Raycasting und Levelgenerierung in `squayce`

– Peter Hofmann, Januar 2024 –

Inhaltsverzeichnis

1 Ziele **2**

2 Welt, Spieler, Kamera **2**

3 Raymarching **4**

3.1 Konstruktion des Strahls . . . 5
3.2 Verfolgung des Strahls bis zum Schnittpunkt . . . 5
3.3 Vermeidung des Fischaugeneffekts . . . 8
3.4 Vereinfachung . . . 9
3.5 Bestimmung der Linienhöhe . . . 11

4 Texturierung der Wände **11**

5 Einfärbung des Bodens **13**

5.1 Optimierung durch Fixkommazahlen . . . 15
5.2 Optimierung durch zusammengefasste Zeichenoperationen . . . 15

6 Levelgenerierung **16**

6.1 Erkennung invalider Runden . . . 19

1 Ziele

Das beschriebene Raycasting beruht auf dem Tutorial von Lode Vandevenne:

https://lodev.org/cgtutor/raycasting.html

Meinem Verständnis half es, den Sachverhalt noch einmal mit eigenen Worten aufzuschreiben. :-) Deshalb ist dieses Dokument auch auf Deutsch verfasst statt Englisch. Außerdem wurden für das vorliegende Spiel `squayce` nicht alle Einzelheiten direkt aus dem Tutorial übernommen. Letztlich ist dieses Dokument mehr als persönlicher Merkzettel für mich statt als Dokumentation für die Allgemeinheit gedacht.

Beschrieben werden soll also, wie Raycasting in `squayce` funktioniert. Auf Code-Beispiele wird daher verzichtet.

Ferner wird ein Blick auf die prozedurale Generierung der Rennstrecken geworfen.

2 Welt, Spieler, Kamera

Die Welt besteht aus einem Gitter. Die einzelnen Zellen werden mit Integern adressiert, die auf ihre linke untere Ecke zeigen. In Abbildung 1 sieht man beispielsweise das rot markierte Feld mit den Koordinaten $(2, 2)$, dessen rechte obere Ecke bei $(3, 3)$ läge. Das grüne Feld hat $(5, 3)$ und die rechte obere Ecke wäre bei $(6, 4)$. Gespeichert wird dieses Gitter im Array `WORLD`, daher die Adressierung mit Integern.

Die Welt ist immer *geschlossen*, das gesamte Spielareal muss also von Wänden umgeben sein. Außenbereiche gibt es nicht.

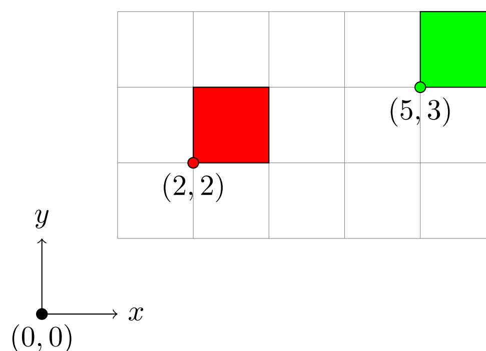

Abbildung 1: Die Welt als Gitter.

Der Spieler kann sich „frei“ in dieser Welt bewegen. Seine Position wird mit dem zweidimensionalen Vektor $\mathbf{p}$ bezeichnet. Implizit ist dem Spieler immer auch die Gitterkoordinate $\mathbf{g}$ zugeordnet, welche sich durch Abrunden ergibt:

$$\mathbf{p} = \begin{pmatrix} p_x \\ p_y \end{pmatrix}$$

$$\mathbf{g} = \begin{pmatrix} \lfloor p_x \rfloor \\ \lfloor p_y \rfloor \end{pmatrix}$$

Mittels der Integerwerte von $\mathbf{g}$ kann also der passende Eintrag im Array `WORLD` ermittelt werden.

Vektoren wie $\mathbf{p}$ werden im Folgenden in horizontaler Schreibweise notiert, also $\mathbf{p} = (p_x, p_y)$.

Abbildung 2 zeigt nun, dass sich der Spieler auch „innerhalb“ der Gitter bewegen kann, da seine Position mit Fließkommazahlen gespeichert ist.

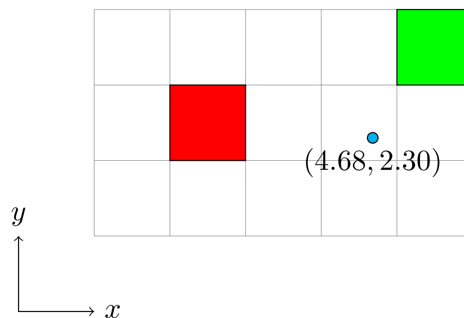

Abbildung 2: Der Spieler an der Position $\mathbf{p} = (4.68, 2.30)$, was der Gitterkoordinate $\mathbf{g} = (4, 2)$ entspricht.

Das Spielerobjekt besitzt ferner den stets normierten Vektor $\mathbf{v}$ (Komponenten v_x und v_y), der die derzeitige Blickrichtung darstellt. Zusammen mit dem Öffnungswinkel $\vartheta \in (0°, 180°)$ lässt sich die in Abbildung 3 gezeigte Kameraebene konstruieren. Wir werden nun Strahlen von der Spielerposition durch die Kameraebene schicken und testen, wo diese die Wände des Weltgitters treffen.

Dem Strahl ist im Programm nur die Richtung direkt zugeordnet – sein Ursprung ergibt sich implizit über die Position des Spielers.

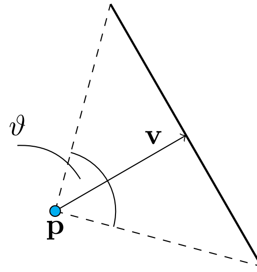

Abbildung 3: Kameraebene vor dem Spieler.

3 Raymarching

Abbildung 4 zeigt die Idee der Strahlenverfolgung: Strahlen beginnen an der Position des Spielers, durchqueren die Kameraebene und treffen dann letztlich auf ein Wandelement. Da die Welt geschlossen ist, wird immer ein solches Element getroffen. Relevant sind bei der Strahlenverfolgung alle Schnittpunkte mit den Gitterlinien.

Raycasting ist deshalb so schnell, weil diese Strahlenverfolgung nur in 2D stattfindet. Es wird also nur pro Spalte des Bildes ein Strahl ausgesendet und nicht pro Pixel. Ist die Distanz vom Spieler zum Schnittpunkt bekannt, dann kann eine vertikale Linie gezeichnet werden.

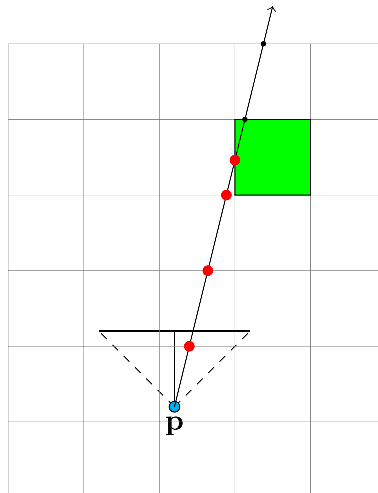

Abbildung 4: Ein Strahl wird vom Spieler aus durch die Kameraebene verschickt.

3.1 Konstruktion des Strahls

Im ersten Schritt konstruieren wir einen Strahl $\mathbf{r}$ für einen Pixel bei x bei gegebener Bildbreite w. Dazu ist es notwendig, auf der Kameraebene einen Vektor $\mathbf{r}_p$ zu definieren, mit dem ein Punkt auf dieser Ebene ausgewählt werden kann. Konkret ist:

$$\mathbf{r} = \mathbf{v} + \alpha \cdot \mathbf{r}_p \quad \text{mit} \quad \alpha \in [-1, 1]$$

Vergleiche hierzu Abbildung 5.

Die Richtung von $\mathbf{r}_p$ ist in 2D einfach zu bestimmen, da sie orthogonal zum Blickvektor $\mathbf{v}$ sein muss, also ergibt sich zunächst:

$$\mathbf{r}'_p = \begin{pmatrix} v_y \\ -v_x \end{pmatrix}$$

Die Länge von $\mathbf{r}_p$ hängt vom Öffnungswinkel ϑ ab:

$$|\mathbf{r}_p| = \tan \frac{\vartheta}{2}$$

Letztlich ist damit ($\mathbf{v}$ ist immer normiert):

$$\mathbf{r}_p = \tan \frac{\vartheta}{2} \cdot \begin{pmatrix} v_y \\ -v_x \end{pmatrix}$$

In Abbildung 6 sind einige Varianten gezeigt – je größer der Öffnungswinkel, desto „breiter“ die Kameraebene.

Entgegen des referenzierten Tutorials von Lode Vandevenne wird die Konstruktion über ϑ bevorzugt, da sie etwas größere numerische Stabilität bietet. Im Tutorial wird $\mathbf{r}_p$ häufig rotiert und kann sich damit im Laufe der Zeit gegenüber $\mathbf{v}$ „verziehen“.

Da nun $\mathbf{r}_p$ bekannt ist, ist nur noch α gesucht. Das Programm iteriert ganzzahlig über jeden Pixel $x \in [0, w-1]$ in horizontaler Richtung. Insgesamt werden also w Pixel besucht. Wir wollen dieses x auf das Intervall $[-1, 1]$ (Weltkoordinaten, Fließkomma) abbilden:

$$\alpha = 2 \cdot \frac{x}{w-1} - 1$$

3.2 Verfolgung des Strahls bis zum Schnittpunkt

Dem Verfahren liegt „Digital Differential Analysis“ (DDA) zugrunde, was es uns erlaubt, gezielt von Schnittpunkt zu Schnittpunkt zu springen.

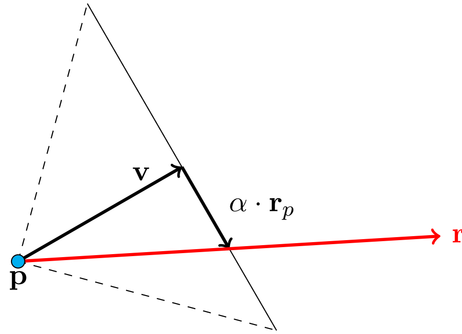

Abbildung 5: Konstruktion eines Strahls.

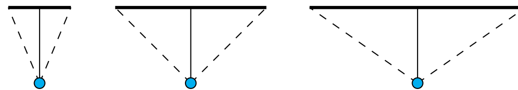

Abbildung 6: Kameraebenen von $\vartheta = 45°$, $\vartheta = 90°$ und $\vartheta = 110°$. Die Ebene befindet sich immer im selben Abstand zum Spielerpunkt.

Benötigt werden zunächst zwei Größen: d_x und d_y. Diese beschreiben, wie weit auf dem Strahl man laufen muss, um von einem Schnittpunkt zum jeweils nächsten Schnittpunkt in x- beziehungsweise y-Richtung zu gelangen. Abbildung 7 illustriert diese Größen. Man betrachte hier das blaue Dreieck oben rechts: Die untere, horizontale Seite hat die Länge 1, da von genau einer ganzzahligen Koordinaten zur nächsten gesprungen wird. Die rechte, vertikale Seite ergibt sich über das Verhältnis der beiden Komponenten des Strahls $\mathbf{r}$ (die Länge des Strahls ist also in diesem Moment irrelevant). Bildlich gesprochen: „Wieviele Schritte in y-Richtung für einen Schritt in x-Richtung?“ Mittels Pythagoras ergibt sich dann für die gesuchten Werte:

$$d_x = \sqrt{1 + \left(\frac{r_y}{r_x}\right)^2}$$

$$d_y = \sqrt{1 + \left(\frac{r_x}{r_y}\right)^2}$$

Darüberhinaus werden die Größen d_{x_0} und d_{y_0} benötigt: Sie geben an, wie weit auf dem Strahl *vom Startpunkt aus* gelaufen werden muss, um zum ersten Schnittpunkt in x-beziehungsweise y-Richtung zu gelangen. Zur Bestimmung kann man die Längen d_x und d_y skalieren, jenachdem, wie weit man sich innerhalb des Gitterfeldes befindet. Es muss

dabei unterschieden werden, in welche Richtung der Strahl zeigt:

$$d_{x_0} = (p_x - g_x) \cdot d_x \qquad \text{falls} \quad p_x > 0$$

$$d_{x_0} = (g_x + 1 - p_x) \cdot d_x \qquad \text{falls} \quad p_x < 0$$

und

$$d_{y_0} = (p_y - g_y) \cdot d_y \qquad \text{falls} \quad p_y > 0$$

$$d_{y_0} = (g_y + 1 - p_y) \cdot d_y \qquad \text{falls} \quad p_y < 0$$

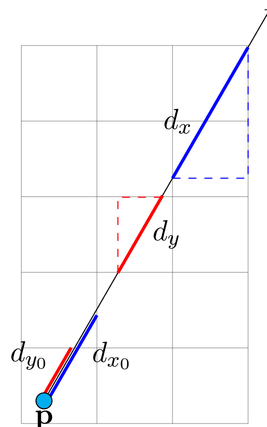

Abbildung 7: Bestimmung der Größen d_x und d_y sowie d_{x_0} und d_{y_0}.

Die Grundidee der Strahlenverfolgung ist nun in Worten:

- Beginne bei $\mathbf{p}$ und der damit verbundenen Gitterkoordinaten $\mathbf{g}$.
- Speichere die aktuell betrachtete Gitterkoordinate als $\mathbf{m} = (m_x, m_y)$ (initial also $m_x = g_x$ und $m_y = g_y$).
- Die Variable t_x wird diejenige Distanz entlang des Strahls enthalten, in der der nächste Schnittpunkt mit der x-Achse vermutet wird. t_y analog. Initial sind also $t_x = d_{x_0}$ und $t_y = d_{y_0}$.
- Wiederhole:
  - Ist das aktuelle $t_x < t_y$? Dann befindet sich in der Distanz t_x ein möglicher Schnittpunkt. (In t_y befindet sich auch einer, dieser wäre aber weiter von $\mathbf{p}$ entfernt und ist daher nicht interessant.) Erhöhe m_x um 1. Analog für t_y und m_y.

- Erhöhe t_x um d_x, da der wiederum nächste Schnittpunkt mit der x-Achse nur in dieser Distanz liegen kann. (Analog t_y.)

- Es wurde soeben ein neues Feld im Gitter betreten. Befindet sich hier eine Wand? Falls ja, dann wurde ein Schnittpunkt gefunden und der Algorithmus terminiert.

Nach Terminierung muss t_x noch einmal um d_x reduziert werden, um die Distanz zum tatsächlich Schnittpunkt (statt zum nächsten Kandidaten) zu erhalten. Die final errechnete Distanz sei d_f.

Diese Beschreibung hat angenommen, dass der Strahl in positive x- sowie y-Richtung läuft. Ist das nicht der Fall, dann müssen m_x oder m_y entsprechend verringert statt erhöht werden.

Man beachte, dass $\mathbf{m}$ „entkoppelt“ ist von $\mathbf{p} + k \cdot \mathbf{r}$, also dem Ablaufen des Strahls: Die $\mathbf{m}$-Werte können im Programm Integerwerte sein und es muss nicht von Weltkoordinaten (Fließkommawerte) auf Gitterkoordinaten (Integer) zurückgeschlossen werden. Das ist gut für Performance und Genauigkeit.

3.3 Vermeidung des Fischaugeneffekts

Berechnet wurde bis hierhin die Distanz von $\mathbf{p}$ zum Schnittpunkt. Für am äußeren Rand des Bildschirms liegende Pixel ist diese Distanz größer als für solche in der Bildschirmmitte – auch dann, wenn der Spieler genau gerade auf eine Wand blickt. Da die Höhe der gezeichneten Linie von dieser Distanz abhängt, ergeben sich also für äußere Pixel kürzere Linien. Das Resultat ist ein meist ungewünschter Fischaugeneffekt.

Wird stattdessen, wie in Abbildung 8 dargestellt, die Distanz d_p verwendet, so kann dieser Effekt vermieden werden: Diese Distanz ist so konstruiert, dass – bei geradem Blick auf eine Wand – alle Pixel dieselbe Distanz erhalten.

Die Abbildungen 9 und 10 vergleichen die Ergebnisse.

Was letztlich benötigt wird, ist eine Transformation der Objekte in das lokale Koordinatensystem der Kamera (so, wie es auch Abbildung 8 suggerieren soll). Das ist in unserem Fall aber stark vereinfacht, da wir nur die Distanz zum Schnittpunkt benötigen.

In Abbildung 8 sind zwei ähnliche Dreiecke gekennzeichnet:

- $\mathbf{p}, \mathbf{H}, \mathbf{A}$

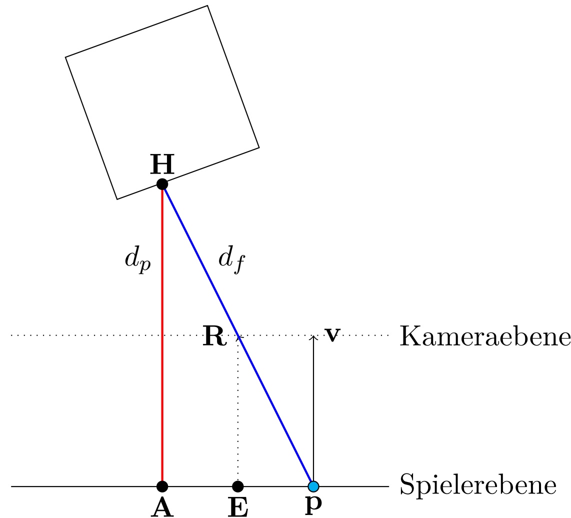

Abbildung 8: Distanz zum Schnittpunkt von Spielerposition (d_f) oder Spielerebene (d_p).

- $\mathbf{p}, \mathbf{R}, \mathbf{E}$

Das gesuchte d_p ist die Strecke von **A** nach **H**.

Die Strecke von **p** nach **R** ist bekannt und zwar die Länge des betrachteten Strahls **R** (da $\mathbf{R} = \mathbf{p} + \mathbf{r}$). Die Länge **R** nach **E** ist ebenfalls bekannt: Sie ist genau 1, da der Blickvektor **v** normiert ist. Über den ersten Strahlensatz lässt sich bestimmen, dass:

$$\frac{d_f}{|\mathbf{r}|} = \frac{d_p}{|\mathbf{v}|}$$

Also ist $d_p = \frac{d_f}{|\mathbf{r}|}$.

3.4 Vereinfachung

Im Tutorial von Lode Vandevenne wurde folgende Vereinfachung durchgeführt: Es wurde beobachtet, dass für den Raymarching-Algorithmus nur das *Verhältnis* von d_x zu d_y rele-

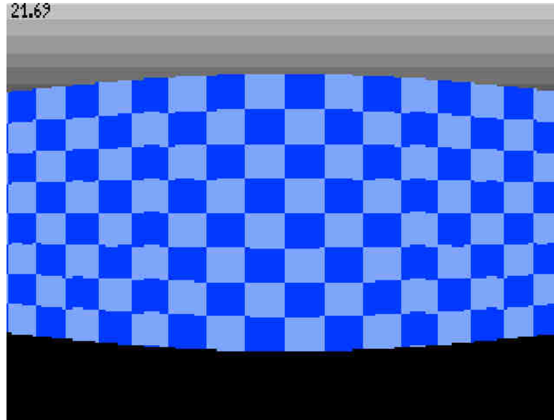

Abbildung 9: Raycasting-Ergebnis unter Verwendung von d_f.

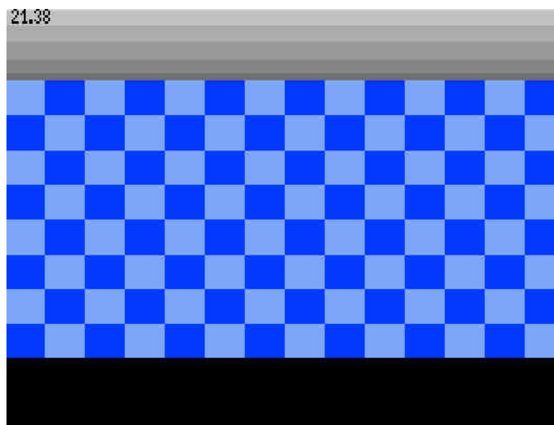

Abbildung 10: Raycasting-Ergebnis unter Verwendung von d_p.

vant ist. Außerdem lassen sich Umformungen durchführen (d_y analog):

$$\begin{aligned} d_x &= \sqrt{1 + \left(\frac{r_y}{r_x}\right)^2} \\ &= \frac{\sqrt{r_x^2 + r_y^2}}{r_x} \\ &= \frac{|\mathbf{r}|}{r_x} \end{aligned}$$

d_x enthält also den Faktor $|\mathbf{r}|$ und das ist genau der Faktor, den wir zur Vermeidung des Fischaugeneffekts am Ende entfernen. Man kann also d_x und d_y stattdessen so definie-

ren:

$$d_x = \left|\frac{1}{r_x}\right|, \quad d_y = \left|\frac{1}{r_y}\right|$$

Die vom Algorithmus berechnete Distanz ist dann automatisch d_p und die zusätzliche Division kann entfallen.

3.5 Bestimmung der Linienhöhe

Ist der Abstand zum Schnittpunkt erst einmal bekannt, kann die zu zeichnende Linienhöhe in y-Richtung festgelegt werden. Da letztlich eine Zentralprojektion durchgeführt werden soll, ist die Linienhöhe umgekehrt proportional zum Abstand des Schnittpunkts. Zusätzlich erfolgt noch eine Multiplikation mit der Anzahl der Pixel in y-Richtung, h, sodass sich ergibt:

$$l = \frac{h}{d_p}$$

Diese Linie wird dann an der aktuellen x-Position gezeichnet und zwar in y-Richtung zentriert. Startpunkt l_0 und Endpunkt l_1 sind dann inklusive Beschränkung der Werte auf gültige sichtbare Pixel:

$$l_0 = \max\left(\frac{h}{2} - \frac{l}{2}, 0\right)$$

$$l_1 = \min\left(\frac{h}{2} + \frac{l}{2}, h - 1\right)$$

4 Texturierung der Wände

Zur Texturierung einer Wand wird die genaue Koordinate des Schnittpunktes benötigt. Hierzu sei noch einmal auf Abbildung 8 verwiesen: Wir sehen, dass wir vom Spielerpunkt $\mathbf{p}$ aus zu $\mathbf{H}$ gelangen können, indem wir $\mathbf{r}$ folgen. Wir müssen nur wissen, um welchen Faktor $\mathbf{r}$ skaliert werden muss, also „wie weit“ wir in dieser Richtung laufen müssen.

Die intuitivste Variante wäre, den Vektor $\mathbf{r}$ zu normieren und dann mit dem Faktor d_f zu skalieren – diese Idee lässt sich direkt aus der Skizze ablesen. Das heißt, wir würden

bilden:

$$\mathbf{H} = \mathbf{p} + \frac{1}{|\mathbf{r}|} \cdot d_f \cdot \mathbf{r}$$

Nun wissen wir aber bereits, dass $d_p = \frac{d_f}{|\mathbf{r}|}$, also können wir schreiben:

$$\mathbf{H} = \mathbf{p} + d_p \cdot \mathbf{r}$$

Aus **H** kann dann die x- oder y-Komponente ausgelesen werden, jenachdem, welche Seite des Gitterfeldes getroffen wurde.

Anstatt eine einfarbige Linie der Höhe l zu zeichnen, werden diese Pixel mit den Farbwerten der Textur gefüllt. Das heißt, für den obersten Pixel der Linie auf dem Bildschirm benutzen wir den obersten Pixel in der Textur, für den zweiten Bildschirmpixel den zweiten Texturpixel und so weiter. t_u bezeichnet denjenigen Texturpixel, der gerade gezeichnet wird. Allerdings ist in den seltensten Fällen die zu zeichnende Linie genauso hoch wie die Textur, also muss die „Schrittweite“ s beim Abtasten der Textur angepasst werden – seien hierzu wieder h die Zahl der Pixel in y-Richtung auf dem Bildschirm und t_h die Höhe der Textur:

$$\frac{1}{h} = \frac{s}{t_h} \iff s = \frac{t_h}{h}$$

Es kann nun vorkommen, dass $l > h$, also dass der Spieler so nahe an der Wand steht, dass die zu zeichnende Linie die verfügbaren Pixel in y-Richtung übersteigen würde. In diesem Fall dürfen wir beim Abtasten der Textur nicht an der Texturkoordinate $t_u = 0$ anfangen. Die anschaulichste Erklärung: Wir nehmen an, die Linie würde tatsächlich über den Bildschirmrand hinausragen und wir machen diese imaginären Schritte beim Abtasten der Textur wirklich. Sobald wir dann beim ersten sichtbaren Pixel ankommen, sind also schon k Schritte vergangen. Dieses k ist genau die Zahl der überschüssigen Pixel. Wir könnten also das Abtasten der Textur bei $k \cdot s$ starten lassen.

Die naive Implementierung sähe so aus ($l_{\text{maybe negative}}$ ist wie l_0, darf aber negativ werden):

$$\begin{aligned} t_u &= s \cdot \max\left(-l_{\text{maybe negative}}, 0\right) \\ &= s \cdot \max\left(-\left(-\frac{l}{2} + \frac{h}{2}\right), 0\right) \\ &= s \cdot \max\left(\frac{l}{2} - \frac{h}{2}, 0\right) \end{aligned}$$

Die Bestimmung des Maximums erfordert jedoch eine weitere `if`-Verzweigung, was der Performance grundsätzlich nicht zuträglich ist. Im Tutorial von Lode Vandevenne wurde daher dieser Trick benutzt:

$$t_u = s \cdot (l_0 - l_{\text{maybe negative}})$$

Für l_0 wurde diese Entscheidung nämlich bereits getroffen. Für Linien, die komplett auf den Bildschirm passen, wird t_u dann bei 0 starten. Bei zu hohen Linien entsteht der gewünschte Offset.

Texturierung ist leider in mehrlei Hinsicht sehr suboptimal:

- Wir müssten bei naiver Vorgehensweise die Textur in y-Richtung iterieren, was der Cache-Lokalität zuwiderläuft.

- Dasselbe Problem haben wir beim Schreiben in den Grafikpuffer ebenso.

- Es gibt kein direktes Integer-Mapping der y-Koordinaten der Textur auf die y-Koordinaten der Linie. Bei naiver Vorgehensweise müssten hier für jeden Pixel Fließkommaberechnungen durchgeführt werden.

Der erste Punkt lässt sich leicht verhindern, indem die Texturen mit vertauschter x- und y-Koordinate im Speicher abgelegt werden. Dieser Trick klappt beim Schreiben in den Grafikpuffer leider nicht – hier ist noch keine elegante Lösung bekannt.

Zur Vermeidung von Fließkommaberechnungen setzt `squayce` an dieser Stelle auf Fixkommazahlen. In den Tests wurde die Framerate dadurch nahezu verdoppelt.

5 Einfärbung des Bodens

Aus Gründen der Performance benutzt `squayce` keine Texturen für den Boden sondern nur einfache Farben. Hier muss noch weitere Optimierung stattfinden, um Texturierung zu ermöglichen.

Wir werden nun jede Pixelreihe in y-Richtung betrachten, beginnend ab der Bildschirmmitte. Die jeweils betrachtete Reihe wird mit y benannt. Ein Strahl durch die Kameraebene hatte bisher nur eine x- und eine y-Komponente. Zur Darstellung des Bodens (oder theoretisch des Himmels) wird nun eine z-Komponente benötigt. Die Kameraebene befindet sich weiterhin im Abstand 1 vor dem Spieler und der Spieler selbst im Abstand 0.5 über dem Boden. Über einen Dreisatz gelangen wir zu diesem r_z:

$$\frac{r_z}{0.5} = \frac{y_{\text{rel}}}{\frac{h}{2}}$$

Dabei ist y_{rel} die Zahl der Pixel seit der Bildschirmmitte. Also genauer:

$$\frac{r_z}{0.5} = \frac{y - \frac{h}{2}}{\frac{h}{2}}$$

$$\iff \quad \frac{r_z}{0.5} = \frac{2}{h} \cdot (y - \frac{h}{2})$$

$$\iff \quad r_z = \frac{1}{h} \cdot (y - \frac{h}{2})$$

Für die Einfärbung wird dann – ähnlich wie beim Zeichnen der Wände – der Abstand von der Kameraebene zum Schnittpunkt eines Strahls mit dem Boden benötigt. In Abbildung 11 sind abermals zwei Dreiecke zu sehen, anhand derer sich ergibt:

$$\frac{1}{r_z} = \frac{d_p}{0.5}$$

$$\iff \quad d_p = \frac{1}{2 \cdot r_z}$$

Also haben wir insgesamt:

$$d_p = \frac{h}{2} \cdot \frac{1}{y - \frac{h}{2}}$$

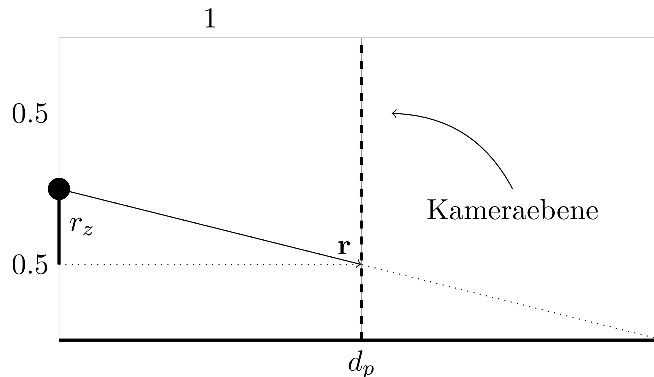

Abbildung 11: Bestimmung von d_p über zwei ähnliche Dreiecke.

Die Idee im Tutorial von Lode Vandevenne ist nun, zwei Strahlen pro y-Zeile zu betrachtet: den ganz linken und den ganz rechten. Zwischen diesen beiden Treffpunkten können wir linear interpolieren, da der Boden parallel zur xy-Ebene liegt. Wieder ein Dreisatz: Ein Pixel in x-Richtung verhält sich zur Bildbreite w wie ein Schritt in Weltkoordinaten (das gesuchte s) zur zurückzulegenden Distanz. Diese Schrittweite ist für x-

und y-Richtung unterschiedlich:

$$\frac{1}{w} = \frac{s_x}{d_p \cdot r_{\text{rechts},x} - r_{\text{links},x}}$$

$$\text{und} \quad \frac{1}{w} = \frac{s_y}{d_p \cdot r_{\text{rechts},y} - r_{\text{links},y}}$$

Diesen Formeln liegt wie schon bei der Texturierung der Wände zugrunde, dass wir über $d_p \cdot \mathbf{r}$ vom Spieler zum Schnittpunkt gelangen können – hier ist für den Moment aber nur die Distanz vom Spieler dorthin relevant, nicht der absolute Punkt.

Sind die beiden Schrittweiten s_x und s_y bekannt, so kann pro y-Zeile über alle x iteriert und in jedem Schritt der betrachtete Punkt auf dem Boden um s_x und s_y verschoben werden. Wir beginnen dabei beim Schnittpunkt mit dem linken Strahl:

$$f = \text{Schnitt}_{\text{Links}} = \begin{pmatrix} \mathbf{p}_x + d_p \cdot r_x \\ \mathbf{p}_y + d_p \cdot r_y \end{pmatrix}$$

Und in jeder Iteration findet dann statt:

$$f' = f + \begin{pmatrix} s_x \\ s_y \end{pmatrix}$$

Jedem dieser f-Werte entspricht eine Gitterkoordinate im Welt-Array. Dort kann nachgeschlagen werden, mit welcher Farbe der Boden zu zeichnen ist.

5.1 Optimierung durch Fixkommazahlen

Diese innere Schleife, die die x-Pixelkoordinaten abtastet, ist ein „hot path". Hier zählt jede Optimierung. Eine davon ist, nicht mit Fließkommazahlen zu rechnen. Statt `double` wird `long int` verwendet, was unter 16-Bit-DOS 32 Bit lang ist. Die oberen 16 Bit sollen der ganzzahlige Teil vor dem Komma sein, die unteren der nach dem Komma. Das kann erreicht werden durch Multiplikation der `double`-Werte mit 65536 (`0xFFFF`) und danach erfolgt die Konversion zu `long int`.

5.2 Optimierung durch zusammengefasste Zeichenoperationen

Zwei Vorgänge sind besonders teuer: das Nachschlagen im Welt-Array und das Zeichnen in den VGA-Puffer.[1] Um beides seltener durchzuführen, wird in jedem Schleifendurchlauf

[1] Würde Texturierung benutzt, wäre das Nachschlagen in den Texturdaten ein weiterer teurer Vorgang. Dafür ist derzeit keine Rechenzeit übrig, weshalb auf Texturierung des Bodens verzichtet wird.

in x-Richtung geprüft, ob sich die Gitterkoordinate tatsächlich geändert hat. Nur, wenn das der Fall ist, wird gezeichnet.

Derzeit ist diese Zusammenfassung durch `if`-Tests implementiert. Ein weiterer Optimierungsansatz wäre hier, wieder den DDA-Algorithmus zu bemühen.

6 Levelgenerierung

Im großartigen Spiel *Worms* von 1995 gab es die Möglichkeit, Levels durch Eingabe einer Zahl auszuwählen. Der Witz daran war aber das, was auf der Box des Spiels stand: „Over 4 billion levels!“ Es konnte sich also nicht um vorgefertigte Levels handeln. Stattdessen sind sie alle prozedural generiert – und die Benutzereingabe bestimmt den *Seed* eines Zufallszahlengenerators.

`squayce` benutzt dasselbe Prinzip.

Begonnen wird mit einer rechteckigen Strecke – `@` kennzeichnet die Straße, Punkte sind Wände:

```
...........
.@@@@@@@@@.
.@.......@.
.@.......@.
.@.......@.
.@.......@.
.@.......@.
.@.......@.
.@.......@.
.@@@@@@@@@.
...........
```

Danach werden iterativ Muster gesucht und ersetzt. Stand jetzt gibt es effektiv nur eine Ersetzungsregel: Wird ein gerades Stück der Länge 3 gefunden, dann wird dies durch eine Schikane ersetzt. Wird also folgendes Muster gesehen:

```
.@
.@
.@
```

Dann wird es mit einer gewissen Wahrscheinlichkeit – und hier kommt der Zufallsgenerator ins Spiel – ersetzt durch:

```
@@
@.
@@
```

Diese Ersetzungsregel existiert für alle Richtungen, nicht nur vertikal.

Sorge muss dafür getragen werden, keine neuen zusammenhängenden Stücke zu erzeugen, sonst entsteht unter Umständen ein Labyrinth und es ist für den Spieler nicht mehr ersichtlich, welchem Weg zu folgen ist.[2] Wird die Strecke also wie oben nach links erweitert, dann dürfen sich in den angrenzenden Bereichen nicht bereits andere Streckenstücke befinden. Erreicht wird das durch Erweiterung der obigen Regel auf Folgendes:

```
..?
..@
..@
..@
..?
```

Es werden also auch zusätzliche umliegende Nachbarn geprüft. Was sich an den mit Fragezeichen markierten Stellen befindet, spielt keine Rolle.

Nach wenigen Iterationen entstehen bereits stark unterschiedliche Formen. Abbildung 12 zeigt einen beispielhaften Ablauf.

Danach wird ein Streckenabschnitt der Form `.@.` gesucht. Nach Konstruktion ist das ein genau vertikaler Streckenabschnitt, der auch existieren muss. Dieser Bereich wird zur Startposition erklärt.

Bis zu diesem Punkt entsteht eine „menschenlesbare“ Rennstrecke, die nun für den User nach `WORLD.TXT` exportiert wird. Dort kann sie mit einem Texteditor begutachtet und auch manuell bearbeitet werden. Auf Wunsch kann beim nächsten Start diese Datei dann wieder eingelesen werden – der Algorithmus fährt dann mit den folgenden Schritten fort.

Anschließend wird die so entstandene Weltkarte „aufgeblasen“. Jedes Feld der Karte wird durch ein 5×5 Felder großes Quadrat ersetzt. Durch den so entstandenen Platz können diejenigen Felder direkt neben den Wänden als „slow zone“ markiert werden: Hier wird der Spieler größerer Reibung ausgesetzt und kommt kaum mehr vorwärts, vergleichbar mit einem Kiesbett.

Der letzte Schritt ist die Einteilung der Weltkarte in Zonen. Zunächst wird die tatsächliche Bounding Box der Strecke bestimmt und dann die vier Quadranten unterschiedlich

[2] Diese Variante wurde für eine gewisse Zeit im Spiel benutzt, hat sich aber als zu verwirrend herausgestellt.

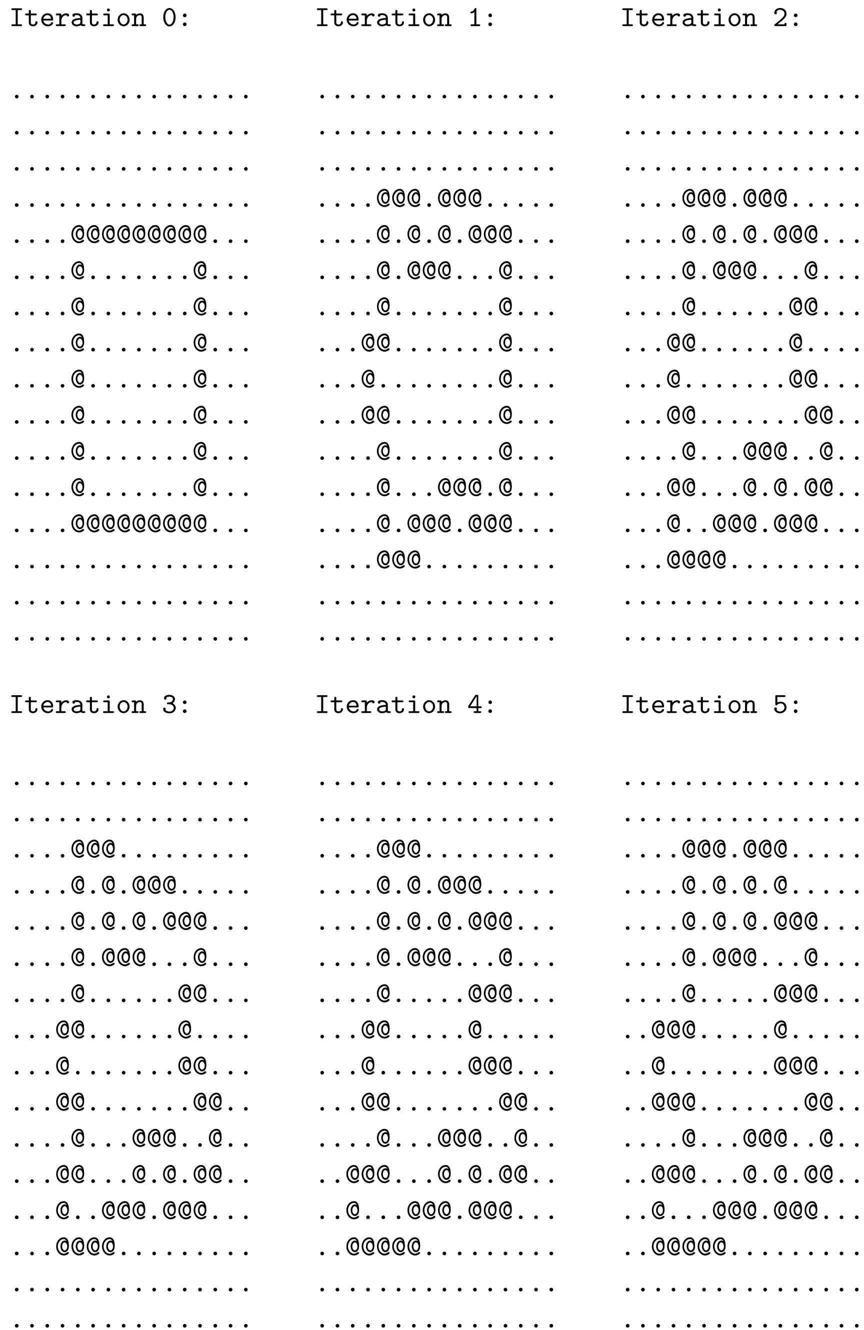

Abbildung 12: Verzerrung der Strecke zur Levelgenerierung.

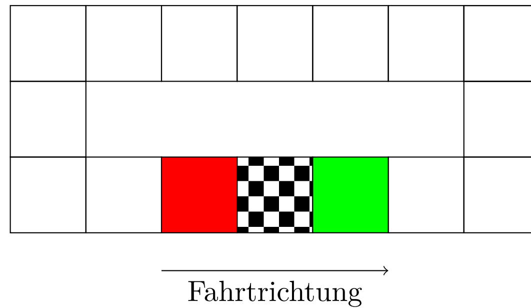

Abbildung 13: Schematische Darstellung einer Strecke. Zu sehen sind nur solche Felder der Weltkarte, die vom Spieler befahrbar sind (also keine Wände). Der Spieler startet an der Ziellinie, kann dann über grün fahren, dann eine Runde entgegen des Uhrzeigersinns drehen und schließlich über rot zurück zur Ziellinie. Dabei ist die Fahrtrichtung nur korrekt, wenn erst grün, dann rot und dann die Ziellinie selbst überquert wird.

eingefärbt, damit die Wände jeweils anders texturiert werden.

6.1 Erkennung invalider Runden

Spieler können umdrehen und rückwärts fahren. Solche Runden sollen als nicht zulässig erkannt werden.

Hierzu wird in Fahrtrichtung vor und nach der Ziellinie jeweils eine Zeile gesondert markiert. Der Spieler muss diese drei Zonen in fester Reihenfolge überqueren: Zuerst das Feld nach der Ziellinie, dann dasjenige vor der Ziellinie und dann die Ziellinie selbst. Diese Abfolge kann nur durch das Fahren einer vollständigen Runde erreicht werden, wie Abbildung 13 illustrieren soll.

Das Spielerobjekt erhält zwei zusätzliche Flags, für jede zusätzliche Zeile eines. Beim Überqueren wird das Flag gesetzt. Wird aber die Ziellinie überquert, so werden beide Flags zurückgesetzt.