Funktionsweise eines Raycasters mithilfe eines selber geschriebenen Beispiels erklärt

Andreas Gwilt

15. Februar 2015

Inhaltsverzeichnis

Zusammenfassung

This looks rather concrete to me. Actually, no it doesn't.

Kapitel 1

Was ist ein Raycaster?

Raycasting ist ein ziemlich weiter Begriff, der meistens für eine Rendermethode benutzt wird. Allgemein ist es eine Technik, um zu sehen, ob ein Strahl eine Fläche schneidet. Man "wirft" einen Strahl, und überprüft, ob er eine Ebene schneidet oder nicht. Diese ist dem Raytracing sehr ähnlich, aber ist viel limitierter und auch schneller. Ich habe, um den Raycaster zu erklären, eine in der ersten Person gerenderten Version von Conways *Spiel des Lebens* geschrieben.

Abbildung 1.1: Wolfenstein 3D, ein Spiel, das einen Raycaster verwendete.

1.1 Welche Probleme löst ein Raycaster?

Raycasting ist eine Methode, um zu überprüfen, ob ein Strahl eine Fläche schneidet oder nicht. Die häufigste Anwendung dafür ist als einfaches Renderverfahren, um

ein Spielfeld in Pseudo-3D darzustellen. Es gibt aber auch mehrere andere Anwendungen, z.B. Kollisionserkennung oder um fest zu stellen, ob etwas sichtbar ist oder nicht. In meinem Beispiel nutze ich die Methode (in der `cast()` Funktion) hauptsächlich dazu, das Spielfeld darzustellen, aber auch als Kollisionserkennung für die `walk()` Funktion. Ich werde mich jedoch auf den Raycaster als Rendermethode konzentrieren. Raycasting wird auch meistens in Spielen benutzt, wo schnelles Rendern wichtig ist, wodurch es als realistische Rendermethode nicht geeignet ist.
In den frühen 1990ern, als Raycasting beliebt wurde, hatte man noch ziemlich wenig Rechenleistung, wollte aber "3D" Spiele schreiben. Eine wirklich dreidimensionales Spiel-Engine (wie die 1996 erschienene Quake Engine) war noch nicht schnell genug, um in einem Spiel benutzt zu werden, also verwendete man Methoden wie Raycasting, um die Illusion von 3D herzustellen. Das vielleicht berühmteste Spiel, was Raycasting benutzte, war wahrscheinlich Wolfenstein 3D (Auch der erste beliebte Ego-Shooter). Wolfenstein 3D, auch Wolf3D genannt, hatte ein zweidimensionales Spielfeld, was in 3D dargestellt wurde.
Raycasting musste jedoch sehr viele Kompromisse eingehen, um so schnell zu sein. Das Spielfeld war ein zweidimensionales Array, also konnte es nur Rechte Winkel geben, und Decke und Boden mussten immer gleich hoch bleiben. Man erkennt an dem Screenshot von Wolf3D deutlich, dass das Spiel nicht wirklich dreidimensional ist. Auch sieht das Bild blockhaft aus, und die Beleuchtung ist nicht realistisch.

1.2 Die Geschichte des Raycasters

Raycasting wurde schon diskutiert, lange bevor PCs leistungsstark genug waren, um es wirklich in Spielen zu benutzen. Eine 1982 veröffentlichte Abhandlung von Scott Roth beschreibt Raycasting als Methode, dreidimensionale Körper zu rendern.

Eines der ersten Spiele, die das Prinzip implementierten, war *Hovertank 3D*, was in April 1991 von id Software in einem Magazin veröffentlicht wurde. *Hovertank 3D* war noch sehr primitiv und hatte noch keine Texturen auf den Wänden, noch dazu wurde es nicht für eine große Zielgruppe veröffentlicht, aber es war der Anfang eines neuen Genre von Spielen. Im darauffolgenden November erschien *Catacomb 3-D*,

was Texturen hatte, und oft als der erste Ego-Shooter gesehen wird.

Zufrieden mit den zwei “Prototypen” entwickelten id Software ihr neues Spiel *Wolfenstein 3D*, das die Technologie von *Catacomb 3-D* übernahm. *Wolfenstein 3D*, was 1992 von Apogee Software veröffentlicht wurde, war ein großer Erfolg und machte Ego-Shooter, und damit den Raycaster, bekannt. Danach schaffte id Software das revolutionäre Spiel *DOOM*. *DOOM* war ein noch viel größerer Erfolg, der eine sehr verbesserten Spiel-Engine hatte. Die neue Engine war jedoch nicht mehr wirklich ein echter Raycaster, sondern benutzte Binary Space Partitioning und andere fortgeschrittene Methoden. Heutzutage findet der Raycaster nicht mehr viel Verwendung als Renderer, aber das Prinzip wird mit dem Raytracer fortgeführt. Der Raytracer arbeitet immer in drei Dimensionen, und ist rekursiv, d.h. er simuliert den kompletten Gang eines Lichtstrahles, inklusiv Reflexionen. Raytracing ist aber noch immer zu langsam für Echtzeit Darstellung, und findet eher als Renderer von stillen Bildern Gebrauch.

Kapitel 2

Wie funktioniert ein Raycaster?

2.1 Allgemein

2.1.1 Daten

Die Informationen (z.B. das Spielfeld oder die Spielerposition) müssen auf bestimmte Art gespeichert werden, um von dem Raycaster benutzt zu werden. In meiner Implementierung habe ich eine Funktion, die einen Strahl in einem bestimmten Winkel von einem bestimmten Punkt wirft: `cast(world, p_x, p_y, a)`. Die Welt wird als zweidimensionales Array gespeichert, in dessen Feldern 1 für eine Wand oder Zelle steht, und 0 für leeren Raum. Man könnte auch Farben speichern, oder mehrere Zahlen für mehrere Texturen/Farben haben. Meistens würde man nur Wände speichern, aber ich habe als Spiel Conways *Spiel des Lebens* genommen, ein zweidimensionaler Zelluläre Automat, in dem das Spielfeld in dem eben beschriebenen Format gespeichert wird. Man kann aber zwischen einem Kartesischen Koordinatensystem, in dem (0,0) unten links ist, und einem Array Kartesischen Koordinatensystem, in dem (0,0) oben links ist, wählen. Ich habe mich für (0,0) unten links entschieden, da es (meiner Ansicht nach) leichter verständlich ist.
Dann muss man die Spielerposition speichern (die Variablen `p_x` und `p_y`). Diese Koordinaten sind aber in einem anderen Format als das Spielfeld. Man will nämlich den Spieler innerhalb eines Feldes bewegen können. Also hat man eine Konstante (`TILE`), die die Länge eines Feldes beschreibt. Um zu bestimmen, in welchem Feld der Spieler steht, rechnet man einfach `world[p_x // TILE][p_y // TILE]`: man

teilt ohne Rest die Spielerkoordinaten durch die Feldlänge `TILE`.
Der Winkel des Spielers oder des Strahls ist auch wichtig. Da die Python-Funktionen für sin, cos, etc. das Bogenmaß annehmen, habe ich alle Winkel so gespeichert. Die Funktionen gehen auch davon aus, dass ein Winkel immer positiv und unter 2π bleiben wird. Die Richtung"der Winkel muss man auch selber definieren, aber ich habe 0 als rechts definiert, und $\pi/2$ als unten (auf die x-Achse zeigend).

2.1.2 Was ist die Aufgabe des Raycasters?

Here goes stuff explaining what things the RC needs to do in a program, as opposed to things like adding sprites, etc.

2.2 Schritt für Schritt Erklärung

2.2.1 Initialisierung

Zuerst müssen einige Variablen gesetzt werden. Einige werden in dem ganzen Spiel konstant bleiben:

```
TILE = 32 # Länge der Ziegel von world
pi = math.pi

plane_x = 1280 # Bildschirm/Fenster Auflösung
plane_y = 720
fov = math.radians(60) # Sichtfeld
step = 20 # Länge eines Schrittes (für walk())
turn = math.radians(5) # Wie weit der Spieler sich dreht
```

Jetzt kommen einige Variablen für das Spielfeld und den Spieler, die zum Teil geändert werden können (z.B. Spielerposition):

```
hl = 100 # Horizontale/vertikale länge des Spielfeldes
vl = 50
world = [[0 for i in range(vl)] for j in range(hl)] # Welt array
p_x = 32 # Spieler x,y
p_y = 32
p_a = 0 # Spieler Winkel (zeigt am Moment nach "rechts"
p_height = TILE / 2 # Spieler höhe

plane_d = (plane_x / 2) / math.tan(fov/2) # distance to plane
ray_angle = fov / plane_x # angle between rays
```

2.2.2 Die Zentrale Schleife

Danach wird `pygame` (Die Graphik-Library) initialisiert, und die zentrale Schleife angefangen. Spiel-schleifen haben in der Regel drei schritte:

1. Befehle von dem Benutzer holen
2. Mit dieser information den Spielzustand erneuern
3. Das Spiel rendern

In meinem Beispiel ist Schritt 1. Nachschauen, welche Tasten gedrückt werden. Entsprechend ändert sich `p_a`, wenn der Spieler sich drehen will, und `p_x`/`p_y`, wenn der Spieler sich bewegt. Dazu wird die `walk(world, p_x, p_y, a)` Funktion abgerufen, die mithilfe der `cast` Funktion prüft, ob der Weg frei ist oder nicht. Danach wird das Spielfeld nach den *Life*-Regeln von der `update(world)` Funktion aktualisiert.
Hauptsächlich interessieren wir uns aber für die `draw(world)` Funktion, die das ganze mithilfe des Raycasters rendert. Bevor das eigentliche Rendern anfängt, wird der Hintergrund gemalt. In diesem fall heißt das einfach, das Fenster weiß zu füllen, und für den Boden ein hellgraues Rechteck zu malen:

```
screen.fill((255,255,255))
pygame.draw.rect(screen, (200,200,200), \
((0,(plane_y/2)),(plane_x,plane_y)))
```

Wie beschrieben wirft man jetzt für jede Spalte des Fensters einen Strahl mit `cast`, und malt mit der Information einen Vertikalen Strich in der Spalte. Dazu muss man zuerst den Winkel des ersten Strahls finden. Dazu subtrahiert man die Hälfte des Blickwinkels von dem Spieler-Winkel `p_a`. Dabei muss man mod 2π bzw. $360°$ rechnen, damit man immer einen Positiven Winkel hat.

```
angle = (p_a - (fov/2)) % (2*pi)
```

Mit dem Startwinkel kann man jetzt mit einer Schleife durch die Spalten iterieren, und mit der `cast` Funktion die Distanz zu der nächsten Wand bestimmen. Nachdem das Fertig ist, wird das Fenster aktualisiert.

```
for col in range(plane_x):
        dist = cast(world, p_x, p_y, angle)
```

```
            if dist > 0:
                    pygame.draw.line(screen, (0,0,0), \
                    (col,((plane_y/2) - dist_to_offset(dist))), \
                    (col, (plane_y/2) + dist_to_offset(dist)))
            angle = (angle + ray_angle) % (2*pi)
    pygame.display.flip()
```

Wenn diese Distanz nicht -1 (keine Kollision) ist, wird eine Linie auf dem Fenster gemalt. Um herauszufinden, wie lang die Linie sein sollte, dient die `dist_to_offset` Funktion, die eine Distanz annimmt, und den Abstand des Anfangs der Linie zu der Mitte der Spalte zurückgibt. Die wichtigste Funktion ist aber `cast`: die Funktion, die die Distanz zur nächsten Wand zurückgibt.

2.2.3 Die `cast` Funktion

2.2.3.1 Struktur

Die `cast` Funktion besteht daraus, zuerst nach Schnittpunkten mit den vertikalen Wänden, und dann mit den horizontalen zu prüfen, und dann die kürzeste Distanz zurückzugeben. Zuerst werden `vvalid` und `hvalid` auf False gesetzt; wir gehen davon aus, dass wir keinen Wert haben. Sobald wir einen vertikalen oder horizontalen Schnittpunkt finden, werden `vvalid` bzw. `hvalid` auf True gesetzt.
Die Reihenfolge vertikal/horizontal ist egal, aber ich habe mit vertikal angefangen.

2.2.3.2 Vertikale Schnittpunkte

Zuerst müssen wir festlegen, ob es eigentlich Sinn macht, den Strahl zu werfen. Wenn der Winkel `a` vertikal ist, überspringen wir die Vertikalen Schnittpunkte:

```
if not (a == 0.5*pi or a == 1.5*pi):
```

Der erste Punkt: Um herauszufinden, wo der Strahl die Vertikalen Grenzen überschreitet, müssen wir einige Werte finden, unter anderem den Ersten Punkt. Betrachten wir zuerst `v_x`. Wenn der Strahl nach rechts zeigt (`(a < 0.5 * pi) or (a > 1.5 * pi)`), ist `v_x` der erste Punkt in der neuen Zelle ($TILE * \langle Spieler\_Zelle \rangle + TILE$). Sonst ist `v_x` der letztePunkt in der Zelle vor der, in der der Spieler steht, also $TILE * \langle Spieler\_Zelle \rangle - 1$. Um die Zelle des Spielers zu finden, teilt man ohne rest durch TILE; in python `p_x // TILE`. Somit haben wir diesen code:

```
if (a < 0.5 * pi) or (a > 1.5 * pi):
        v_x = (p_x // TILE)*TILE + TILE
else:
        v_x = (p_x // TILE)*TILE - 1
```

Die Fälle $a = 0.5\pi$ und $a = 1.5\pi$ haben wir schon ausgeschlossen. Man könnte natürlich auch `((p_x // TILE) + 1) * TILE` rechnen, aber ich glaube es macht keinen Unterschied.

Schwieriger wird es, `v_y` auszurechnen. Dazu werden wir $v\_y = p\_y + \Delta y$ rechnen, indem wir Δy mit Tangens berechnen (Siehe Abb. 2.1). Da wir hier ein rechtwinkliges Dreieck haben, ist $\tan a = \frac{-\Delta y}{\Delta x}$, wobei zu beachten ist, dass Δy negativ ist, weil der Winkel "nach unten" Zeigt. Erst bei einem negativen winkel (bzw. $a > \pi$) ist Δy positiv.

Δx ist ziemlich leicht zu berechnen: einfach $v\_x - p\_x$. Somit haben wir

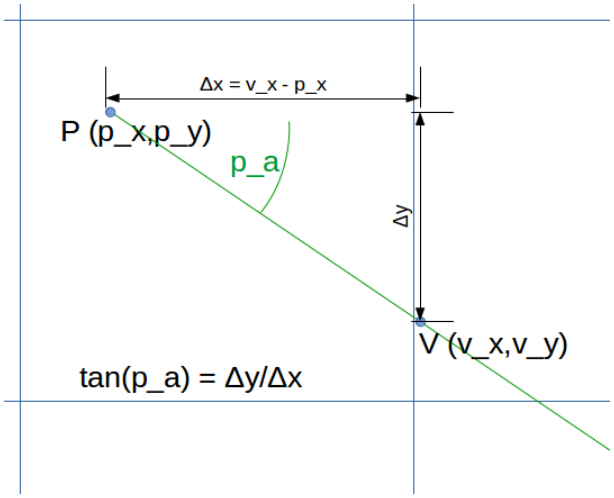

Abbildung 2.1: Punkt V finden

$$\tan a = \frac{-\Delta y}{v\_x - p\_x}$$

oder

$$\Delta y = -\tan a \cdot (v\_x - p\_x)$$

. Also ist $v\_y = p\_y - \tan a \cdot (v\_x - p\_x)$. Nur in dem Spezialfall $a = \pi$ oder $a = 0$ ist $v\_y = p\_y$, da der Strahl sich nicht auf der y-Achse bewegt. Unser Code ist also:

```
if a == 0 or a == pi:
        v_y = p_y
else:
        v_y = int(p_y - math.tan(a)*(v_x-p_x))
```

Das Inkrement von Punkt V: Als nächstes müssen wir herausfinden, wie viel nach links/rechts (`x_i`)und wie viel nach oben/unten (`y_i`) wir gehen müssen, um den neuen Punkt zu finden. Diese Werte werden für jede Iteration der Schleife zu den alten Koordinaten addiert. Weil wir immer auf der nächsten Vertikalen überprüfen, ist `x_i` +/- `TILE`. Wie bei `v_x` ist `x_i` positiv, wenn der Strahl nach rechts verläuft, und sonst negativ. Da wir das schon für `v_x` überprüfen, können wir den Code in den gleichen `if`-Block stellen, um dies zu erlangen:

```
if (a < 0.5 * pi) or (a > 1.5 * pi):
        x_i = TILE
        v_x = (p_x // TILE)*TILE + TILE
else:
        x_i = -1 * TILE
        v_x = (p_x // TILE)*TILE - 1
```

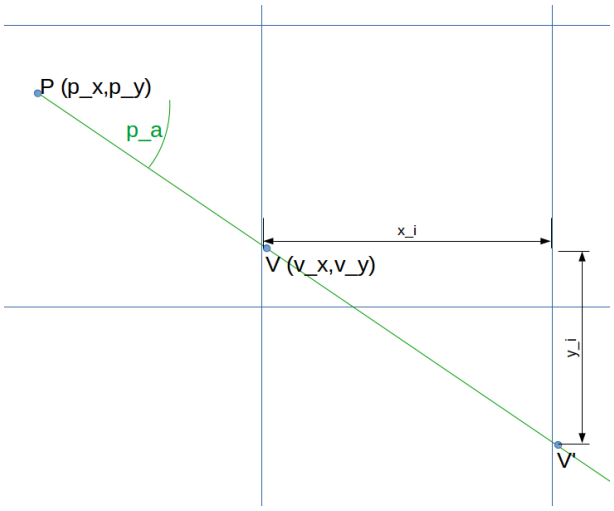

Abbildung 2.2: x_i und y_i

Danach kommt y_i, wozu wir wieder den Tangens brauchen. Dieses Mal ist es aber ein bisschen einfacher: Wieder haben wir

$$\tan a = \frac{-y\_i}{x\_i}$$

aber dieses mal Kennen wir schon `x_i`, also formen wir einfach um zu

$$y\_i = -\tan a \cdot x\_i$$

Auch hier ist $y\_i = 0$, wenn der Strahl horizontal ist. Eigentlich müssen wir nicht einen zusätzlichen `if`-Block dafür haben, denn $\tan 0 = 0$ und $\tan \pi = 0$, aber es hilft,

darüber bewusst zu sein. Jetzt ist unser ganzer Code, um diese Werte zu bestimmen, wie folgt:

```
# Get x_i (+TILE if pointing right, -TILE if left).
# Get v_x (x coord of the first point).
if (a < 0.5 * pi) or (a > 1.5 * pi): # pointing right
        x_i = TILE
        v_x = (p_x // TILE)*TILE + TILE
else:
        x_i = -1 * TILE
        v_x = (p_x // TILE)*TILE - 1

# Get y_i, using tan.
# Get v_y, using magic.
if a == 0 or a == pi: # completely horizontal ray
        y_i = 0
        v_y = p_y
else:
        y_i = int(math.tan(a) * (-1) * x_i)
        v_y = int(p_y - math.tan(a)*(v_x-p_x))
```

Leser, die den Code gelesen haben, haben vielleicht gemerkt, dass bei `y_i` und `v_y` um den Ausdruck die `int()` Funktion benutzt wird. Die Koordinaten müssen noch immer `int`s bleiben, aber python konvertiert in einem Versuch, hilfreich zu sein, `int`s automatisch in `float`s.

Die Schleife: Jetzt, wo wir die nötigen Werte haben, können wir die Schleife beginnen, die durch die Punkte iteriert. In der Schleife prüfen wir einfach, ob der punkt (v_x, v_y) in einem "vollenFeld oder nicht steht. Wie vorher geschrieben, ist dies Methode, um von Koordinaten zu `world`-Koordinaten zu kommen, einfach $\lfloor k \div TILE \rfloor$, in python `k // TILE`.
Die Überprüfung, ob Punkt V auf einem vollen Feld steht, ist also einfach `world[v_x // TILE][v_y // TILE]`: die Zellen sind sowieso 1 oder 0. Wenn das also 1 ist, wird `vvalid` also auf `True` gesetzt, und die Schleife verlassen.
Weil es passieren kann, dass der erste Punkt V schon außerhalb des Spielfeldes ist, ist diese ganze Überprüfung in einem `try` gepackt. Wenn ein `IndexError` passiert, `break`ed es wieder die Schleife und geht weiter zu den horizontalen Überprüfungen. Ansonsten werden zu `v_x` und `v_y` `x_i` bzw. `y_i` addiert, und die Schleife fortgesetzt, solang V noch ein gültiger Punkt ist. Der resultierende Code ist also:

```
# hl is the number of cells along the horizontal
for i in range(hl):
        try:
                if world[v_x // TILE][v_y // TILE]:
                        vvalid = True
                        break
        except IndexError:
                if debug:
                        print("v. error!", v_x,v_y, \
                        "\n Ray:", p_x, p_y, a)
                break
        v_x += x_i
        v_y += y_i
        if v_x >= (hl*TILE) or v_y >= (vl*TILE):
                break
```

Zuletzt wird noch die Distanz zwischen den Punkten P und V berechnet, mit dem Satz des Pythagoras:

```
vdist = math.sqrt((v_x-p_x)**2 + (v_y-p_y)**2)
```

Damit haben wir die Erste vertikale Grenze, die der Strahl schneidet.

2.2.3.3 Horizontale Schnitte

Für die horizontalen Schnittpunkte machen wir ungefähr das gleiche wie für vertikale, nur die Achsen sind vertauscht: `y_i` ist hier +/- `TILE`, und `x_i` muss mit Tangens berechnet werden. Das meiste kann aber leicht von dem vorigen übertragen werden. Zuletzt werden wir eine mögliche Distanz in `hdist` haben, wenn `havlid True` ist.

2.2.3.4 Vergleich und `return`

Jetzt haben wir also minimal 0 Distanzen, und maximal 2. Mit einer einfachen `if` / `elif` Kette können wir daraus die niedrigste Distanz finden: dann haben wir den ersten Punkt, an dem der Strahl auf etwas trifft.
Zuerst sehen wir, ob beide Werte `True` sind. Wenn schon, geben wir die kürzere Distanz zurück:

```
if vvalid and hvalid: # both rays collide
        return min(vdist, hdist)
```

Danach wird einzeln geprüft, ob `vvalid` bzw. `hvalid` Wahr sind. Entsprechend wird dann `vdist` oder `hdist` zurückgegeben:

```
elif vvalid: # only vertical collision
        return vdist
elif hvalid:
        return hdist
```

Schließlich wird -1 zurückgegeben, wenn der Strahl auf nichts trifft:

```
else: # no collision
        return -1
```

Kapitel 3

Zusammenfassung

One or two paragraphs of TL;DR.