(这个模型也可以省略很多 只是一些思路)

[变量设置和解释 2](#_Toc130519304)

[武器拆分 2](#_Toc130519305)

[伤害 SH 2](#_Toc130519306)

[射速 SS 2](#_Toc130519307)

[弹夹消耗光时间 HGSJ 3](#_Toc130519308)

[后坐力函数 HZL 3](#_Toc130519309)

[放大倍率 FDBL 3](#_Toc130519310)

[子弹散布 二维概率密度函数ZDSB(x,y)=p 3](#_Toc130519311)

[换弹时间 HDSJ 3](#_Toc130519312)

[持枪移动速度 CQSD 4](#_Toc130519313)

[射击时移动速度 SJSD 4](#_Toc130519314)

[子弹衰减函数 ZDSJ(MBJL) 4](#_Toc130519315)

[枪托带来的准星自动飘移函数ZDPY(T,X,Y) 4](#_Toc130519316)

[玩家拆分 4](#_Toc130519317)

[目标血量MBXL 4](#_Toc130519318)

[后坐力控制 4](#_Toc130519319)

[反应力 FYL 4](#_Toc130519320)

[目标大小 MBDX=(X,Y) 5](#_Toc130519321)

[目标位置 MBWZ=(X,Y,Z) 5](#_Toc130519322)

[目标移动方式 MBYD(T,MBJL)=(X,Y,Z) 5](#_Toc130519323)

[目标在屏幕的位置SJWZ(MBDX, MBWZ)=(X1,Y1,X2,Y2) 5](#_Toc130519324)

[准星修正方式 7](#_Toc130519325)

[枪械未考虑因素 8](#_Toc130519326)

[计算方法 8](#_Toc130519327)

[如何命中 8](#_Toc130519328)

[子弹伤害期望 8](#_Toc130519329)

[某时刻一发子弹命中率 9](#_Toc130519330)

[连续输出武器 整个弹夹输出 9](#_Toc130519331)

[(后添加的)间断输出武器 整个弹夹输出 9](#_Toc130519332)

[DPS 10](#_Toc130519333)

[简单模型总结 10](#_Toc130519334)

[简单模型再总结 10](#_Toc130519335)

[简化算数的泰勒展开近似 11](#_Toc130519336)

[你是怎么打不中的(复杂模型前提) 12](#_Toc130519337)

[真正的压枪 12](#_Toc130519338)

[走位 14](#_Toc130519339)

[跟枪 14](#_Toc130519340)

[如何总体评价武器的标准 15](#_Toc130519341)

[输出评价 15](#_Toc130519342)

[TTK评价 15](#_Toc130519343)

[附录 程序解释实现 17](#_Toc130519344)

[结构体解释 17](#_Toc130519345)

[函数类似名解释 20](#_Toc130519346)

[拐点序列类型函数 21](#_Toc130519347)

[位置序列添加速度函数 21](#_Toc130519348)

[速度序列添加位置函数 22](#_Toc130519349)

[序列输出T时刻屏幕偏差函数 23](#_Toc130519350)

[序列输出T时刻实际位置函数 24](#_Toc130519351)

[序列输出T时刻屏幕速度偏差函数 25](#_Toc130519352)

[随机分布类函数 27](#_Toc130519353)

[正态分布随机数生成函数 27](#_Toc130519354)

[子弹散布概率密度型函数 27](#_Toc130519355)

[子弹散布累计概率型函数 28](#_Toc130519356)

[武器与目标的一些运动函数(除衰减外以被序列模式替代 但没删除 外一以后用) 30](#_Toc130519357)

[压枪函数 31](#_Toc130519358)

[跟枪函数 33](#_Toc130519359)

[实际位置函数 36](#_Toc130519360)

[使用拐点的复杂模型 36](#_Toc130519361)

[未使用拐点的无压枪跟枪简单模型 37](#_Toc130519362)

[最后计算模型 37](#_Toc130519363)

# 变量设置和解释

首先将准星设为原点

变量解释里(X,Y)不一定是相对于准星的位置

## 武器拆分

### **伤害 SH**

一发子弹的伤害若伤害随时间变化设为关于时间函数 SH(t)

### 射速 SS

单位时间内子弹数量 若伤害随时间变化设为关于时间函数 SS(t)

### 弹夹消耗光时间 HGSJ

### 后坐力函数 HZL

随时间变动的函数HZL(T)=(X,Y) 函数HZL的值域(得出的值)是二元X,Y

X为准星横坐标挪动量 Y为准星纵坐标挪动量

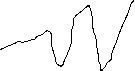
比如某枪械后坐力可以表示成如下函数(随手画的)

纵坐标y

(或者准星左右挪动量)

横坐标x

(或者准星左右挪动量)



时间T(或者子弹数)

时间T(或者子弹数)

### 放大倍率 FDBL

### 子弹散布 二维概率密度函数ZDSB(x,y)=p

x,y为坐标 p为概率

ZDSB函数有旋转不变性可以用极坐标单值函数表示ZDSB(φ)

不过为了后面程序便于计算还是易于用2维坐标表示

简单解释函数ZDSB作用(并非武器散布 只是例子)

子弹击中绿圈内概率为100%(并非具体数字 只是例子)

子弹击中蓝圈内概率为70%

子弹击中绿圈内概率为50%

### 换弹时间 HDSJ

### 持枪移动速度 CQSD

### 射击时移动速度 SJSD

### 子弹衰减函数 ZDSJ(MBJL)

函数HZL的定义域(得出的值)是衰减百分比 自变量MBJL是目标距离(MBJL变量介绍在后面)

### 枪托带来的准星自动飘移函数ZDPY(T,X,Y)

准星自动飘移为布朗运动 可以描述为概率函数

函数ZDPY的定义域是三元T,X,Y T为时间 X,Y为挪动量 值域是在T时刻 准星在坐标X,Y的概率

在这里用ZDPY对T的偏数 表示微小时刻X,Y可能位移位置的概率密度

在程序里用蒙特卡洛(随机大数量模拟)可以表示为在每个T时刻后对坐标X,Y加一个随机数

## 玩家拆分

### 目标血量MBXL

### 后坐力控制

简单分析 对后坐力函数返回的位移量乘一个数

复杂分析 对于不同玩家重新绘制不同的后坐力函数(是不是太麻烦了)

这个后面再讨论

### 反应力 FYL

对于目标中心对 准星的偏移做一个修正 及目标中心在(A1,B1)时在FYL时刻后对准星进行一个修正(例如简单修正:准星以速度XX向方向A1,B1位移XX时刻)

### 目标大小 MBDX=(X,Y)

目标在游戏中判定框的大小长Y宽X

用方框模型简化运算

目标可以由多个方框组合

每一个可以线性相加

### 目标位置 MBWZ=(X,Y,Z)

目标中心位置在相对于玩家原点的位置(X,Y,Z)

### 目标移动方式 MBYD(T,MBJL)=(X,Y,Z)

目标在自身移动时以时间和目标距离为自变量 返回目标中心偏移量

### 目标在屏幕的位置SJWZ(MBDX, MBWZ)=(X1,Y1,X2,Y2)

在这里我们简化目标可以攻击到的判刑框为方形 输入目标位置 目标距离 目标大小 返回目标简单判定框左下点 和右上点对于准星的相对坐标

这里给出一般游戏显示公式

屏幕

游戏距离长度为1/3

但是在屏幕上就不是了

所以要乘上放缩系数FSXS

固定长度PMJL (游戏距离)

距离原点3个单位的长宽高都为1个单位的目标

(都是游戏内距离)

视线原点

我们拿到游戏距离的目标移动最终要换算成在屏幕的像素移动的

为什么先算目标中心和判定框在屏幕的位置不先算判定框左下右上位置再算到屏幕上

因为后面可能会有玩家对目标中心的跟抢函数

在视野中的目标位置中心在屏幕坐标为(无后座)(无移动)(无变形)

目标判定框占据屏幕大小(无后座) (无移动)(无变形)

实际位置(无后座) (无移动)(无变形)

在视野中的目标位置中心在屏幕坐标为

实际位置

### 准星修正方式

对于反应准星和目标中心的偏差的修正方式

0阶预判 就是没有预判 只将准星以速度为X修正到反应时刻T秒(FYL)前的位置

1阶预判 对于目标的简单线性位移进行额外修正 将准星以速度为X修正到预测度过反应时刻T秒(FYL)目标位置 并根据目标移动速度Y跟随目标移动

2阶预判 对于目标的移动函数进行二次拟合 将准星以速度为X修正到预测度过反应时刻T秒(FYL)目标位置 并根据目标移动速度Y和加速度A跟随目标变速度移动

举例 目标跳跃运动

目标X

时间T

准星X

时间T

0阶预判

目标X

时间T

1阶预判

目标X

时间T

2阶预判(这里应该是一段一段二次曲线拼接 做不到只能鼠绘了)

每个折点(两端平滑直线(或曲线)相交处)意味着反应到目标与准星差距后进行修正

0阶只能将准星瞄到反应时的位置 在下一次反应之前不会移动准星 和目标差距大



1阶可以将准星瞄到反应时的位置后 对目标之后的位置预判并跟枪 在下一次反应之前会移动准星近似跟踪目标 但对于目标减速后再加速跟踪效果有待提高

2阶可以将准星瞄到反应时的位置后 对目标之后的位置预判并跟枪 例如在目标左位移急停后可以在下一次反应前向右方射击(有时可能被左位移急停后再次左移骗到,但是拿函数类比 2阶可以做到拟合(一段一段的抛物线) 但对在0处停止后继续增加无法理解



所以2阶对左位移急停后再次左移骗到)

### 枪械未考虑因素

子弹下坠 可以通过瞄准镜刻度或者开枪观察落点修正

子弹飞行速度 除非超长距离或者你是在玩战舰世界 子弹速度超出正常人反应时间

射击时枪口影响 使得玩家模型过于复杂

不同准星的变化

# 计算方法

## 如何命中

### 子弹伤害期望

在抛去出伤害 散布 目标外所有因素

对最简单模型分析对于已知ZDSB和SJDX=(X1,Y1,X2,Y2)

子弹伤害期望为

只计算圆和方框相交区域的积分

### 某时刻一发子弹命中率

目标不移动 玩家不跟 准星不随机抖动

### 连续输出武器 整个弹夹输出

连续输出武器 射速越快越接近模型

子弹伤害若随时间增加(ps比如无主之地某些枪)设为关于时间函数SH(t) 若不随时间增加设为常数SH

射速同理(pss还是比如无主之地某些枪)

这个决定了一个武器在只有散布和后座时武器的整体期望伤害

设整体输出函数ZTSC

### (后添加的)间断输出武器 整个弹夹输出

对于每一发子弹间断计算其理想伤害

间断输出武器 可以适合任何枪械 计算量会增大(ps 也许可能离散算更快 毕竟积分是离散和两个t相隔无穷小的极限)

比如射速为2子弹为6枪 0.5秒一枪(ps应该是单位时间,这里默认成秒)

则计算0.5秒期望伤害 1秒期望伤害 1.5秒期望伤害 2.0秒期望伤害2.5秒期望伤害

累加起来得到弹夹输出

其他同连续模型

### DPS

### 简单模型总结

在攻击长宽都为单位1的单位在单位1的时间内期望总伤害

目前简化模型使用 后坐力HZL 子弹散布ZDSB

目标距离 射速 伤害 目标位置 目标大小 耗光时间均设为常量

反应力 子弹衰减 准星修正方式为无

缺点 无法对可以打多个弹夹射速累计的进行计算(ps如复仇女神 pss这个也好修改 不过会增加工作量)

默认摁住连射而不是玩家控制射击(ps玩家控制射击利用间断模型可以解决pss但是玩家控制的射击时间和瞄准方式与反应力有关)

### 简单模型再总结

引理

证明简化公式

看起来很长

但是从普通三重积分变为了普通定积分

如果要问积分原函数的C怎么没有 是不是不严谨

只能说我有一个巧妙的证明 只可惜这里地方太小

关键是如何求

离散形式

### 简化算数的泰勒展开近似

一维正态分布函数 均值,最大值在原点 方差为

二维正态分布函数 均值,最大值在原点 x,y的方差为都

一维泰勒展开 (证明略)

二维泰勒展开

因为计算机无法无限运算,在这里将 概率密度<0.001的省略为0

在正态分布-3.5倍σ到3.5倍σ处 累计密度为0.9995

在n=20时 在误差小于0.0001

(已找到简化方法 泰勒展开废弃)

正态分布

## 你是怎么打不中的(复杂模型前提)

### 真正的压枪

简单模型 在HZL函数后乘一个放缩系数a(太简单粗暴了)

复杂模型 对于不同人重新绘制后坐力函数(那运算还要自己重绘后坐力 开始的后坐力函数还有必要吗)

我们将压枪拆分 本质上还是准星和想打的目标点有偏移 但是你知道大概偏移函数

我们将后坐力函数拆分成几个等级

等级1 无后座 如同激光笔一样 是个人都可以压枪



等级2 近似的向一个方向匀速移动 参数

对于大多数人 有一定的fps/tps基础的人 可以立刻对这种后坐力进行压枪

哪怕没有基础 进行简单的训练也可以做到压枪

对于方向和速度的掌握有微小的不确定性



大多数的人可以简化成一个如图

黑色箭头为后坐力方向

蓝色箭头都为可能的压枪方向

HZL函数可以变成在圆内的布朗运动(随机运动)

程序实现为

或者在 为随机分布中取一点(ps取正态分布的话后坐力分布和子弹散布不相关 若子弹也取正态分布两者和可以直接等于一个新正态分布 就只考虑目标运动的偏差了 若目标无偏差直接正态分布积分可得出理想伤害)

其中的随机分布随时间增大而线性增大

或者为如下模型

其中和由a控制

等级3 近似的向一个方向加速移动 参数



对于方向的掌握有微小的不确定性

速度不确定性较大

不过也可以简化成等级2模型 概率密度区域要大一些

或者为的随机分布随时间增大而二次增大

等级4 近似为折线 参数

绝大多少的fps/tps都在使用的后坐力去掉微小抖动后都可以简化为如下图的折现模型

对于有基础的玩家 在处理不陌生但不熟练的一种后坐力时 大致的压枪行为

其中黑色为后坐力产生的子弹偏移轨迹

蓝色箭头为准星进行压枪的可能规矩

压枪与后坐力之间偏差由来

1 对于方向和速度的不准确性这种在前面已经讨论过

在这里近似为正太分布

2对于压枪转折时机的不正确性 人是有反应力的 对于后坐力方向的

转向是有延迟 对于没有练习的人 在转折时刻会继续按照之前的速度移动移动长的时刻 即在移动了

在t时刻转向

在ti时刻没有反应过来

在ti+hzl时刻有反应过来并向修正方向移动

修正方向向量

在偏差位置向下一段线段中某一点方向修正

C=1时向终点修正准星 c=0时向起点修正准星(更打不中了)

下面是完整压枪过程函数

首先将拆解为多个直线

后坐力第一段的压枪的计算

在时准星偏差

第二段压枪的计算

后面几阶段压枪以此类推

等级5 超越正常反应上限的转向间隔或完全随机散布

这阶段没什么好说的 正常人完全无法压枪 近似为随机散布比较好(瓦洛兰特的机枪后面几十发)

### 走位

后座的准星的偏移MBWZ不止取决于你的压枪还取决于对方的走位MBYD

### 跟枪

# 如何总体评价武器的标准

## 输出评价

假设简单情况 (ps 见简单模型总结) (全是废话)

简单线性函数

缺点 对于不同武器配件价值无法简单等价

设只计算一弹夹内DPS x=1

设武器1配件0 DPS=25 武器1配件1 DPS=50

武器2配件0 DPS=1025 武器2配件2 DPS=1050

显然计算出配件1于配件2等价 这是不对的

## TTK评价

一个武器的杀人效率才是重要的评价标准(前面全是废话)

我们用TTK来评价武器

如果一个弹夹的整体输出ZTSC > MBXL

则期望TTK

容错输出=ZTSC-MBXL

如果在第n-1个弹夹的整体输出 (n-1)\*ZTSC<MBXL

在第n个弹夹的输出n\*ZTSC>MBXL

则期望TTK

简单来说带换弹击杀目标时间

容错输出=

在同样TTK下容错越多越好

但是不同TTK的容错很难判断

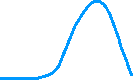
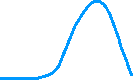
正所谓 给我十枪猎兽不如一把喷火

对于只有一把武器的游戏

如下模型可以大致描述武器对于不同玩家的优劣

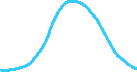
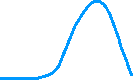
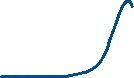
对于相同的枪(除子弹数量和耗光时间外无不同)

整体的理想单弹夹输出和期望单弹夹输出变高了



虽然武器TTK没有变 但是容错变高了 失误函数带来的额外换弹时间变少了

所以可以说武器换了弹夹会更强一些(ps特殊情况 玩家很强 强到期望十分接近理论 那么在理想单弹夹输出/目标血量>0.7的情况下 弹夹提升非常微小 我们的改进TTK很符合这点)(pss 玩家越强 期望越接近理论 函数越偏锋如下)



新人 正常 高手

能打出正常能打出一半伤害 能打出平常能打出3/4伤害 在大部分情况十分接近理论值

函数怎么绘制还没想好 找到好几个 都有点符合 但没那么符合

要么新人正常的符合就不会符合高手 要么正常高手的符合就不符合新人

(ps这下黑体辐射了)

(pss而且函数确实像)

# 附录 程序解释实现

## 结构体解释

#include <math.h>

#include <iostream>

#include <cstdlib>

#include <cmath>

#include <limits>

using namespace std;

以上是头文件 表示我要去借用别人的代码有哪些

const double PI = 3.14159265358979323846;

定义常数PI=3.14…

double 游戏距离到屏幕距离放缩系数 = 1;

即在游戏内实际长度为单位1的线 在距离实际长度z个单位时 在屏幕的长度为

放缩系数/z 详细解释见目标在屏幕位置

typedef struct 玩家技术 {//定义在最开始 后面有函数用到这个结构体

double 简单压枪倍率;

double 简单目标跟踪倍率;

double 压枪方向熟练程度;

double 压枪速度熟练程度;

double 压枪时间熟练程度;

double 跟枪速度熟练程度;

double 跟枪方向熟练程度;

double 反应力;

double 跟枪等级;

}玩家技术;

其中简单压枪倍率和跟踪倍率已经不在复杂模型使用 压枪方向熟练程度 压枪速度熟练程度 压枪时间熟练程度 为下图

黑色箭头为后座 长度为速度大小 方向为箭头方向

蓝色箭头为压枪 长度为速度大小 方向为箭头反方向

速度熟练度越高 两者长度越小

方向熟练度越高 两者夹角越小

时间熟练度越高 转折时间越接近

跟枪速度和跟枪方向同理

跟枪等级为 章节 跟枪中的跟枪等级

typedef struct 屏幕偏差 {//目标点距离准星屏幕偏差

double X\_屏幕;

double Y\_屏幕;

}屏幕偏差;

目标在距离准星的(X\_屏幕, Y\_屏幕)位置 如下



X\_屏幕=2, Y\_屏幕=1.6

X\_屏幕=-1.7, Y\_屏幕=1



X\_屏幕=1.8, Y\_屏幕=-0.4



typedef struct 目标中心实际位置 {//目标位置距离原点偏差

double X\_实际;//目标位置距离原点偏差x

double Y\_实际;//目标位置距离原点偏差y

double Z\_实际;//目标位置距离原点偏差z

}目标中心实际位置;

目标在距离玩家游戏内中心的(X\_实际, Y\_实际, Z\_实际)位置 如下

X\_实际=1.7, Y\_实际=1.8, Z\_实际=0.9



X\_轴

Z\_轴

Y\_轴

typedef struct 屏幕拐点序列 {//记录了后坐力发生剧烈转折时子弹位置与时间或者子弹在不压枪情况下子弹位置和距离第一次开火时间

double 位置\_X\_屏幕[100] = { 0 };

double 位置\_Y\_屏幕[100] = { 0 };

double 开火时间\_T[100] = { 0 };

double 速度\_X\_屏幕[100] = { 0 };//在这里用空间换时间 速度要重复计算 不如直接存起来

double 速度\_Y\_屏幕[100] = { 0 };

double 加速度\_X\_屏幕[100] = { 0 };

double 加速度\_Y\_屏幕[100] = { 0 };

}屏幕拐点序列;//这里直接输入所有子弹的位置与时间也可以,直接将后坐力拆成超多拐点的折现 子弹XY坐标也可以做成后坐力偏差类型 建议数组 拐点[0]={0,0,0}

这个记录了某些作用对屏幕的影响,这些影响应该是与目标Z位置无关的(比如后座,鼠标移动带来的压枪和跟枪)

对于输入的转折点 其中速度\_X\_屏幕,速度\_Y\_屏幕,加速度\_X\_屏幕,加速度\_Y\_屏幕,是没必要,我们建立它的目的是减少计算(利用内存空间,节省运算时间)

我们输入的位置\_X\_屏幕,位置\_Y\_屏幕,开火时间\_T是什么形式 解释如下图



简化为

=>



他们的X与Y坐标见屏幕偏差 我们还要额外记录他们的出现的时间

typedef struct 实际拐点序列 {//记录了游戏内的 实际 坐标的拐点序列

double 位置\_X\_实际[100] = { 0 };

double 位置\_Y\_实际[100] = { 0 };

double 位置\_Z\_实际[100] = { 10 };

double 转折时间\_T[100] = { 0 };

double 速度\_X\_实际[100] = { 0 };//在这里用空间换时间 速度要重复计算 不如直接存起来

double 速度\_Y\_实际[100] = { 0 };

double 速度\_Z\_实际[100] = { 0 };

double 加速度\_X\_实际[100] = { 0 };

double 加速度\_Y\_实际[100] = { 0 };

double 加速度\_Z\_实际[100] = { 0 };

}实际拐点序列;

这个记录了某些作用对游戏内距离的影响,这些影响应该是与目标位置有关的(比如目标与玩家的相对移动).

其他道理同上.

typedef struct 目标大小 {//目标简单判定框大小 三维立方体判定框在屏幕存在斜线 不可以使用一重积分方法 复杂模型可以由多个框组成 这里暂时使用单个方框

double X\_实际;//方形判断框的宽

double Y\_实际;//方形判断框的长

}目标大小;

这个记录了目标判定框在游戏内大小,被下方的序列替代.

typedef struct 目标大小序列 {//目标简单判定框大小随时间变化序列 复杂模型可以由多个框组成 这里暂时使用单个方框

double X\_实际[10] = { 10 };//方形判断框的宽

double Y\_实际[10] = { 10 };//方形判断框的长

double 变形时间T[10] = { 0 };

}目标大小序列;

这个记录了目标判定框在游戏内大小,在T时间后变为另一种大小,包括了上面那个结构.

typedef struct 边框实际位置 {//目标在屏幕实际位置坐标

double 左下角\_X\_屏幕;

double 左下角\_Y\_屏幕;

double 右上角\_X\_屏幕;

double 右上角\_Y\_屏幕;

}边框实际位置;

这个记录了目标判定框在屏幕的左下角右上角(X\_屏幕,Y\_屏幕)坐标.

typedef struct 目标总体 {//说实话 一些函数已经完全被序列替代了

目标大小 目标游戏内起始大小;

目标大小序列 目标游戏内大小序列;

目标大小函数类型 目标游戏内大小函数指针;

目标中心实际位置 目标游戏内起始位置;

目标移动函数类型 目标游戏内移动函数指针;

实际拐点序列 移动序列;

int 目标血量;

边框实际位置 目标在屏幕位置;

}目标总体;

青蓝色字体是他的类型,后面黑色是名字.

记录了目标所有的基本属性,在复杂模型中(目标游戏内起始大小 目标游戏内大小函数指针 目标游戏内起始位置 目标游戏内移动函数指针)已经废弃.

我们使用两个序列描述大小与位置

typedef struct 枪械 {//枪械的一个集合 包含许多基本要素

int 武器伤害;//伤害常量

float 武器射速;//射速

float 耗光时间;//弹夹耗光时间

float 放大倍率;//放大倍率

float 换弹时间;//换弹时间

float 持枪移动速度;//持枪时移动速度

float 射击移动速度;//射击时移动速度

后坐力函数类型 后坐力函数指针;

屏幕拐点序列 枪械子弹序列;//预设100发子弹的后座位置或者关键拐点位置

子弹衰减函数类型 子弹衰减函数指针;

double 子弹散布参数;

概率密度函数类型 子弹概率密度函数指针;

概率密度函数类型 子弹概率累计函数指针;

}枪械;

记录了一个枪械所有的基本属性,在复杂模型中(后坐力函数指针)已经废弃.

(持枪移动速度和射击移动速度)未被使用,我们使用序列描述后坐力,对于不同游戏可能散布函数不一样,我们使用子弹概率指针指向不同函数.不同枪械,子弹衰减也有不同,我们用指针指向不同函数.

## 函数类似名解释

typedef 屏幕偏差(\*后坐力函数类型)(double);//下方所有类型定义都为函数指针

原来描述后坐力的函数的类型,这一类函数输入(时间),输出(该时间的偏差坐标),不同后座需要同一类型的不同函数,这个指针可以选择一个其中后坐力,已经废弃

typedef double(\*概率密度函数类型)(double, double, double);

描述只考虑散布子弹命中率的函数的类型,函数输入(子弹坐标X,Y和子弹散布大小),输出(该子弹命中概率),不同游戏可能用不同函数,这个指针可以选择一个其中分布,不过大部分可能使用正态分布

typedef double(\*子弹衰减函数类型)(double);

函数输入(目标Z轴距离),输出(子弹衰减倍率)不同枪械衰减倍率不同, 指针可以选择一个衰减.

typedef 目标大小(\*目标大小函数类型)(double);

函数输入(时间),输出(目标大小)被序列替代,已经废弃

typedef 目标中心实际位置(\*目标移动函数类型)(double, 目标中心实际位置);

函数输入(时间),输出(目标实际位置)被序列替代,已经废弃

## 拐点序列类型函数

### 位置序列添加速度函数

屏幕拐点序列 屏幕拐点位置\_转速度函数(屏幕拐点序列 位置拐点) {//对于只有位置的拐点序列进行一阶近似 添加其速度

int 数组指针;

for (数组指针 = 0; 数组指针 < 99; 数组指针++) {

if (位置拐点.开火时间\_T[数组指针 + 1] == 0) {//从数组1开始搜索 因为末尾后没定义的数组默认为0 所以如果循环到末尾 结束 下同

break;

}

double 时间差 = 位置拐点.开火时间\_T[数组指针 + 1] - 位置拐点.开火时间\_T[数组指针];//避免重复计算 下同

位置拐点.速度\_X\_屏幕[数组指针] = (位置拐点.位置\_X\_屏幕[数组指针 + 1] - 位置拐点.位置\_X\_屏幕[数组指针]) / 时间差;

位置拐点.速度\_Y\_屏幕[数组指针] = (位置拐点.位置\_Y\_屏幕[数组指针 + 1] - 位置拐点.位置\_Y\_屏幕[数组指针]) / 时间差;

位置拐点.加速度\_X\_屏幕[数组指针] = 0;

位置拐点.加速度\_Y\_屏幕[数组指针] = 0;

}

return 位置拐点;

}

输入的屏幕拐点序列可以只存拐点的(位置X,位置Y,时间T)

函数将一阶近似,将拐点后转折的方向与速度输入如下图

函数计算出

=>



简化为

=>



原理 for(i=0;i<100;i++)遍历

{速度X=(位置X[i+1]-位置X[i])/时间差

速度Y=(位置Y[i+1]-位置Y[i])/时间差}

速度=路程差/时间差

实际拐点序列 实际拐点位置\_转速度函数(实际拐点序列 位置拐点) {//对于只有位置的拐点序列进行一阶近似 添加其速度

int 数组指针;

for (数组指针 = 0; 数组指针 < 99; 数组指针++) {

if (位置拐点.转折时间\_T[数组指针 + 1] == 0) {

break;

}

double 时间差 = 位置拐点.转折时间\_T[数组指针 + 1] - 位置拐点.转折时间\_T[数组指针];

位置拐点.速度\_X\_实际[数组指针] = (位置拐点.位置\_X\_实际[数组指针 + 1] - 位置拐点.位置\_X\_实际[数组指针]) / 时间差;

位置拐点.速度\_Y\_实际[数组指针] = (位置拐点.位置\_Y\_实际[数组指针 + 1] - 位置拐点.位置\_Y\_实际[数组指针]) / 时间差;

位置拐点.速度\_Z\_实际[数组指针] = (位置拐点.位置\_Z\_实际[数组指针 + 1] - 位置拐点.位置\_Z\_实际[数组指针]) / 时间差;

位置拐点.加速度\_X\_实际[数组指针] = 0;

位置拐点.加速度\_Y\_实际[数组指针] = 0;

位置拐点.加速度\_Y\_实际[数组指针] = 0;

}

return 位置拐点;

}

原理同上,只不过多了Z坐标,不多解释

### 速度序列添加位置函数

屏幕拐点序列 屏幕拐点速度\_转位置函数(屏幕拐点序列 速度拐点) {//对于没有位置拐点序列进行简单积分 添加其偏差

int 数组指针;

for (数组指针 = 0; 数组指针 < 99; 数组指针++) {

if (速度拐点.开火时间\_T[数组指针 + 1] == 0) {

break;

}

double 时间差 = 速度拐点.开火时间\_T[数组指针 + 1] - 速度拐点.开火时间\_T[数组指针];

速度拐点.位置\_X\_屏幕[数组指针 + 1] = 速度拐点.速度\_X\_屏幕[数组指针] \* 时间差 + (速度拐点.加速度\_X\_屏幕[数组指针] / 2) \* 时间差 \* 时间差;

速度拐点.位置\_Y\_屏幕[数组指针 + 1] = 速度拐点.速度\_Y\_屏幕[数组指针] \* 时间差 + (速度拐点.加速度\_Y\_屏幕[数组指针] / 2) \* 时间差 \* 时间差;

}

return 速度拐点;

}

若输入的屏幕拐点序列只存拐点的(速度X,速度Y,加速度X,加速度Y,时间T)

我们通过 路程=速度\*时间+加速度\*时间\*时间/2

将路程累加起来 如下图

我们不知道速度的起始位置



我们将箭头按照顺序拼装起来,得到下一个箭头的起始位置(小黑点)

实际拐点序列 实际拐点位置\_转速度函数(实际拐点序列 位置拐点) {//对于只有位置的拐点序列进行一阶近似 添加其速度

int 数组指针;

for (数组指针 = 0; 数组指针 < 99; 数组指针++) {

if (位置拐点.转折时间\_T[数组指针 + 1] == 0) {

break;

}

double 时间差 = 位置拐点.转折时间\_T[数组指针 + 1] - 位置拐点.转折时间\_T[数组指针];

位置拐点.速度\_X\_实际[数组指针] = (位置拐点.位置\_X\_实际[数组指针 + 1] - 位置拐点.位置\_X\_实际[数组指针]) / 时间差;

位置拐点.速度\_Y\_实际[数组指针] = (位置拐点.位置\_Y\_实际[数组指针 + 1] - 位置拐点.位置\_Y\_实际[数组指针]) / 时间差;

位置拐点.速度\_Z\_实际[数组指针] = (位置拐点.位置\_Z\_实际[数组指针 + 1] - 位置拐点.位置\_Z\_实际[数组指针]) / 时间差;

位置拐点.加速度\_X\_实际[数组指针] = 0;

位置拐点.加速度\_Y\_实际[数组指针] = 0;

位置拐点.加速度\_Y\_实际[数组指针] = 0;

}

return 位置拐点;

}

同上 多一种坐标Z

### 序列输出T时刻屏幕偏差函数

屏幕偏差 屏幕拐点型\_转屏幕偏差函数(屏幕拐点序列 屏幕拐点, double 时间变量) {//输入时间和拐点序列 输出在T时刻的屏幕偏差

屏幕偏差 后坐力偏差临时变量 = { 0,0 };

int 数组指针 = 0;

while (屏幕拐点.开火时间\_T[数组指针 + 1] <= 时间变量) {//寻找一个开火时间[i]<时间<开火时间[i+1]的位置 下同

数组指针++;

if (数组指针 >= 100) {//找遍所以未找到对应 强制结束 下同

return 后坐力偏差临时变量;

}

}

double 时间差 = 时间变量 - 屏幕拐点.开火时间\_T[数组指针];

后坐力偏差临时变量.X\_屏幕 = 屏幕拐点.位置\_X\_屏幕[数组指针] + 屏幕拐点.速度\_X\_屏幕[数组指针] \* (时间差) + 屏幕拐点.加速度\_X\_屏幕[数组指针] \* (时间差) \* (时间差) / 2;

后坐力偏差临时变量.Y\_屏幕 = 屏幕拐点.位置\_Y\_屏幕[数组指针] + 屏幕拐点.速度\_Y\_屏幕[数组指针] \* (时间差) + 屏幕拐点.加速度\_Y\_屏幕[数组指针] \* (时间差) \* (时间差) / 2;

return 后坐力偏差临时变量;

}

While语句将寻找到一个区间,开火时间T[i]<时间变量(后面称为小t)<开火时间T[i+1]

T[i+1]的点

t时后坐力偏移

T[i]的点



If语句表示若搜索完全部序列仍未找到 时间变量<开火时间T[i+1] 将返回{0,0}

t时刻的X偏差=T时刻X偏差+T时刻速度X偏差\*时间差+ T时刻加速度X偏差/2\*时间差\*时间差

t时刻的Y偏差同理

屏幕偏差 实际拐点型\_转屏幕偏差函数(实际拐点序列 实际拐点, double 时间变量) {//输入时间和拐点序列 输出在T时刻的屏幕偏差

目标中心实际位置 目标位置临时变量 = { 0,0,0 };

屏幕偏差 后坐力偏差临时变量 = { 0,0 };

int 数组指针 = 0;

while (实际拐点.转折时间\_T[数组指针 + 1] <= 时间变量) {

数组指针++;

if (数组指针 >= 100) {

return 后坐力偏差临时变量;

}

}

double 时间差 = 时间变量 - 实际拐点.转折时间\_T[数组指针];

目标位置临时变量.X\_实际 = 实际拐点.位置\_X\_实际[数组指针] + 实际拐点.速度\_X\_实际[数组指针] \* (时间差) + 实际拐点.加速度\_X\_实际[数组指针] \* (时间差) \* (时间差) / 2;

目标位置临时变量.Y\_实际 = 实际拐点.位置\_Y\_实际[数组指针] + 实际拐点.速度\_Y\_实际[数组指针] \* (时间差) + 实际拐点.加速度\_Y\_实际[数组指针] \* (时间差) \* (时间差) / 2;

目标位置临时变量.Z\_实际 = 实际拐点.位置\_Z\_实际[数组指针] + 实际拐点.速度\_Z\_实际[数组指针] \* (时间差) + 实际拐点.加速度\_Z\_实际[数组指针] \* (时间差) \* (时间差) / 2;

后坐力偏差临时变量.X\_屏幕 = 目标位置临时变量.X\_实际 \* 游戏距离到屏幕距离放缩系数 / 目标位置临时变量.Z\_实际;

后坐力偏差临时变量.Y\_屏幕 = 目标位置临时变量.Y\_实际 \* 游戏距离到屏幕距离放缩系数 / 目标位置临时变量.Z\_实际;

return 后坐力偏差临时变量;

}

While语句和if语句作用同上

t时刻的(X,Y,Z)位置同上

t时刻的X偏差=t时刻的X位置\*放缩系数/ t时刻的Z位置

t时刻的Y偏差同理

### 序列输出T时刻实际位置函数

目标中心实际位置 实际拐点型\_转实际位置函数(实际拐点序列 实际拐点, double 时间变量) {

目标中心实际位置 目标位置临时变量 = { 0,0,0 };

int 数组指针 = 0;

while (实际拐点.转折时间\_T[数组指针 + 1] <= 时间变量) {

数组指针++;

if (数组指针 >= 100) {

return 目标位置临时变量;

}

}

double 时间差 = 时间变量 - 实际拐点.转折时间\_T[数组指针];

目标位置临时变量.X\_实际 = 实际拐点.位置\_X\_实际[数组指针] + 实际拐点.速度\_X\_实际[数组指针] \* (时间差)+实际拐点.加速度\_X\_实际[数组指针] \* (时间差) \* (时间差) / 2;

目标位置临时变量.Y\_实际 = 实际拐点.位置\_Y\_实际[数组指针] + 实际拐点.速度\_Y\_实际[数组指针] \* (时间差)+实际拐点.加速度\_Y\_实际[数组指针] \* (时间差) \* (时间差) / 2;

目标位置临时变量.Z\_实际 = 实际拐点.位置\_Z\_实际[数组指针] + 实际拐点.速度\_Z\_实际[数组指针] \* (时间差)+实际拐点.加速度\_Z\_实际[数组指针] \* (时间差) \* (时间差) / 2;

return 目标位置临时变量;

}

与输出屏幕偏差类型同理 将屏幕偏差换成实际位置而已

### 序列输出T时刻屏幕速度偏差函数

屏幕偏差 屏幕拐点型\_转屏幕速度偏差函数(屏幕拐点序列 屏幕拐点, double 时间变量) {//输入时间和拐点序列 输出在T时刻的屏幕偏差的速度

屏幕偏差 后坐力速度偏差临时变量 = { 0,0 };

int 数组指针 = 0;

while (屏幕拐点.开火时间\_T[数组指针 + 1] <= 时间变量) {

数组指针++;

if (数组指针 >= 100) {

return 后坐力速度偏差临时变量;

}

}

double 时间差 = 时间变量 - 屏幕拐点.开火时间\_T[数组指针];

后坐力速度偏差临时变量.X\_屏幕 = 屏幕拐点.速度\_X\_屏幕[数组指针] + 屏幕拐点.加速度\_X\_屏幕[数组指针] \* (时间差);

后坐力速度偏差临时变量.Y\_屏幕 = 屏幕拐点.速度\_Y\_屏幕[数组指针] + 屏幕拐点.加速度\_Y\_屏幕[数组指针] \* (时间差);

return 后坐力速度偏差临时变量;

}

求速度就是对位置函数对t(时间变量)求导

即t时刻的X偏差=T时刻X偏差+T时刻速度X偏差\*时间差+ T时刻加速度X偏差/2\*时间差\*时间差 的导数

t时刻的速度X偏差= T时刻速度X偏差+ T时刻加速度X偏差\*时间差

t时刻的速度Y偏差同理

屏幕偏差 实际拐点型转屏幕速度偏差函数(实际拐点序列 实际拐点, double 时间变量) {

中心实际位置 目标位置临时变量 = { 0,0,0 };

屏幕偏差 后坐力速度偏差临时变量 = { 0,0 };

int 数组指针 = 0;

while (实际拐点.转折时间T[数组指针 + 1] <= 时间变量) {

数组指针++;

if (数组指针 >= 100) {

return 后坐力速度偏差临时变量;

}

}

double 时间差;

时间差 = 时间变量 - 实际拐点.转折时间T[数组指针];

目标位置临时变量.Z\_实际 = 实际拐点.位置\_Z\_实际[数组指针] + 实际拐点.速度\_Z\_实际[数组指针] \* (时间差)+实际拐点.加速度\_Z\_实际[数组指针] \* (时间差) \* (时间差) / 2;

后坐力速度偏差临时变量.X\_屏幕 = ((实际拐点.速度\_X\_实际[数组指针] \* 实际拐点.位置\_Z\_实际[数组指针] - 实际拐点.速度\_Z\_实际[数组指针] \* 实际拐点.位置\_X\_实际[数组指针] + (实际拐点.加速度\_X\_实际[数组指针] \* 实际拐点.位置\_Z\_实际[数组指针] - 实际拐点.位置\_X\_实际[数组指针] \* 实际拐点.加速度\_Z\_实际[数组指针]) \* 时间差 + ((实际拐点.加速度\_X\_实际[数组指针] \* 实际拐点.速度\_Z\_实际[数组指针] - 实际拐点.速度\_X\_实际[数组指针] \* 实际拐点.加速度\_Z\_实际[数组指针]) / 2) \* 时间差 \* 时间差) \* 游戏距离到屏幕距离放缩系数) / (目标位置临时变量.Z\_实际 \* 目标位置临时变量.Z\_实际);

后坐力速度偏差临时变量.Y\_屏幕 = ((实际拐点.速度\_Y\_实际[数组指针] \* 实际拐点.位置\_Z\_实际[数组指针] - 实际拐点.速度\_Z\_实际[数组指针] \* 实际拐点.位置\_Y\_实际[数组指针] + (实际拐点.加速度\_Y\_实际[数组指针] \* 实际拐点.位置\_Z\_实际[数组指针] - 实际拐点.位置\_Y\_实际[数组指针] \* 实际拐点.加速度\_Z\_实际[数组指针]) \* 时间差 + ((实际拐点.加速度\_Y\_实际[数组指针] \* 实际拐点.速度\_Z\_实际[数组指针] - 实际拐点.速度\_Y\_实际[数组指针] \* 实际拐点.加速度\_Z\_实际[数组指针]) / 2) \* 时间差 \* 时间差) \* 游戏距离到屏幕距离放缩系数) / (目标位置临时变量.Z\_实际 \* 目标位置临时变量.Z\_实际);

return 后坐力速度偏差临时变量;

}

设X实际位移函数

设Z实际位移函数

屏幕位置X函数为

设

所以屏幕速度X函数为 上面屏幕位置X函数对时间求导

## 随机分布类函数

### 正态分布随机数生成函数

double 正态分布随机数(double 方差, double 期望)//见Box-Muller算法

{

const double 双浮点最小值 = std::numeric\_limits<double>::min();

double 正态分布随机数;

double 均匀分布随机数1, 均匀分布随机数2;

do

{

均匀分布随机数1 = rand() \* (1.0 / RAND\_MAX);

均匀分布随机数2 = rand() \* (1.0 / RAND\_MAX);

} while (均匀分布随机数1 <= 双浮点最小值);

正态分布随机数 = sqrt(-2.0 \* log(均匀分布随机数1)) \* cos(2 \* PI \* 均匀分布随机数2);

return 正态分布随机数 \* 方差 + 期望;

};

同注释 建议搜索Box-Muller算法 真不明白就找我

### 子弹散布概率密度型函数

double 子弹散布正态分布概率密度函数(double X, double Y,double 方差倒数) {//正态分布概率密度方差倒数 描述准星扩散大小 值越大越集中 (为什么不用方差 方差数小时 1/方差 误差会被放大)

double 概率 = 0;//子弹击中X,Y的概率

概率=(1 \* 方差倒数 \* 方差倒数) / (PI \* 2) \* exp(-((X \* X + Y \* Y) \* 方差倒数 \* 方差倒数) / 2);

return 概率;

}

正常二维正态分布为

其中为相关系数,若不为0会使散布变为椭圆,故设为0

为期望或者偏差,即平均值和众数位置,我们默认准星中心为{0,0},所以两者为0

为方差,或者可以认为是这个散布的范围,越小越好.我们设两者相等,散布会是圆形,若不相等会是椭圆型

方程简化为

double 子弹散布均匀分布概率密度(double X, double Y, double 边界) {//子弹散布累计概率密度 既子弹击中属于(-无穷,X)(-无穷,Y)大片区域的概率 现已经废弃

double 概率 = 0;//子弹击中X,Y的概率

if (X > (-1 \* 边界) && X<边界 && Y>(-1 \* 边界) && Y < 边界){

概率 = 1 / (边界 \* 边界);

}

return 概率;

}

方框之内概率相同 为1/(边界^2)

方框外概率为0 估计用不到这个函数了

### 子弹散布累计概率型函数

double 子弹散布正态分布累计概率密度(double X, double Y,double 方差倒数) {//子弹散布累计概率密度 既子弹击中属于(-无穷,X)(-无穷,Y)大片区域的概率

double 概率 = 0;//子弹击中X,Y的概率

概率 = (erf(X \* 方差倒数 / sqrt(2)) + 1) / 2 \* (erf(Y \* 方差倒数 / sqrt(2)) + 1) / 2;//我是sb 这么简单的函数找了好久没找到还想自己写

return 概率;

}

首先c++math库函数提供了erf(x)

我们对方框积分可以变为

我们将简化为便于表达

我们这个函数表达了的值 求积分只需调用方式如下

区域内概率总和(X\_1,X\_2,Y\_1,Y\_2)=正态分布累计概率(X\_2,Y\_2)- 正态分布累计概率(X\_1,Y\_2)- 正态分布累计概率密度(X\_2,Y\_1)+ 正态分布累计概率密度(X\_1,Y\_1)

double 子弹散布均匀分布累计概率密度(double X, double Y ,double 边界) {//子弹散布累计概率密度 既子弹击中属于(-无穷,X)(-无穷,Y)大片区域的概率 现已经废弃

double 概率 = 0;//子弹击中X,Y的概率

if (X > 边界 && Y > 边界) {

概率 = 1;

}

else if (Y > 边界&& X > -1 \* 边界) {

概率 = (X + 边界) / (2 \* 边界);

}

else if (X > 边界 && Y > -1 \* 边界) {

概率 = (Y + 边界) / (2 \* 边界);

}

else if (X > -1 \* 边界 && Y > -1 \* 边界) {

概率 = (X + 边界) \* (Y + 边界) / (4 \* 边界);;

}

else {

概率 = 0;

}

return 概率;

}

这个函数计算以下面积

这个没必要用 也没必要说 真不懂那就问吧

## 武器与目标的一些运动函数(除衰减外以被序列模式替代 但没删除 外一以后用)

屏幕偏差 p2020后坐力函数(double time) {//现在已经被拐点序列代替

屏幕偏差 临时变量;

临时变量.X\_屏幕 = time;

临时变量.Y\_屏幕 = time;

return 临时变量;

}

这几个函数都挺简单的 没必要解释吧

说实话我想用lambda表达式 但没啥可读性 所以放弃了

double p2020子弹衰减(double 目标位置Z) {//子弹衰减函数 (瞎设的数值)

double 衰减倍率;

if (目标位置Z <= 20) {//如果与目标距离小于20伤害不变

衰减倍率 = 1;

}

else if (目标位置Z <= 50) {//如果与目标距离大于20 小于50伤害乘0.7

衰减倍率 = 0.7;

}

else {//如果与目标距离大于50伤害乘0.5

衰减倍率 = 0.5;

}

return 衰减倍率;

}

目标中心实际位置 目标移动(double 时间, 目标中心实际位置 目标a) {//目标移动函数 现已经废弃 被拐点序列替代

目标a.X\_实际 = 目标a.X\_实际 + 时间;

目标a.Y\_实际 = 目标a.Y\_实际 + 时间 / 5;

目标a.Z\_实际 = 目标a.Z\_实际 + 时间;

return 目标a;

}

目标大小 目标大小变动(double 时间) {//现已经废弃 被拐点序列替代

目标大小 临时变量;

临时变量.X\_实际 = 10;

临时变量.Y\_实际 = 20;

return 临时变量;

}

## 压枪函数

屏幕拐点序列 压枪函数(玩家技术 玩家技术实体,屏幕拐点序列 后坐力拐点) {//详细解释见文档

屏幕拐点序列 临时压枪拐点序列;

int 开头指针;

int 间隔指针=1;

double 压枪结束到拐点速度X\_屏幕;

double 压枪结束到拐点速度Y\_屏幕;

double 压枪结束到拐点时间间隔;

double 速度误差;

double 方向误差;

double 时间误差;

for (开头指针 = 0; 开头指针 < 100; 开头指针++) {//遍历后坐力序列

if (后坐力拐点.开火时间\_T[开头指针 + 间隔指针] == 0) {//后坐力序列到达尽头 强制结束

break;

}

if (后坐力拐点.开火时间\_T[开头指针 + 间隔指针] > 临时压枪拐点序列.开火时间\_T[开头指针]) {

压枪结束到拐点时间间隔 = 后坐力拐点.开火时间\_T[开头指针 + 间隔指针] - 临时压枪拐点序列.开火时间\_T[开头指针];

压枪结束到拐点速度X\_屏幕 = (后坐力拐点.位置\_X\_屏幕[开头指针 + 间隔指针] - 临时压枪拐点序列.位置\_X\_屏幕[开头指针]) / 压枪结束到拐点时间间隔;

压枪结束到拐点速度Y\_屏幕 = (后坐力拐点.位置\_Y\_屏幕[开头指针 + 间隔指针] - 临时压枪拐点序列.位置\_Y\_屏幕[开头指针]) / 压枪结束到拐点时间间隔;

这个分隔符是我打了一列-----再摁回车自动生成的

压枪结束指一个阶段的结束,不是整个序列的结束

压枪结束到拐点是求下一阶段理想压枪方向和速度,解释如下,其中黑色是后座,蓝色是压枪.

后坐力转折点n+1

位置x[n+1],y[n+1] 时间t[n+1]

压枪转折点i 位置x[i],y[i] 时间t[i]

后坐力转折点n 位置x[n],y[n] 时间t[n]

压枪结束到拐点速度

时间差=后坐力.t[n+1]- 压枪t[i]

(后坐力.x[n+1]- 压枪x[i] )/时间差y同理

速度误差 = 正态分布随机数(1 / 玩家技术实体.压枪速度熟练程度, 0);

方向误差 = 正态分布随机数(1 / 玩家技术实体.压枪方向熟练程度, 0);

时间误差 = 正态分布随机数(1 / 玩家技术实体.压枪时间熟练程度, 0);

临时压枪拐点序列.速度\_X\_屏幕[开头指针] = (压枪结束到拐点速度X\_屏幕 \* cos(PI \* 方向误差) - 压枪结束到拐点速度Y\_屏幕 \* sin(PI \* 方向误差)) \* (1 + 速度误差);

临时压枪拐点序列.速度\_Y\_屏幕[开头指针] = (压枪结束到拐点速度Y\_屏幕 \* cos(PI \* 方向误差) + 压枪结束到拐点速度X\_屏幕 \* sin(PI \* 方向误差)) \* (1 + 速度误差);

实际压枪速度=理想压枪速度\*速度误差\*方向误差(一个旋转矩阵)

方向误差影响角度

速度误差影响长度

计算 a为速度误差 x为方向误差

临时压枪拐点序列.位置\_X\_屏幕[开头指针 + 1] = 临时压枪拐点序列.位置\_X\_屏幕[开头指针] + 临时压枪拐点序列.速度\_X\_屏幕[开头指针] \* (压枪结束到拐点时间间隔 \* (1 + 时间误差));

临时压枪拐点序列.位置\_Y\_屏幕[开头指针 + 1] = 临时压枪拐点序列.位置\_Y\_屏幕[开头指针] + 临时压枪拐点序列.速度\_Y\_屏幕[开头指针] \* (压枪结束到拐点时间间隔 \* (1 + 时间误差));

临时压枪拐点序列.开火时间\_T[开头指针 + 1] = 临时压枪拐点序列.开火时间\_T[开头指针] + (压枪结束到拐点时间间隔 \* (1 + 时间误差));

实际压枪位置=实际压枪速度\*时间误差+上一个实际压枪位置

临时压枪拐点序列.加速度\_X\_屏幕[开头指针] = 0;

临时压枪拐点序列.加速度\_Y\_屏幕[开头指针] = 0;

}

else {

间隔指针 ++;

开头指针 --;

这个if语句的作用 如下图

后坐力转折点i+n+1 时间t+0.1

压枪转折点i 时间 t+0.5

后坐力转折点i+n 时间 t

显然后坐力转折点i+n+1被略过了我们不能回去让t+0.1-t-0.5,那么我们略过该点,去与i+n+2做运算,那么判断略过某些转折点就使间隔指针n加一,i的位置每循环结束就加一,我们先减一一遍,那么循环结束一次i的位置没变n加一了

}

}

return 临时压枪拐点序列;

};

## 跟枪函数

屏幕拐点序列 跟枪函数(屏幕拐点序列 后坐力拐点,屏幕拐点序列 压枪拐点,实际拐点序列 目标移动, 玩家技术 玩家技术实体) {//详细解释见文档

屏幕拐点序列 跟枪序列;

double 前速度累计量\_X屏幕=0;

double 前速度累计量\_Y屏幕=0;

屏幕偏差 临时偏差;

屏幕偏差 临时速度偏差;

double 速度误差;

double 方向误差;

double 一阶跟枪类型能力;

double 二阶跟枪类型能力;

int 数组指针;

屏幕偏差 后坐力屏幕偏差;

屏幕偏差 压枪屏幕偏差;

屏幕偏差 目标运动屏幕偏差;

屏幕偏差 后坐力屏幕速度偏差;

屏幕偏差 压枪屏幕速度偏差;

屏幕偏差 目标运动屏幕速度偏差;

跟枪序列.开火时间\_T[1] = 玩家技术实体.反应力;

if (玩家技术实体.跟枪等级 < 1) {

一阶跟枪类型能力 = 玩家技术实体.跟枪等级;

二阶跟枪类型能力 = 0;

}

else {

一阶跟枪类型能力 = 1;

二阶跟枪类型能力 = 玩家技术实体.跟枪等级;//玩家技术实体.跟枪等级最大为2

}

这里的二阶没有做 绝大部分只有很少的情况有二阶跟枪

如目标加速度下坠 目标做载具等无法快速变向的情况适合二阶

对于一般对枪左右转向瞬间其加速度非常大 如果这时候进行二阶预判 跟枪效果会比一阶差

一阶跟枪解释:蓝色为零阶跟枪,只会跟踪到看到的位置,不管目标的移动

绿色为目标移动速度,红色为一阶跟枪,结合目标速度,在一个跟枪小阶段结束时更接近目标

绿色目标速度向量与红蓝起点无关,是目标自己的属性,不与准星偏差大小有关

for (数组指针 = 1; 数组指针 < 99; 数组指针++) {//1阶跟枪

速度误差 = 正态分布随机数(1 / 玩家技术实体.跟枪速度熟练程度, 0);

方向误差 = 正态分布随机数(1 / 玩家技术实体.跟枪方向熟练程度, 0);

后坐力屏幕偏差 = 屏幕拐点型\_转屏幕偏差函数(后坐力拐点, 数组指针 \* 玩家技术实体.反应力);

压枪屏幕偏差 = 屏幕拐点型\_转屏幕偏差函数(压枪拐点, 数组指针 \* 玩家技术实体.反应力);

目标运动屏幕偏差 = 实际拐点型\_转屏幕偏差函数(目标移动, 数组指针 \* 玩家技术实体.反应力);

计算在 i\*反应力 时刻 后坐力屏幕偏差 压枪屏幕偏差 目标运动屏幕偏差

临时偏差.X\_屏幕 = 压枪屏幕偏差.X\_屏幕 - 后坐力屏幕偏差.X\_屏幕 + 目标运动屏幕偏差.X\_屏幕 - 跟枪序列.位置\_X\_屏幕[数组指针];

临时偏差.Y\_屏幕 = 压枪屏幕偏差.Y\_屏幕 - 后坐力屏幕偏差.Y\_屏幕 + 目标运动屏幕偏差.Y\_屏幕 - 跟枪序列.位置\_Y\_屏幕[数组指针];

其中黑色为后座偏差,紫色为之前压枪结束一小段的偏差 ,红色为压枪偏差, 蓝色为反方向压枪偏差, 橙色为目标偏差, 绿色为理想压枪方向

黑色+紫色+蓝色+绿色=橙色

绿色=橙色-黑色-蓝色-紫色=橙色-黑色+红色-紫色

后坐力屏幕速度偏差 = 屏幕拐点型\_转屏幕速度偏差函数(后坐力拐点, 数组指针 \* 玩家技术实体.反应力);

压枪屏幕速度偏差 = 屏幕拐点型\_转屏幕速度偏差函数(压枪拐点, 数组指针 \* 玩家技术实体.反应力);

目标运动屏幕速度偏差 = 实际拐点型\_转屏幕速度偏差函数(目标移动, 数组指针 \* 玩家技术实体.反应力);

临时速度偏差.X\_屏幕 = 压枪屏幕速度偏差.X\_屏幕 - 后坐力屏幕速度偏差.X\_屏幕 + 目标运动屏幕速度偏差.X\_屏幕;

临时速度偏差.Y\_屏幕 = 压枪屏幕速度偏差.Y\_屏幕 - 后坐力屏幕速度偏差.Y\_屏幕 + 目标运动屏幕速度偏差.Y\_屏幕;

其中黑色为后座速度偏差, ,红色为压枪速度偏差, 蓝色为反方向压枪速度偏差, 橙色为目标速度偏差, 绿色为理想压枪一致速度

黑色+蓝色+绿色=橙色

绿色=橙色-黑色-蓝色=橙色-黑色+红色

跟枪序列.速度\_X\_屏幕[数组指针] = ((临时偏差.X\_屏幕 / 玩家技术实体.反应力 + 临时速度偏差.X\_屏幕 \* 一阶跟枪类型能力) \* cos(PI \* 方向误差) - (临时偏差.Y\_屏幕 / 玩家技术实体.反应力 + 临时速度偏差.Y\_屏幕 \* 一阶跟枪类型能力) \* sin(PI \* 方向误差)) \* (1 + 速度误差);

跟枪序列.速度\_Y\_屏幕[数组指针] = ((临时偏差.Y\_屏幕 / 玩家技术实体.反应力 + 临时速度偏差.Y\_屏幕 \* 一阶跟枪类型能力) \* cos(PI \* 方向误差) + (临时偏差.X\_屏幕 / 玩家技术实体.反应力 + 临时速度偏差.X\_屏幕 \* 一阶跟枪类型能力) \* sin(PI \* 方向误差)) \* (1 + 速度误差);

理想速度=临时屏幕偏差/反应力+临时速度偏差

实际速度=理想速度\*方向误差\*速度误差(这个解释见压枪序列)

跟枪序列.加速度\_X\_屏幕[数组指针] = 0;//先不急写这个

跟枪序列.加速度\_Y\_屏幕[数组指针] = 0;

跟枪序列.开火时间\_T[数组指针 + 1] = (数组指针 + 1) \* 玩家技术实体.反应力;

跟枪序列.位置\_X\_屏幕[数组指针 + 1] = 跟枪序列.位置\_X\_屏幕[数组指针]+跟枪序列.速度\_X\_屏幕[数组指针] \* (玩家技术实体.反应力) + (跟枪序列.加速度\_X\_屏幕[数组指针] / 2) \* (玩家技术实体.反应力) \* (玩家技术实体.反应力);

跟枪序列.位置\_Y\_屏幕[数组指针 + 1] = 跟枪序列.位置\_Y\_屏幕[数组指针]+跟枪序列.速度\_Y\_屏幕[数组指针] \* (玩家技术实体.反应力) + (跟枪序列.加速度\_Y\_屏幕[数组指针] / 2) \* (玩家技术实体.反应力) \* (玩家技术实体.反应力);

得到速度将其积分得到位置 解释见位置序列转速度

}

return 跟枪序列;

};

## 实际位置函数

### 使用拐点的复杂模型

边框实际位置 实际位置函数(屏幕拐点序列 后坐力拐点, 屏幕拐点序列 压枪拐点, 实际拐点序列 目标移动, 屏幕拐点序列 跟枪拐点, 目标大小序列 目标大小实体,玩家技术 玩家技术实体, double 时间变量) {//这个是重载函数 就是名字一样但参数不一样的函数叫 重载

边框实际位置 临时变量={0,0,0,0};//

目标大小 目标大小T时刻位置={10,10};

int 数组指针 = 0;

while (目标大小实体.变形时间T[数组指针+1] <= 时间变量) {

数组指针++;

if (数组指针 >= 100) {

break;

}

}

屏幕偏差 后坐力偏差 = 屏幕拐点型\_转屏幕偏差函数(后坐力拐点, 时间变量);

屏幕偏差 压枪偏差 = 屏幕拐点型\_转屏幕偏差函数(压枪拐点, 时间变量);

目标中心实际位置 目标移动位置 = 实际拐点型\_转实际位置函数(目标移动, 时间变量);

屏幕偏差 跟枪偏差 = 屏幕拐点型\_转屏幕偏差函数(跟枪拐点, 时间变量);

if (目标移动位置.Z\_实际 == 0) {//目标与玩家z轴位置为0 即完全贴合 这是不可能的 报错并返回

return 临时变量;

}

目标大小T时刻位置.X\_实际 = 目标大小实体.X\_实际[数组指针];

目标大小T时刻位置.Y\_实际 = 目标大小实体.Y\_实际[数组指针];

临时变量.左下角\_X\_屏幕 = (目标移动位置.X\_实际 - 目标大小T时刻位置.X\_实际 / 2) \* 游戏距离到屏幕距离放缩系数 / (目标移动位置.Z\_实际) - 后坐力偏差.X\_屏幕 + 压枪偏差.X\_屏幕- 跟枪偏差.X\_屏幕;

临时变量.左下角\_Y\_屏幕 = (目标移动位置.Y\_实际 - 目标大小T时刻位置.Y\_实际 / 2) \* 游戏距离到屏幕距离放缩系数 / (目标移动位置.Z\_实际) - 后坐力偏差.Y\_屏幕 + 压枪偏差.Y\_屏幕- 跟枪偏差.Y\_屏幕;

临时变量.右上角\_X\_屏幕 = (目标移动位置.X\_实际 + 目标大小T时刻位置.X\_实际 / 2) \* 游戏距离到屏幕距离放缩系数 / (目标移动位置.Z\_实际) - 后坐力偏差.X\_屏幕 + 压枪偏差.X\_屏幕- 跟枪偏差.X\_屏幕;

临时变量.右上角\_Y\_屏幕 = (目标移动位置.Y\_实际 + 目标大小T时刻位置.Y\_实际 / 2) \* 游戏距离到屏幕距离放缩系数 / (目标移动位置.Z\_实际) - 后坐力偏差.Y\_屏幕 + 压枪偏差.Y\_屏幕- 跟枪偏差.Y\_屏幕;

return 临时变量;

};

解释起始可同上解释

### 未使用拐点的无压枪跟枪简单模型

边框实际位置 实际位置函数(目标中心实际位置 目标起始位置, 目标移动函数类型 目标移动函数指针, 目标大小函数类型 目标大小函数指针, 后坐力函数类型 后坐力函数指针, double 时间) {//老函数

边框实际位置 临时变量;

目标大小 目标大小TIME时刻位置 = 目标大小函数指针(时间);

目标中心实际位置 目标移动位置 = 目标移动函数指针(时间, 目标起始位置);

屏幕偏差 偏差变量 = 后坐力函数指针(时间);

临时变量.左下角\_X\_屏幕 = (目标移动位置.X\_实际 - 目标大小TIME时刻位置.X\_实际 / 2) \* 游戏距离到屏幕距离放缩系数 / (目标移动位置.Z\_实际) - 偏差变量.X\_屏幕;

临时变量.左下角\_Y\_屏幕 = (目标移动位置.Y\_实际 - 目标大小TIME时刻位置.Y\_实际 / 2) \* 游戏距离到屏幕距离放缩系数 / (目标移动位置.Z\_实际) - 偏差变量.Y\_屏幕;

临时变量.右上角\_X\_屏幕 = (目标移动位置.X\_实际 + 目标大小TIME时刻位置.X\_实际 / 2) \* 游戏距离到屏幕距离放缩系数 / (目标移动位置.Z\_实际) - 偏差变量.X\_屏幕;

临时变量.右上角\_Y\_屏幕 = (目标移动位置.Y\_实际 + 目标大小TIME时刻位置.Y\_实际 / 2) \* 游戏距离到屏幕距离放缩系数 / (目标移动位置.Z\_实际) - 偏差变量.Y\_屏幕;

return 临时变量;

};

## 最后计算模型

double 简单模型累计概率形(枪械 枪械实体, double (累计概率函数指针)(double, double, double)) {//0.8版本的简单模型

double 临时伤害总和变量 = 0;

double 时间变量 = 0;

屏幕偏差 后坐力偏差临时变量;

int 循环变量;

for (循环变量 = 0; 循环变量 < 1000; 循环变量++) {

时间变量 = (double)循环变量 / 1000;

后坐力偏差临时变量=p2020后坐力函数(时间变量);//后坐力函数最后会进人枪械集合 作为其中一个元素

临时伤害总和变量 = 临时伤害总和变量 + (累计概率函数指针(后坐力偏差临时变量.X\_屏幕 + 0.5, 后坐力偏差临时变量.Y\_屏幕 + 0.5, 1) - 累计概率函数指针(后坐力偏差临时变量.X\_屏幕 + 0.5, 后坐力偏差临时变量.Y\_屏幕 - 0.5, 1) - 累计概率函数指针(后坐力偏差临时变量.X\_屏幕 - 0.5, 后坐力偏差临时变量.Y\_屏幕 + 0.5, 1) + 累计概率函数指针(后坐力偏差临时变量.X\_屏幕 - 0.5, 后坐力偏差临时变量.Y\_屏幕 - 0.5, 1))/1000;

printf("总伤害 %lf 准星X %lf 准星Y %lf 时间 %lf\n", 临时伤害总和变量, 后坐力偏差临时变量.X\_屏幕, 后坐力偏差临时变量.Y\_屏幕, 时间变量);

}

临时伤害总和变量 = 临时伤害总和变量 \* 枪械实体.武器伤害 \* 枪械实体.武器射速;

return 临时伤害总和变量;

};

double 简单模型概率密度形(枪械 枪械实体, double (概率密度函数指针)(double, double, double)) {

double 临时伤害总和变量 = 0;

double 时间变量 = 0;

double X坐标变量 = 0;

double Y坐标变量 = 0;

double 概率积分临时变量;

屏幕偏差 后坐力偏差临时变量;

int 循环时间变量dt;

int 循环X坐标变量dx;

int 循环Y坐标变量dy;

for (循环时间变量dt = 0; 循环时间变量dt < 1000; 循环时间变量dt++) {

时间变量 = (double)循环时间变量dt / 1000;

后坐力偏差临时变量 = p2020后坐力函数(时间变量);//后坐力函数最后会进人枪械集合 作为其中一个元素

概率积分临时变量 = 0;

for (循环Y坐标变量dy = 0; 循环Y坐标变量dy < 100; 循环Y坐标变量dy++) {

for (循环X坐标变量dx = 0; 循环X坐标变量dx < 100; 循环X坐标变量dx++) {

概率积分临时变量 = 概率积分临时变量 + 概率密度函数指针(后坐力偏差临时变量.X\_屏幕 - 0.5 + ((double)循环X坐标变量dx / 100), 后坐力偏差临时变量.Y\_屏幕 - 0.5 + ((double)循环Y坐标变量dy / 100), 1) / 10000;

}

}

临时伤害总和变量 = 临时伤害总和变量 + 概率积分临时变量 / 1000;

printf("总伤害 %lf 准星X %lf 准星Y %lf 时间 %lf\n", 临时伤害总和变量, 后坐力偏差临时变量.X\_屏幕, 后坐力偏差临时变量.Y\_屏幕, 时间变量);

}

临时伤害总和变量 = 临时伤害总和变量 \* 枪械实体.武器伤害 \* 枪械实体.武器射速;

return 临时伤害总和变量;

};

double 一般模型累计概率形(枪械 枪械实体, 目标总体 目标实体,int 运算精度) {//无压枪 目标无位移

double 临时伤害总和变量 = 0;

double 时间变量 = 0;

double 命中概率 = 0;

int 循环变量;

目标中心实际位置 临时目标位置变量 = 目标实体.目标游戏内起始位置;

for (循环变量 = 0; 循环变量 < 枪械实体.耗光时间 \* 运算精度; 循环变量++) {

时间变量 = (double)循环变量 / (double)运算精度;

临时目标位置变量 = 目标实体.目标游戏内移动函数指针(时间变量, 目标实体.目标游戏内起始位置);

目标实体.目标在屏幕位置 = 实际位置函数(目标实体.目标游戏内起始位置, 目标实体.目标游戏内移动函数指针, 目标实体.目标游戏内大小函数指针,枪械实体.后坐力函数指针, 时间变量);

命中概率 = 枪械实体.子弹概率累计函数指针(目标实体.目标在屏幕位置.右上角\_X\_屏幕, 目标实体.目标在屏幕位置.右上角\_Y\_屏幕, 枪械实体.子弹散布参数) - 枪械实体.子弹概率累计函数指针(目标实体.目标在屏幕位置.右上角\_X\_屏幕, 目标实体.目标在屏幕位置.左下角\_Y\_屏幕, 枪械实体.子弹散布参数) - 枪械实体.子弹概率累计函数指针(目标实体.目标在屏幕位置.左下角\_X\_屏幕, 目标实体.目标在屏幕位置.右上角\_Y\_屏幕, 枪械实体.子弹散布参数) + 枪械实体.子弹概率累计函数指针(目标实体.目标在屏幕位置.左下角\_X\_屏幕, 目标实体.目标在屏幕位置.左下角\_Y\_屏幕, 枪械实体.子弹散布参数);

临时伤害总和变量 = 临时伤害总和变量 + 命中概率 \* 枪械实体.武器伤害 \* 枪械实体.武器射速 \* 枪械实体.子弹衰减函数指针(临时目标位置变量.Z\_实际)/ (double)运算精度;

printf("总伤害 %lf 右上角X\_屏幕 %lf 右上角Y\_屏幕 %lf 左下角X\_屏幕 %lf 左下角Y\_屏幕 %lf 时间 %lf\n", 临时伤害总和变量, 目标实体.目标在屏幕位置.右上角\_X\_屏幕, 目标实体.目标在屏幕位置.右上角\_Y\_屏幕, 目标实体.目标在屏幕位置.左下角\_X\_屏幕, 目标实体.目标在屏幕位置.左下角\_Y\_屏幕, 时间变量);

}

return 临时伤害总和变量;

};

double 复杂模型累计概率形(枪械 枪械实体, 目标总体 目标实体,玩家技术 玩家实体, int 运算精度) {//无压枪 目标无位移

double 临时伤害总和变量 = 0;

double 时间变量 = 0;

double 命中概率 = 0;

int 循环变量;

屏幕拐点序列 后座序列 = 枪械实体.枪械子弹序列;;

屏幕拐点序列 压枪序列;

实际拐点序列 移动序列 = 目标实体.移动序列;;

屏幕偏差 移动屏幕序列;

屏幕拐点序列 跟枪序列;

边框实际位置 目标边框位置;

后座序列 = 屏幕拐点位置\_转速度函数(后座序列);

压枪序列 = 压枪函数(玩家实体, 后座序列);

移动序列 = 实际拐点位置\_转速度函数(移动序列);

跟枪序列 = 跟枪函数(后座序列, 压枪序列, 移动序列, 玩家实体);

for (循环变量 = 0; 循环变量 < 枪械实体.耗光时间 \* 运算精度; 循环变量++) {

时间变量 = (double)循环变量 / (double)运算精度;

目标边框位置 = 实际位置函数(后座序列, 压枪序列, 移动序列, 跟枪序列, 目标实体.目标游戏内大小序列, 玩家实体, 时间变量);

命中概率= 枪械实体.子弹概率累计函数指针(目标边框位置.右上角\_X\_屏幕, 目标边框位置.右上角\_Y\_屏幕, 枪械实体.子弹散布参数) - 枪械实体.子弹概率累计函数指针(目标边框位置.右上角\_X\_屏幕, 目标边框位置.左下角\_Y\_屏幕, 枪械实体.子弹散布参数) - 枪械实体.子弹概率累计函数指针(目标边框位置.左下角\_X\_屏幕, 目标边框位置.右上角\_Y\_屏幕, 枪械实体.子弹散布参数) + 枪械实体.子弹概率累计函数指针(目标边框位置.左下角\_X\_屏幕, 目标边框位置.左下角\_Y\_屏幕, 枪械实体.子弹散布参数);

临时伤害总和变量 = 临时伤害总和变量 + 命中概率 \* 枪械实体.武器伤害 \* 枪械实体.武器射速 \* 枪械实体.子弹衰减函数指针(实际拐点型\_转实际位置函数(移动序列,时间变量).Z\_实际) / (double)运算精度;

printf("总伤害 %lf 右上角X\_屏幕 %lf 右上角Y\_屏幕 %lf 左下角X\_屏幕 %lf 左下角Y\_屏幕 %lf 时间 %lf\n", 临时伤害总和变量, 目标边框位置.右上角\_X\_屏幕, 目标边框位置.右上角\_Y\_屏幕, 目标边框位置.左下角\_X\_屏幕, 目标边框位置.左下角\_Y\_屏幕, 时间变量);

}

return 临时伤害总和变量;

};