit hohen Kosten, also hohen Werten besetzt.

1.3.2 Ausblick

Verbindet man nun das Unterscheiden in Spieler & Gegner, A* und ColorChange ergibt sich eine gute Möglichkeit. Jedes Feld (ScaledField) hat einen Score. Dieser Score sind die Kanten kosten für A*. A* benötigt einen geringen Score für Kanten bei denen es sich lohnt langzulaufen und einen höhen bei denen es sich nicht lohnt. Aus diesem Grund habe ich einen festen Bereich gewählt in dem sich der Score eines Feldes bewegen kann. Zu beachten ist auch, dass A* nicht mit negativen Werten umgehen kann.

1.3.3 Berwerten der Informationen von ColorChange

Was lohnt sich und was lohnt sich nicht? Felder auf denen sich die eigene Farbe bereits befindet lohnen sich nicht, denn es kann nicht weiter mit der eigenen Farbe eingefärbt werden. Felder auf denen sich viel Farbe von Gegner befindet lohnen sich genauso wie weiße Felder. Felder mit Farben von Gegner haben aber noch eine weitere Eigenschaft, wenn man sie umfärbt, verliert der Gegner dabei Punkte.

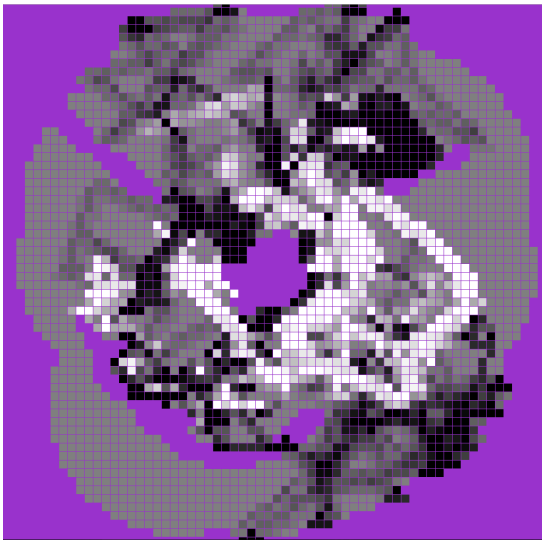


Figure 1.3: Darstellung nach einem Spieldurchlauf

Das Ganze lässt sich sehr gut mit Graustufen darstellen. Der maximale Score ist Weiß und der minimale Score ist Schwarz, alles dazwischen sind Graustufen. Damit beim Initilisieren die Darstellung stimmt, werden alle Felder mit der Hälfte des MaxScores initialisiert.

```
public static double maxScore = 12;
public double score = maxScore / 2;
```

Chapter 2

Push basierte Architektur

Die Art und Weise wie die Kommunikation zwischen Server und Client stattfindet, bietet eine gute Gelegenheit ein push-Event basiertes System zu gestalten. Beim Connecten vom Client zum Server findet eine Initialisierungsphase statt. In dieser Phase werden alle Informationen über das ausgangs Spielfeld beschafft. Der Austausch geschieht bevor das Spiel gestartet wurde.
Nachdem das Spiel gestartet wurde, können die Änderungen, die von allen Pinseln gemacht werden, durch den Client erfragt werden. Diese Abfrage dient als Einstiegspunkt für das Pushsystem.

Zu Beginn ist eine Schleife die alle ColorChange Events (1) zieht. Sobald ein ColorChange gepulled

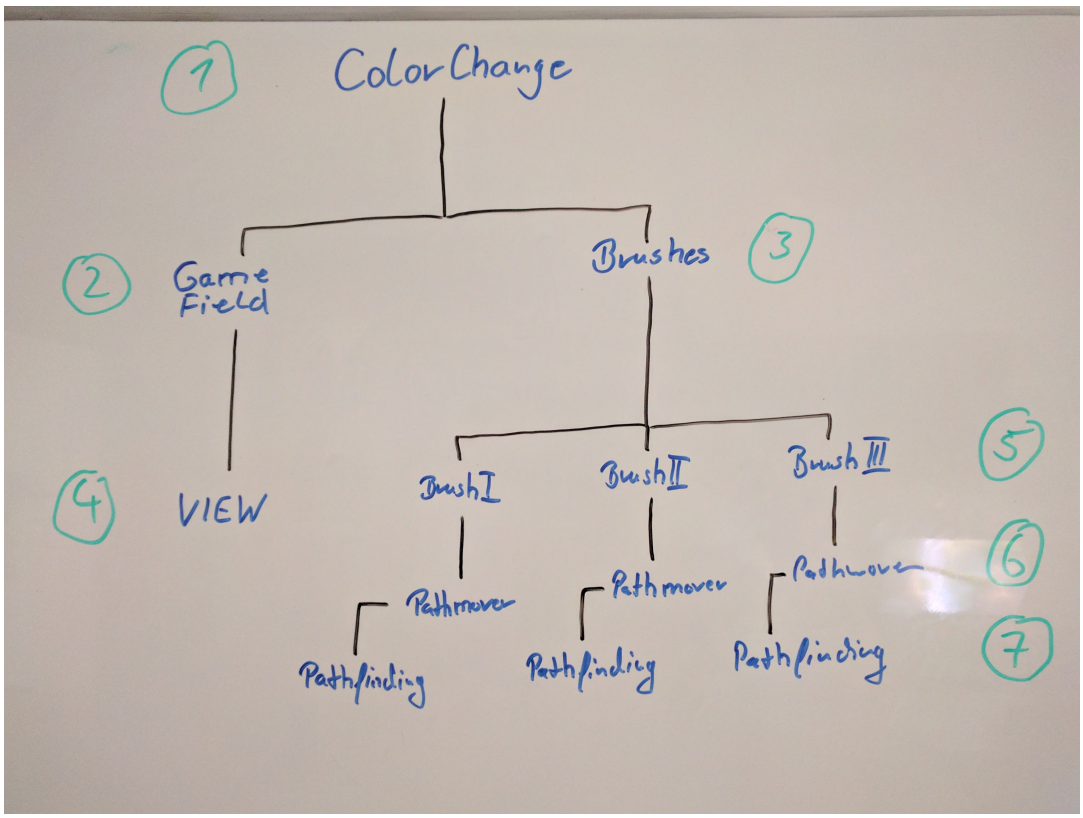


Figure 2.1: Events die durch ColorChange angestoßen werden

wurde, werde die nächsten beiden Events angestoßen. Das Spielfeld (2) verarbeitet die Änderung der Farbe, wendet diese auf das Spielfeld an. Mit den geänderten Werten wird die View (4) benachrichtigt und diese updated dann die Felder. Das andere Event welches auch von ColorChange angestoßen wird, ist ein Controller der die Position der Pinsel (3) verfolgt und updated. Sobald sich die Position eines Pinsels verändert hat, werden alle Pinsel benachrichtigt. Jeder Pinsel (5) filtert, ob es sich um den Pinsel handelt, den er darstellt. Zu jedem Pinsel gibt es einen Pathmover (6) der sich darum kümmert die Richtungen zu setzen, je nachdem ob ein Wegpunkt erreicht wurde. In bestimmten Zeitintervallen wird der Pfad neu berechnet den der Pinsel zu gehen hat.

2.1 Events Observen

Das Pushsystem wird durch das Observer Pattern implementiert. Jede Node ist ein Subject und die Ebenen 2-7 zugleich Observer. Zu beginn muss diese Relation einmal implementiert werden.

```
//ColorChange
colorChangeController = new ColorChangeController();
//Game Field
game = new GameField(myPlayerNumber);
game.observe(colorChangeController.connect());
//View
view = new ViewController();
view.observe(game.connect());
//Brushes
brush = new BrushController(myPlayerNumber);
brush.observe(colorChangeController.connect());
//Brush I & pathmover & pathfinder
PathMover mover = new PathMover(brush.getMyBrushOne(), game, client);
mover.observe(brush.connect());
//Brush II & pathmover & pathfinder
PathMover mover2 = new PathMover(brush.getMyBrushTwo(), game, client);
mover2.observe(brush.connect());
// Brush III & pathmover & pathfinder
PathMover mover3 = new PathMover(brush.getMyBrushThree(), game, client);
mover3.observe(brush.connect());
//start pull of events
colorChangeController.observeColorChange(client);
```

2.2 ColorChange (1)

Die Wurzel des gesamten "push" Baumes ist das ColorChange Event. Solange das Spiel läuft wird das Event permanent in einer Schleife abgefragt. Sobald eine Änderung stattfindet werden die Observer benachrichtigt.

```
        ColorChange colorChange;

        while (client.isAlive()) {
            if ((colorChange = client.pullNextColorChange()) != null) {
                subject.onNext(colorChange);
            }
        }
```

2.3 Spielfeld (2)

Das Spielfeld kümmert sich darum jedes ColorChange Event zu übernehmen und in die richtige Darstellung zu übersetzt. Jeder Pinsel hat bestimmte Eigenschaften die auch unterschiedlich verarbeitet werden. Auch diese Eigenschaften müssen in das gewählte Format übersetzt werden. Der langsame große Pinsel malt deutlich mehr Fläche mit der gleichen Intensität wie der kleine Pinsel ein. Der kleine Pinsel färbt nur das Feld ein auf welchem er zum Zeitpunkt des Events steht. Der große Pinsel malt zusätzlich zu dem Feld wo er sich befindet, auch die Felder oben, rechts, unten und links ein. Die Spraydose, der dritte Pinsel, wird genauso verarbeitet wie der große Pinsel nur mit geringerer Intensität. Das die Pinsel genau diese Felder einfärben hängt sehr von der Wahl der Skalierung ab. Die Wahl der Felder zum Einfärben ist also genau auf die Auflösung von 16x16 Pixel angepasst.

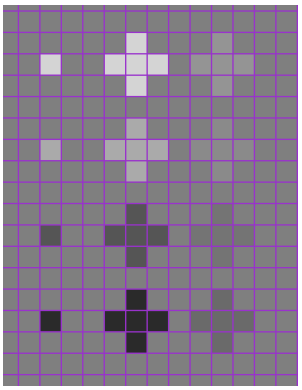


Figure 2.2: Darstellung der Pinsel

Konkret betrachtet gibt es einen Bereich von 0 - 12 für den Score eines Feldes. Der große und kleine Pinsel verändern mit jedem Event den Score um einen Wert von 2 und die Spraydose einen Wert von 0.5. Die Figure 3.1 zeigt angewendete ColorChanges. In der Mitte ist der große Pinsel, links der kleine und rechts die Spraydose. In der ersten Reihe wurden die Pinsel zweimal für den eigenen Spieler eingezeichnet, in der 2ten Reihe einmal. In der dritten Reihe wurden die Pinsel einmal für einen Gegner eingezeichnet und in der 4ten zweimal für einen Gegner. Die eingetragenen Änderungen der Felder werden jeweils pro Feld weiter an die Observer übergeben.

2.4 View (4)

Die veränderten Felder werden in der View neu / überzeichnet. Um die Felder korrekt in die View einzuzeichnen benötigt es lediglich die gepspeicherten Koordinaten vom ScaledField. Da dort die linke obere Ecke des Feldes in echten Koordinaten hinterlegt ist. Hinzukommt die Berechnung der Farbe zum

Einzeichnen. Der momentane Score wird durch den Max Score dividiert und als Ergebnis gibt es einen Wert zwischen 0 und 1. Dieser Wert kann dann als RGB für ein Color Objekt genutzt werden.
Die View hat zum glück keinen Einfluss auf den Rest des Programms. Denn beim Testen ist Sie gerne mal mit einer Exception abgestürzt, aber der Algorithmus hat trotzdem funktioniert (Oder wie beim Turnier finale, weswegen ich leider kein Bild meiner Darstellung vom Sieg zeigen kann).

2.5 Bots (Pinsel) (3)

Die andere Seite bei der Push-Architektur stellen die Pinsel selbst dar. Das ColorChange Event enthält Informationen darüber welcher Pinsel gerade an welcher Position ist. Um mit den Pinsel etwas anstellen zu können ist es wichtig zu wissen, wo sich diese befinden. In einen nur dafür bestimmten Controller werden die Informationen von ColorChange ausgewertet und in den entsprechenden Pinsel umgewandelt. Der Pinsel Controller wird von genau 3 PinselMover, den eigenen Pinseln, observiert. Jedes mal wenn sich ein Pinsel ändert werden alle Observer benachrichtigt.

```
brushes[colorChange.player][colorChange.bot].x = colorChange.x;
brushes[colorChange.player][colorChange.bot].y = colorChange.y;
subject.onNext(brushes[colorChange.player][colorChange.bot]);
```

2.6 Bot controling (6)

Jeder Pinsel hat seinen eigenen Controler und observiert die Events vom BrushController. Sobald ein Event gesendet wird, wird als erstes überprüft, ob es sich um den Pinsel handelt welcher vom Controller gesteuert wird. Als Nächstes wird überprüft, ob es noch einen Pfad von der Wegfindung gibt oder ob er schon abgearbeitet wurde. Solange ein Pfad existiert wird näherungsweise abgefragt ob sich der Pinsel in der Nähe des nächsten Wegpunkts befindet und somit der nächste Wegpunkt gesetzt werden kann. Außerdem wird der Richtungsvektor erneuert mit der momentanen Position.
Sollte einmal der Pfad leer sein und noch kein neuer Berechnet, bewegt sich der Pinsel zufällig in eine Richtung. Allerdings ist ein festes Intervall gesetzt in dem der Pfad neu berechnet wird.
In einem Intervall von 2 Sekunden wird der Pfad neu berechnet, den der Pinsel gehen soll. Es werden jedes mal die aktuellsten Spieldaten für die Berechnung benutzt.

2.7 Pathfinding (7)

Die Wegfindung benötigt zum Berechnen des Pfades ein Startfeld, Zielfeld und die Kanten auf denen der Pfad berechnet werden kann. Das Startfeld ist die momentane Position des Pinsels. Das Zielfeld muss errechnet werden. Hierbei sollte es sich möglichst um ein Feld handeln welches in einem Bereich liegt der möglichst viele Punkte bringt. Die Kanten sind die Felder des Spielfelds. Zu Beginn der Berechnung wird das momentane Spielfeld abgefragt und ein Clone erzeugt (Damit es keine Probleme beim Multithreading

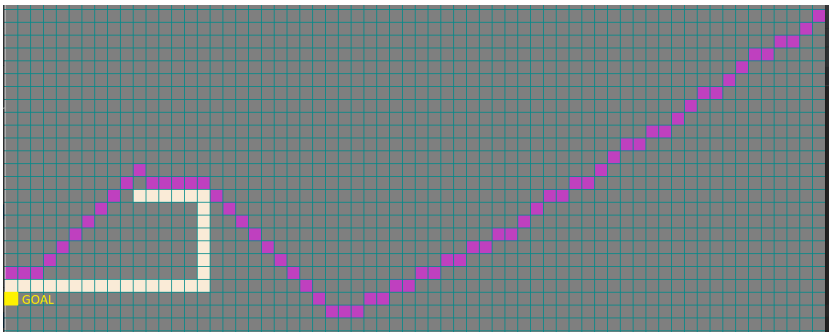
gibt).

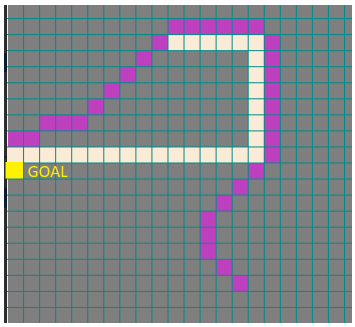
2.7.1 Heatmap

Zur Identifizierung des Zielfeldes benutze ich eine zwei schichtige Heatmap. Die erste Ebene fasst 4x4 skalierte Felder also 64 x 64 Pixel zusammen. Aus diesen Feldern werden dann wieder jeweils 2x2 zu 128x128 Pixel zusammen gefasst. Es wird jeweils der Durchschnittwert von allen Feldern in dem zusammengefassten Feld errechnet.
Der kleine Pinsel betrachtet aus der ersten Schicht das maximale Feld.
Für den großen Pinsel und die Spraydose wird aus der 128x128 Heatmap die Maximalfelder gewählt. Der große Pinsel erhält das Feld mit dem "höchsten" Score und die Spraydose den zweit "höchsten". "Höchste" da das gesamte Scoresystem darauf ausgelegt ist, dass die Felder mit dem geringsten Score, die Felder sind die am ehesten Punkte bringen. Also ist der Heatpoint das Feld mit dem geringsten Score.
Für den großen Pinsel und die Spraydose wird jeweils in der 64x64 Ebene nun der Heatpoint ausgewählt der in dem vorigen Feld der 128x128 Heatmap liegt. Das gleiche passiert dann nochmal für die skalierte Ebene 16x16.
Als Ergebnis wird ein Punkt auf dem skalierten Feld ermittelt welches das Zielfeld ist.

2.7.2 A*

Die Implementierung von A* mit Heuristik für das gegebene Umfeld ist relativ einfach. Die gesamte Vorarbeit diente dazu alles auf den A* auszurichten. Startfeld von der Position des momentanen Pinsels. Zielfeld von der Heatmap. Und das Spielfeld selber welches die Map der Kanten liefert die A* benutzen kann.
Die Implementierung ist dann aber doch nicht so ganz glatt gelaufen. Aber zum Glück mit Hilfe von Tests konnten sehr schnell die Fehler des Algorithmus ausgemerzt werden.





Chapter 3

Fazit

Die Implementierung der zweiten AI lief deutlich besser. Ich konnte einige Erfahrungen aus dem ersten Projekt übernehmen und anwenden. Mir war es von Anfang an wichtig die richtigen Elemente für meinen Algorithmus auszuwählen. Meine erste Entscheidung war es A* zu benutzen. Anhand von A* habe ich dann alle anderen Entscheidungen getroffen. Der Grund warum ich die Gegner als gleich betrachte, ist eher aus der Einfachheit heraus. Bei der ersten AI habe ich gemerkt, dass das Implementieren viel Zeit in Anspruch nimmt und somit entschieden einen simpleren einfacheren Algorithmus zu implementieren. Nach dem Turnier und sogar Diskutieren mit freunden über das Spiel gibt es wohl noch sehr viele Stellschrauben mit denen man den Algorithmus verbessern kann. Zusammenfassend hat mir das Schreiben beider AI's sehr viel Spaß gemacht und ich konnte viel lernen.

Und das Turnier habe ich gewonnen!

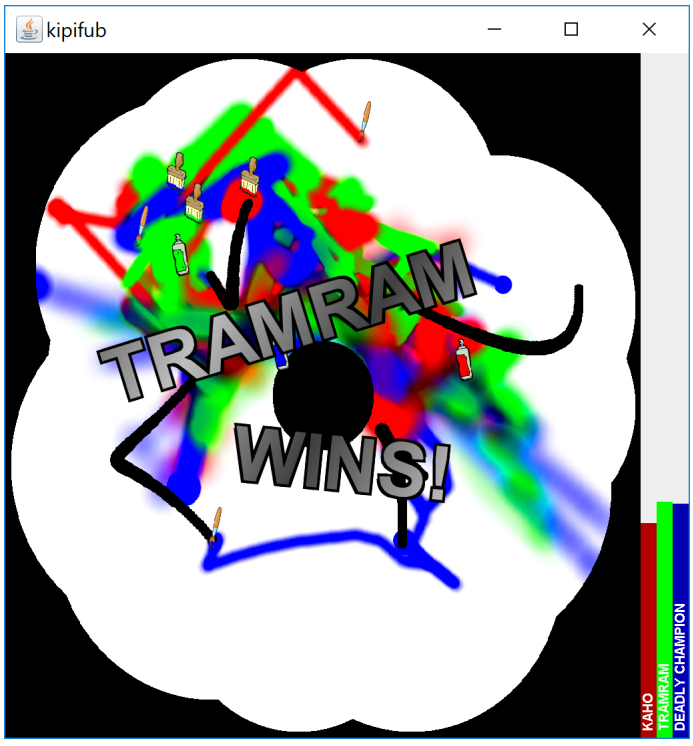


Figure 3.1: Darstellung der Pinsel