

Celeste 机制详解

反向钟, 865466388@qq.com

写于 2022 年 11 月 1 日至 2024 年 10 月 14 日

目录

前言	7
I 源码解读	9
1 一帧中的运算顺序	9
2 基础运动与信息面板	12
2.1 时间	12
2.2 状态	12
2.3 空间	13
2.4 速度	14
2.5 加速度	14
2.6 碰撞箱与伤害箱	14
2.7 操作	14
3 StNormal 中的一般运动	15
3.1 水平运动	15
3.2 竖直运动	15
4 跳跃	16
5 抓墙	18
5.1 爬墙	18
5.2 抓跳	18
5.3 Wallboost	18
5.4 体力	18
5.5 SlipCheck	19
5.6 cb	19
5.7 Climbhop	19
6 冲刺	20
6.1 Ultra 机制	22
6.2 各类取消冲刺的方法	22
7 物理更新与固块碰撞	24
7.1 MoveH	24
7.2 OnCollideH / OnCollideV	24
7.3 Retention	24
8 实体碰撞	26

8.1	实际碰撞箱/实际伤害箱	26
8.2	各类弹跳	26
9	移动块 / 移动单向板	32
9.1	骑乘	32
9.2	穿墙	32
9.3	Solid	33
9.4	OnSquish	34
9.5	JumpThru	38
9.6	Koral clip	40
10	LiftBoost	42
10.1	移动块自身获得 LiftSpeed	42
10.2	Player 从移动块获得 LiftSpeed	42
10.3	生效: Player 从 LiftBoost 获得速度加成	42
11	蹲姿	45
12	AutoJump, forceMoveX	48
13	Holdable	50
13.1	平 u 接水母并瞬间丢弃水母	50
13.2	另一种 NormalUpdate() 中的 Drop()	51
13.3	Offgrid Holdable	51
13.4	水母	53
14	运算顺序 Order of Operation	54
14.1	深度机制	54
14.2	EntityList.UpdateLists()	54
14.3	深度	55
14.4	风与玩家的更新顺序	55
14.5	使用 TAS Helper 来研究运算顺序	55
15	Fling Bird	56
15.1	机制	56
15.2	StFlingBird 的时长, 水平位移, 竖直位移	57
15.3	补充	58
16	一个一个 State 全写下来...	59
16.1	Coroutine, StateMachine	59
16.2	各 State 一览	60

17 拾遗	62
17.1 冻结帧	62
17.2 Cassette	62
17.3 切版	62
17.4 各类方块被触发	63
17.5 BadelineBoost	64
17.6 Puffer	64
17.7 JumpThru	65
17.8 Bubble Corner Glide	65
17.9 Cutscene	66
17.10望远镜, TalkComponent	66
17.11按钮	68
17.12WindMove	69
17.13云	69
17.14Kevin 块	70
17.15果冻双跳	71
17.16Wall Slide	71
17.17钥匙机制	71
17.18岩浆块	75
17.19漂浮块	75
17.20Collider 相互碰撞的检测机制	78
17.21镜头	78
17.22整数坐标不再是整数 (offgrid)	79
17.23LastAim	80
17.24idu 和 didu	80
17.25Spinner Cycle	81
17.26(位置) 浮点数操纵	84
17.27浮点数	84
17.28暂停	87
17.29Engine, Scene, Level, Session	88
17.30地图	90
17.31进入房间的具体机制	91
17.32切板卡死	93
17.33随机数	94
17.34CelesteTAS, Celeste Studio 相关的运算顺序	96
17.35你可能不知道的一些 TAS mod Feature	96
17.36杂项中的杂项	96
18 挑战/异变与 Mod	101

18.1 超冲	101
18.2 部分挑战/异变 (超冲, 无抓, 打嗝等) 相关	102
18.3 Ultra Jump 异变	102
18.4 部分 mod 实体的特性	102
18.5 Portaline	102
18.6 Ceiling Ultra	103
19 还未处理的一些原始材料	104
19.1 Cassette Block	104
19.2 XNA/FNA desync	104
19.3 一些申必应用	104
19.4 游戏崩溃与异常	107
20 Code mod 相关	107
21 习题集	107
22 其他常用工具	109
II 实际应用	
1 基础	110
1.1 干净利落地爬上平台	110
1.2 咖啡跳	110
1.3 dd 穿刺	110
1.4 Coyote Hyper	110
1.5 gu / gultra / grounded ultra	110
1.6 DD 咖啡	111
1.7 移动块多 cb	111
1.8 rcb	111
1.9 水母 u, cutscene u	111
1.10 du	111
1.11 落地同时保留向下的速度 / 下降蹲姿保留	111
2 进阶	111
2.1 亚像素调整	111
2.2 半体力 / 无体力	112
2.3 Cpop	112
2.4 踩刺	113
2.5 正向速度爬刺墙	113
2.6 大风爬刺墙	113

2.7	Corner glide	113
2.8	Overclock	113
2.9	Wallboost 急刹车	113
2.10	Wallboost 白嫖距离	113
2.11	grounded wallboost / delayed wallboost	113
2.12	idu	113
2.13	moon spring boost	114
2.14	gultra retention	114
2.15	core boost	114
2.16	Cross-screen cornerboosts	115
2.17	float rounding error	115

III 附录 116

1	学名-俗名/译名对照表	116
2	状态表	118
3	深度表	119
4	英文社区常见术语	122
5	FlingBird	123
6	常用数据表	123
6.1	快速重生	124
7	源码中常用的部分函数	124

IV 学习资料 125

前言

关于第 0 版 (2024.10.14)

因为本文长期停更, 因此选择暂且先发布. 不过, 由于没有修订, 你可能会看到部分章节有大量没有处理的原始材料, 或者有前后相冲突的话语, 又或者因为时间跨度大, 部分内容不适用于最新版 `Everest/CelesteTAS/Celeste Studio`.

关于本文

本文档原名 “My TAS Reference”. 这是……算是我的个人笔记一样的东西, 基本是从我刚入坑 TAS 就开始写起的, 目标是写成百科全书一样. 限于书写时间跨度大, 个人 TAS 经验不足, 且编程水平也不足, 难以保证完全的正确性. 还请读者谨慎吸收.

这不是教程, 因此按照自己的需要跳到对应的章节阅读即可.

约定

在本文中,

- 模组: 默认讨论只安装了最新版本的 `Everest` 与 `CelesteTAS`, 且不开启任何异变/协助下, 游戏的行为.

按照惯例, 在制作与播放 TAS 时我们应当只使用 `Everest` 与 `CelesteTAS`, 以及 mod 图所必须的前置 mod 的最新版本. 大部分常见 mod 内容并不迥异于原版内容, 熟知原版内容即可触类旁通. 而少部分常见却又奇怪的 mod 内容, 会有专门的篇章进行探讨. 因此我们默认只安装了 `Everest` 与 `CelesteTAS`.

为了视觉效果, 部分插图额外使用了 `TAS Helper` (及其前置 `SpeedrunTool`). 尽管它们理应不影响正常情况下的游戏行为, 我们还是要谨慎使用这些 mod.

对于机制的解读依赖于对 `Celeste.exe` 的反编译, 为方便起见我们使用的是 `Everest` 补丁过的版本而非原版. `Everest` 本身是在几乎不影响原版的情况下进行了拓展, 例如 `Everest` 在部分地方进行了性能优化, 例如用 `LengthSquared() < 576f` 代替 `Length() < 24f`. 在部分地方加入了新特性, 例如给 `Decal` 加入了 `Color` 字段并予以相关支持. 若要仔细去区分机制是否是原版的而非 `Everest` 的, 意义并不大.

尽管如此, 依然有游戏内容在非常微妙的地方发生了变动. 例如 `Everest` 对 `FNA/XNA desync` 的修复, 这实际上影响了原版的表现. 更加明显的例子是 `Everest` 的切版防卡死的机制. `Everest` 还修复了许多可能在 mod 图甚至原版中产生 bug 的地方 (见 `Celeste.Mod.mm/Patches`), 有时会为此注入新的方法 (如 `Level.LoadNewPlayer`, 不少 mod 制作者甚至不知道这是 `Everest` 的补丁).

`CelesteTAS` 几乎不影响实际的游戏内容. 目前笔者唯一知道的是, `TAS.EverestInterop.DesyncFixer` 会将 `Debris` 的随机性在 TAS 播放时, 从完全的随机变为对于同一个 TAS 固定的随机. 这种改动完全是可以忽略的.

Everest 在不断更新, 比如原本的补丁 Level.LoadNewPlayer 已被 Level.LoadNewPlayerForLevel 取代. 而 CelesteTAS 尽管很少对游戏机制产生影响, 但作为制作 TAS 的工具的特性却也是在不断更新与增多. 因此, 本文谈论的内容难免有时效性的问题.

- 谈论对象/动作/机制等时, 优先采用中文的较为通用的且不易引起歧义的叫法 (中文俗名), 其次若本文有合适的译名则使用译名, 再否则用不易引起歧义的英文俗名, 最后用它在源码中的命名 (学名). 参考 [附录-表格 1](#).
- 涉及到 信息面板, Celeste Studio 等的地方, 指的都是通过 CelesteTAS 相关组件观察到的种种信息. 并没有严格区分这几者.
- Maddy 即 Madeline, 游戏的主角. 文中有时也以 玩家 或 Player 指代.
- 在谈论 速度时, 有时只谈论水平速度/竖直速度中的一者, 具体是哪个取决于语境.
- 在谈论水平方向时, 使用 正方向来指代左/右中的一者, 在具体语境中可以指运动的方向 (Speed.X 的正负性), 操作的方向 (moveX), 面朝的方向 (Facing), 或判定的方向等, 需根据语境判断.

尽管游戏本身自带的正方向是朝右, 但不至于混淆的情况下, 默认谈论我们关心的正方向上的行为. 例如朝左冲刺的情形下, 做出了向左的 Corner Boost, 则说此时的 (水平) 速度是 $240 + 40 = 280$ (而不是说 $(-240) + (-40) = (-280)$). 例如朝右上冲刺, 然后向左反抓打断, 则速度为 $169.7 - 40 = 129.7$.

正方向一词可以在语境中很快地改变. 例如上冲, 左侧有墙, 试图蹭墙跳. 因为蹭墙跳的判定范围是 5 px (正方向朝左), 在判定范围内, 所以成功触发蹭墙跳, 水平速度变为 170 (正方向朝右). 总之就是我们尽可能地省去不言自明的正负性.

- 在谈论竖直方向时, 以向下为正. 但如果不易引起误会, 谈论向上的位移/速度/加速度时我们也省去正负性.
- 在谈论一些倒计时类/布尔类的状态时, 直接用状态名表示倒计时 > 0 / 取值 (赋值) 为真. 如 NoControl, onGround.
- 物理量的单位: 时间 f (帧) 或 s (秒), 长度 px (像素), 速度 px/s, 加速度 px/s/f.

一些源码会用到浮点数, 写成 Xf 或 X . 源码中时间的 $1f = 1s$, 空间的 $1f = 1px$, 速度的 $1f = 1px/s$.

Part I

源码解读

1 一帧中的运算顺序

程序上的一帧, 是这样的:

- (0) 这一帧开始.
- (1) Maddy 开始主动更新.
- (2) 若 `Speed.Y ≥ 0` 且脚底下 1px 处是否是地面, 则 `onGround`.
- (3) 如果 `dashRefillCooldownTimer > 0`¹, 那么 `Timer -1`. 否则如果关卡是可以恢复冲刺的², 并且脚下是合适的地面³, 并且 Maddy 不与尖刺 (的碰撞箱) 重合⁴, 那么恢复冲刺次数.⁵
- (4) 其他一堆 `Timer` 也随之更新. 其中一些带有倒计时的操作/判定, 比如 `Wallboost`, `Wall Speed Retention` 等, 在这一步进行判定并改变 `Speed`.

此外还有若 `onGround` 则恢复体力⁶.

若 `onGround` 则狼跳帧重置为 6 帧.

如果 `forceMoveXTimer > 0` 则 `moveX = forceMoveX`, 否则 `moveX = Input.MoveX.Value`, 由玩家的输入决定. `forceMoveX` 和 `moveX` 的取值集合都是 $\{-1, 0, +1\}$.

执行 `climbhop` 的位移.

若 `moveX ≠ 0` 且 `InControl`, 且不在 `StClimb`, `StPickUp`, `StRedDash`, `StHitSquash`, 则 `Facing = moveX`. 否则继承之前的值. `Facing` 的取值集合是 $\{-1, +1\}$.

产生 `lastAim`, 以供冲刺使用. 基本上就是 `lastAim` 是基于 `Aim` 算出的只有八个方向的模长为 1 的向量 (为了手柄考虑, 在四个正方向上有一些容错. 在容错之外则是 `new Vector2(Math.Sign(value.X), Math.Sign(value.Y)).SafeNorm` 异变 -360° 冲刺 模式下, 则可以是任何方向向量). `Aim` 与 `Input.Feather` 几乎一样 (除了前者受 `DashOnly` 而后者受 `MoveOnly` 键影响).

¹这就是 `10,R,X; 1,R,J` 最早得在第 11 帧才能恢复充能的原因.

²即 `level.Session.Inventory.NoRefills = false`.

³即满足两种情形之一:

(a) 下移一格后, 与并非 `NegaBlock` 的 `Solid` 碰撞. (但很少有人用 `NegaBlock`.)

(b) 下移一格后, 与 `JumpThru` 碰撞, 当前不与 `JumpThru` 碰撞 (即正好脚踩 `JumpThru`, 而不能半身陷在 `JumpThru` 中).

⁴但是圆刺, 煤球等, 都是可以的.

⁵由于 `onGround`, 冲刺次数恢复, 状态机三者的运算顺序, 使得可以 Maddy 上一帧还未 `onGround` (但视觉上已经 `onGround`), 下一帧 Maddy 已经成功输入了冲刺.

⁶`StClimb` 的 `onGround` 恢复体力是在 `ClimbUpdate` 里的, 但是在最前面. 与这一步之间没什么别的涉及体力的操作, 所以等效为在这一步恢复体力. 除了在 `base.Update()` 前有个 `IsTired` 产生一个 `Input.Rumble(RumbleStrength.Light, RumbleLength.Short)`? 谁在乎呢.

(5) Base.update(). 主要是被 Add 进去的各个组件 (Component) 更新. 包括:

((1)) 状态机更新一次. 这大体上应该是:

- 若状态为 X , 则调用 Xupdate() 或 XCoroutine(), 获得返回值 Y 状态. 若 $Y \neq X$ 则调用 XEnd(), YBegin(), 进入状态 Y .

玩家的大部分输入都是在 Xupdate() 中起效的, 少部分是在与实体碰撞的时候起效的.

一次状态机更新中只会运行一次 update/coroutine, 这意味着状态的结束是 Xupdate()→XEnd()→YBegin(), 而 Yupdate() 要等到下一次状态机更新.

((2)) liftSpeedTimer 更新.

(6) 判定下落蹲姿取消等. 根据 Jumpthrough 等去移动位置. 如果冲刺方向水平且 DashAttacking 且向下 3px 有地面/单向板则向下修正.

(7) 物理更新 & 固块碰撞: 根据 Speed 去移动位置. 并使用红色碰撞箱进行固块碰撞, 这里包括墙角修正, 1.2 倍加速等. 先水平位移 + 碰撞, 再竖直位移 + 碰撞⁷.

(8) 判定游泳并强制更新状态.

(9) 更新手持物位置.

(10) 用碰撞箱 (而非伤害箱) 触发 trigger.

(11) 镜头移动, level.Camera.Position = position + (cameraTarget - position) * (1f - (float)Math.Pow(0.01f / num, Engine.DeltaTime)).

(12) 实体碰撞: 范围是所有具有组件 PlayerCollider 的实体. 固块并不具有 PlayerCollider. 发生碰撞时使用 PlayerCollider.OnCollide, 这会改变 Player 和实体自身的一些状态. 使用粉色伤害箱. 例如碰弹簧等. 主要是会改变 Maddy 的位置和速度. 有一些碰撞也会强制更新 Maddy 的状态. (例如碰到 FlingBird), 但 Maddy 的新状态自然在下一帧才会 update, 以改变 Maddy 的种种属性. 死亡也是在这一步发生.

这里如果产生了位置移动, 并不会调用 onCollideH/V.

(13) level.EnforceBounds(this): 即上下左右 bound, 如果可能的话会产生切版.

(14) 更新头发.

(15) Maddy 更新完毕.

(16) 更新其他实体. 这一步并不严格在 Maddy 之后, 但大多数情况都是如此. 包括:

- (a) WindController 更新. 它在更新时会对于具有 WindMover 组件的对象使用 WindMover 的 Move 动作, 对 Maddy 就是调用 WindMove(level.Wind * 0.1f * Engine.DeltaTime). 表现为被风移动位置.

⁷例如斜下冲电箱, 上一帧头顶正好与电箱下边缘齐平, 而水平还稍有一点点距离. 如果严格按照速度方向, 那么撞不到. 但如果先水平, 再竖直, 就会在水平位移的时候撞到电箱.

(b) 平台 (Platform)(包括 Solid, JumpThrough) 更新位置. 这包括移动块推动/挤压 Maddy(可能致死).

(c) 望远镜检测玩家是否意图使用它.

(d) 更新其他实体的速度, 位置等. (例如 Spikes, Spring, FlingBird)

这两者的顺序取决于 actualDepth. 越浅的越早更新.

少部分实体, 例如 TheoCrystal, Glider, 似乎会早于 Maddy 更新.

值得注意的是, 包括 Maddy 在内的实体的更新顺序不是一成不变的. 比如 4A 的风的更新晚于 Maddy, 但重试后 (这一面的) 风的更新就早于 Maddy. 而 7A 的上风的更新早于 Maddy, 但在 [g-02] 如果选择自己进入而非 BadelineBoost 抛上来, 也会导致风的更新晚于 Maddy. (据说是因为状态变为 StIntroJump, 导致 actualDepth 改变)

(17) Celeste Studio 显示此时的信息.

(18) 进入下一帧.

还有些琐碎的东西略.

关于游戏的主循环, 参考 [图 17.29](#).

2 基础运动与信息面板

2.1 时间

Celeste 锁定 60 帧运行. 严格来说现实时间 $1s \neq$ 游戏中 60 帧, 但做 TAS 并不关心这个. 每帧, 游戏计时器 (In-game Timer) 显示的数字增加 $0.017s$. 这与现实时间或 `Engine.DeltaTime` 都略有偏差. 但依然没人在乎.

各类计时器内部以浮点数计⁸, 每帧走单位时间 $Engine.DeltaTime = 0.0166667 \approx 1/60$ ⁹, 除以单位时间后就得到计时器对应的帧数. 当 `Timer ≤ 0` 时, 会触发某些事件/允许某些操作. Celeste Studio 显示的信息已经自动转换为了对应的帧数, 非常方便.

本文中并不固定使用一种时间单位. 鉴于文章性质, 我们需要大量的接触源码, 因此方便的做法是,

- 在解释机制时, 对各类计时器的值, 我们原封不动地将它以浮点数的形式写入本文, 写作 Xf .
- 在一些总结性的论述中, 使用 帧 作为单位, 但并不使用帧的符号 f , 以避免误解. 除非不至于产生歧义.
- 我们在游戏计时器以外几乎不会用到单位 秒 (s).

游戏计时器是在 `level.UpdateTime()` 处. 这里面

```
long ticks = TimeSpan.FromSeconds(Engine.RawDeltaTime).Ticks;
```

按定义展开, 首先 $num = 16.6667$, 然后 $num2 = 17.16667$. 被强转成 `long` 类型得到 $(long)num2 = 17$. 然后乘 10000 得到 17,0000 个嘀嗒. 而一秒是 10^7 嘀嗒. 于是最终就每帧给计时器增加 $0.017s$.

`level.TimerStarted` 应该控制了是否走时.

2.2 状态

Celeste Studio 显示的状态与源码里的状态基本是一致的, 除去 `CanDash` 定义略有不同.

信息面板上的状态分为两大类, 倒计时类和状态类.

倒计时类形如 $X(n)$, 表示之后的 n 帧都是 X 状态 (对于这一帧的情况则不做任何保证). 不显示 $X(0)$ (尽管这帧还在 X 状态). 例如 `NoControl (40)`, `Jump(11)`, `DashCD(11)`, `Frozen(3)`, `Coyote(4)`.

状态类形如 Y , 表示这一帧是否处于 Y 状态. 例如 `CanDash` 表示这帧冲刺冷却已结束, 且有冲刺次数, 可以在这帧输入冲刺. `StDash` 表示这帧处于冲刺状态.

因此对于平地 `ultra`

⁸这样做可能是为了保证若改变 异变-游戏速度, 大部分机制基本都还能正常工作.

⁹部分情形下, 应修正为

```
Engine.DeltaTime = Engine.RawDeltaTime * TimeRate * TimeRateB * GetTimeRateComponentMultiplier(scene).
```

这里 `Engine.RawDeltaTime = 0.0166667`. `TimeRate` 是游戏中主要的改变时间流速的要素, 如剧情, 如 `Oshiro/Seeker` 进攻时的时间减速等. `TimeRateB` 是原版的异变和帮助模式给出的修改. 而 `GetTimeRateComponentMultiplier(scene)` 则是在较新的版本里, `Everest` 提供给模组用于修改时间流速的接口, 不过许多模组并没有使用它.

对于日常情形, `Engine.DeltaTime = Engine.RawDeltaTime * TimeRate` 就够用了.

14	R, D, X
1	J
14	R, D, X
1	J

在第 1 帧, 我们见到 Frozen(3), DashCD(11), Ground¹⁰. 这一帧不是冻结帧.

在第 $2 \leq i \leq 4$ 帧期间 Frozen(4-i), DashCD(11), Ground. 其中 Frozen(0) 不显示. 它们是冻结帧. 在冻结帧期间, 包括游戏计时器在内的所有除去 FreezeTimer 的计时器, 都停止走时.

在第 $5 \leq i \leq 14$ 帧, 我们见到 DashCD(15-i), Ground.

在第 15 帧, 我们见到 Ground¹¹. 其中 DashCD(0) 不显示.

在第 16 帧, 我们见到 Frozen(3), Jump(10), DashCD(11).

实践上在交流中, 人们对某些状态到底有几帧是比较含糊的. 或者说由于游戏内部运算顺序的问题, 人们对这帧到底算不算这个状态, 似乎没有统一的标准. 所幸大多数情况下我们只需要在 Celeste Studio 里面大概写一写, 然后看信息面板的提示, 就知道具体精确到帧该怎么写了.

我的原则是, 大部分倒计时类按照 Celeste Studio 显示的最大数字 +1 计. 例如 varJumpTime 12 帧, 冲刺 CD 12 帧, 狼跳 5 帧¹²(因为在最大数字的这一帧也往往具有这个状态). 少部分如 NoControl(40), 是 40 帧, 移动块的 LiftBoost 是 10 帧 (因为在下一帧才能吃到!). Celeste Studio 没显示的倒计时类则按生效的帧数决定 (例如 dashAttack 18 帧 (不含冻结帧)). 对于状态机中的状态, 按照 Celeste Studio 显示了几帧就算几帧¹³(例如 StFlingBird 最短 30 帧 (含 3 帧冻结帧), StDash 最长 14 帧 (含 3 帧冻结帧)¹⁴). 这听起来依然是一个很混乱的标准... 所以我无法保证之后的内容能够把时长都说准确... 希望我每次都记得注明到底怎么算这个状态吧...

2.3 空间

单位是像素 (px).

整数坐标 Position, 亚像素/小数部分. 碰撞等使用整数坐标.

大体来说, MoveHExact 只移动整数坐标, MoveH 则是移动 ExactPosition (呃, 这命名可能有点微妙). 同理 MoveVExact, MoveV.

(具体一点点, MoveH 接受 float 型参数, 对 movementCounter.X 进行调整后, 调用 MoveHExact. 而 MoveHExact 接受 int 型参数.) 大多数情形

(如果在 MoveHExact 移动的方向上, 产生了碰撞, 那样会将亚像素重置为 0, 并产生 CollisionData, 交给 onCollide 进行处理. 否则的话就真的是只移动整数坐标.)

一个砖块是 16 像素?.

¹⁰在 CelesteTAS v3.21.0 之后, 这被 Coyote(5) 替代.

¹¹旧版本的 TAS mode 还会显示 CanDash, 这是一个 bug.

¹²注意部分人会吧离地的那一帧也算狼跳, 这样就是 6 帧狼跳! 尽管由于运算顺序, 这一帧在系统看来并不算狼跳.

¹³尽管可能这帧按照状态 X 来更新, 但因为与某些实体碰撞, 变成了状态 Y. 为了方便计算这种情形也视为属于 Y.

¹⁴呃, 中途遇到其他 Frozen 不算. 这里只计自带的帧数.

整数坐标 = Round(精确坐标), 用于渲染, 判定等.

呃, 说起来为啥亚像素指示器不是显示 $-0.5 - +0.5$, 这让人感觉有点怪. 呃好吧 0.000006 这种离临界点的距离似乎也不错.

亚像素某种意义上是在这个像素游戏中的连续性. 尽管渲染是像素的, 但是一部分内部运算 (尤其低速运动) 依然需要亚像素, 来使得物体正确的运动. 你也不想一个运动特别慢的 ZipMover 动了半天, 但每帧都因为不到 1px 而直接被 round 到 0 吧? 这种情况当然是亚像素会很方便.

2.4 速度

速度 (Speed) 是 Maddy 自身的属性, 是一个二维矢量.

位移 = 这一帧相对上一帧的位移.

Speed = 这一帧 (Celeste Studio 显示的) Speed.

一般情况下, 位移 = Speed * Engine.DeltaTime¹⁵. 因此大致上 Speed(的水平/竖直分量) 的单位是像素/秒 (px/s).

但 Maddy 的位移还会受到其他因素影响, 例如被移动块推动, 被弹簧弹, 墙角修正, JumpThrough 修正等. 于是 Celeste Studio 定义了 Vel = Velocity. 这是个游戏内部不存在的概念, 定义是 Vel = 位移 * 60.0 (不受 Engine.TimeRate 影响)¹⁶. 反映的是在现实中一秒中走过的像素数.

由于 Engine.DeltaTime \neq 1/60, Speed 与 Vel 总有微小的偏差.¹⁷

2.5 加速度

尽管源码中加速度的单位是 px/s², 但我们都将其转化为 px/s/f. 代价只是数值看起来不那么整.

2.6 碰撞箱与伤害箱

2.7 操作

它们大部分在状态机更新中起作用. 少部分在之前的确定朝向等地方, 会用到. 呃, 等效来看, 把确定朝向等也视作在状态机更新中, 也不是不行.

应该无法做出亚帧级操作 (关于这个概念, 参考 <https://www.bilibili.com/read/cv3629602/>). 但我没看过相关源码, 不确定.

同一帧内, 先更新状态 (各类倒计时), 然后位置 + 状态 + 输入 = 实际结果, 然后更新速度/位置.

由于位置是在之后更新的, 所以大部分情况下, 上一帧的位置 + 这一帧的状态 + 这一帧的输入 = 结果.

少部分例外如风吹动带来位置改变, 在判定输入成果之前.

¹⁵注意这受到 Engine.TimeRate 的影响.

¹⁶定义见 CelesteTAS-EverestInterop/Source/TAS/GameInfo.cs.

¹⁷呃, 除非都精准地为 0.

3 StNormal 中的一般运动

在大部分情形下, Maddy 处在 StNormal 当中.

3.1 水平运动

Approach 函数.

空气阻力项在 NormalUpdate() 中以形如

```
Speed.X = Calc.Approach(Speed.X, num3 * (float)moveX, 400f * num2 * Engine.DeltaTime)
```

的形式出现.

等效于空气阻力为 10.83 px/s/f...

冰面上的阻力为... 比空气小, 因此在相同跳跃次数的情况下, 尽量多在冰面上滑行而减少在空中滞留的时间是有利的. (呃, 如果是无限长场地呢, 我还没算过)

呃, 我不知道怎么给这个过程命名, 地面变速/空中变速? 地面加速/地面阻力/空中加速/空气阻力? 我很想直接叫地面阻力/空气阻力, 尤其是注意到在高于 90 的速度时, 按反方向或是什么都不按, 居然加速度一样, 我就特别想这么叫. 低速情形... 叫法就直接同高速情形, 或许也未尝不可.

3.2 竖直运动

Fastfalling, Slowfall.

MaxFall.

只是在!onGround 的情况下会有这样的竖直速度变化. 特别的, 你可以以 0.0X 的速度保持在地面上滑行, 使得不受向下刺的伤害.

<https://discord.com/channels/403698615446536203/1074148268407275520/1089355729191841873>

4 跳跃

有一些基础的动作，它们可以在不同的 State 下触发，但触发条件和顺序可能不一样。我认为它们本身比 State 更重要，因此以它们的视角来展开。而对于比较特殊的自成一体的 State (羽毛, 鸟等)，就以 StateUpdate() 的视角来写。

呃，我不是很确定是否还需要分为跳跃，抓墙，冲刺三大母题。

或许我在这里应该着重声明：它们本质上应该被视为不同的操作。它们仅仅是具有了类似的形式。并不存在一者是另一者的升级。

在 StNormal 中，优先顺序为：狼跳 > 抓跳 > 蹭墙跳 > 踢墙跳 > 水面跳跃。

更细一些：地面跳跃 = 狼跳 > 右墙抓跳 > 右墙蹭墙跳 > 右墙踢墙跳 > 左墙抓跳 > 左墙蹭墙跳 > 左墙踢墙跳 > 水面跳跃。

以下：按下跳跃键 = 按下跳跃键且此时不会触发可对话的实体。则触发条件为：

狼跳 (Jump)(这里把地面跳跃和狼跳统称狼跳，因为都是检查 jumpGraceTimer > 0f)：在地面上 (此时狼跳时间为 6f)，或者在狼跳时间内按下跳跃键。水平速度 += 40 * moveX。

抓跳 (ClimbJump)：位置允许取消蹲姿，面朝墙壁方向，体力 > 0，按下抓键 + 按下跳键，手上没有东西，正方向 3 格内无 ClimbBlocker，且 WallJumpCheck 通过，则可以抓跳。也就是大多数情况下都是 3 格判定范围，上冲之后且还在 DashAttacking 的 StNormal 有 5 格判定范围。对 Maddy 自身的效果除去 wallboost 以及体力等之外，基本同 Jump。因此会取决于 moveX 获得朝墙方向 +40 或 +0。

蹭墙跳 (SuperWallJump)：位置允许取消蹲姿，DashAttacking 且 SuperWallJumpAngleCheck 且 WallJumpCheck 且按下跳键。水平速度为 170 + LiftBoost。

踢墙跳 (WallJump)：位置允许取消蹲姿，WallJumpCheck 且按下跳键。水平速度为 130 + LiftBoost。

水面跳跃：位置允许取消蹲姿，且下移 2px 会撞到水体。效果同 Jump。

WallJumpCheck：如果 DashAttacking 且 DashDir 朝正上，且在正方向上移动 5px 不会碰到反向的刺，则 num = 5。否则 num = 3。然后检测无关紧要的 ClimbBoundsCheck (关卡边缘相关) 以及 !ClimbBlocker.EdgeCheck (冰墙相关？但我尚且未发现 ClimbBlocker.edge = true 的对象，这使得 !ClimbBlocker.EdgeCheck 必然返回 true...)，如果通过的话检测正方向 num 距离内是否与 Solid 碰撞。是的话返回 true。否则 false。(注意，因此抓跳也容许 4 格间距！)

位置允许取消蹲姿：字面意思。如果本身就不在蹲姿也算。

例：在图 1 中这个位置的下一帧按下 J，一般来说按下跳键只可能触发踢墙跳，水平速度变为 +130。但这里因为还有狼跳帧，所以优先触发跳跃，水平速度为 $0 + 40 * 0 = 0$ 。若按下 L, J，则水平速度为 $-10.83 + 40 * (-1) = -50.83$ ，然后撞墙变为 0。若按下 R, J，则水平速度变为 $10.83 + 40 * (+1) = +50.83$ 。

在 StDash 中，优先顺序为：SuperJump > SuperWallJump > ClimbJump/WallJump。特别的，当你还有狼跳帧 (且冲刺方向是水平) 的时候，你无法横冲 cb +40。(但先撞墙获得 retention，然后再翻越过去恢复速度，这依然



图 1: Jump 优先于 WallJump

是可行的)(典型例子, 充能 hyper 起手 (比如一些 gu), 这样的话必然是有狼跳帧)

在 StClimb 中, 只有踢墙跳和抓跳, 触发条件: 按下跳键 (不需要可对话的实体云云), 位置允许取消蹲姿. 此时如果 $moveX = -Facing$, 则 $WallJump(moveX)$, 否则 $ClimbJump()$. 注意, 不需要这一帧按着抓键. 由于进入 StClimb 的瞬间 Facing 都被设定好了, 必然是面对着墙壁的, 而 StClimb 期间 Facing 也无法变动, 因此玩不出花样, 绝无可能向着离开墙的方向抓跳 (这需要 $moveX = -Facing$, 但这会触发 $WallJump(moveX)$)(理论速度为 $-40+LiftBoost$). 因此只可能产生三种情形:

- 向离开墙的方向 WallJump (速度 $-130+LiftBoost$).
- 向着墙抓跳 (速度为 $40+LiftBoost$).
- 中性抓跳 (速度为 $LiftBoost$).

在拓展异变的情况下¹⁸, 会将 Normal Update 中用到的 $jumpGraceTimer$ 用拓展异变的 $float canJump$ 代替. 总而言之效果是 StNormal 中, 狼跳 > 抓跳 > 蹭墙跳 > 踢墙跳 > 空气跳.

¹⁸ExtendedVariants.Variants.JumpCount.patchJumpGraceTimer/canJump, <https://github.com/max4805/ExtendedVariantMode/blob/40780c8>

5 抓墙

StClimb 之 climbNoMoveTimer, 这是一段“不应期”, 使得你不会立刻被传送带带上去.

StClimb 期间, Speed.X = 0.

StClimb 维持不动的情况下的 height 限制. 1,G;1 循环来稳定在超出此限制的位置.

5.1 爬墙

当向上爬的时候, 头顶 1px 是墙的时候, 会直接将速度归零. 而由于向上爬的最终速度是 -45, 因此正常情况下, 向上爬墙顶头无法重置纵向亚像素.

正常向上爬能爬多高

如何维持在更高的位置

5.2 抓跳

StNormal 下抓墙优先于抓跳.

StClimb 不会再触发抓墙, 因此触发抓跳.

因此一个没控好距离的抓跳会需要 2 帧且吃不到加速, 控好的只需要 1 帧.

5.3 Wallboost

Wallboost 给 Stamina += 27.5f. 因此可以 overclock.

5.4 体力

所有会体力变化的情形:

wallboost, + 27.5

各类体力 = 110. (体力 < 20 时/冲刺数不足时使用水晶, booster/Badeline 球/羽毛/鸟/Badeline boss, 各类弹跳, 进入 StSwim, dreamblock, 所有 onGround 的帧, 无限体力 assist 的帧)

向上爬墙, 每帧 -...

抓墙, 每帧 -...

不从地面起的抓跳, -27.5

爬墙状态下, accelerating and then alternating between D and U gets you 7/8 of the speed but 1/2 of the stamina lost

5.5 SlipCheck

5.6 cb

corner boost 这个名字偏现象, 实际上似乎会把每一次通过正方向抓跳来获得速度 +40 (以及 LiftBoost) 的都叫做 cb. (正方向即指与速度方向同向). 反方向则 rcb.

cb is climbjump + retention

从机制角度, 何时是 corner jump, 何时是 corner boost (即 +40 是否吃到)

低速情况下允许双 cb, 因为机制...

低速情况下, 各个速度区间所能达到的最大速度值. (e.g. wallboost 126 吃双 cb)

抓跳需要能 uncrouch. 所以即使从物理上来看蹲姿 cb 可以过的一些地方, 实际上却过不了.

由于判定抓墙需要足够体力, 因此体力很低时也能抓跳, 正如 2px, 向上速度条件一样.

5.7 Climbhop

Climbhop 的触发效果大体可以分为两部分机制, 其一是竖直速度变为至少 -120, 横向速度变为 100 (), 且获得中性的 forceMoveXTimer = 0.2f, 其二是随 climbHopSolid 的运动. 此外, Climbhop 还会产生 noWindTimer = 0.3f, 因此 4A 中我们能不受风影响地爬上悬崖.

而 ClimbHop 本身的触发, 需要在 StClimb 中竖直速度严格向上, 前方无固体, 且不在 wallboosting (传送带那个). 或者在 StSwim 中的那个, 我懒了所以此略.

Climbhop Cancel 是少有的纵向高速移动的手段.

会有 forceMoveX= 0.

呃, 据说 anarchy Collab 有张图, 可以让你对此加深认知.

Climbhop 会使得移动块 climbHopSolid 的位移取整后叠加到 Player 上. forceMoveXTimer > 0f 期间, 一直有效. 直到 forceMoveXTimer <= 0f, 此时会设置 climbHopSolid = null.

最近 (20230831) 的 1A 优化, 利用 climbhop cancel 实现冲刺期间蹭带刺的墙.

呃, 想要完全理解 climbhop cancel, 好像并不是那么容易. 我们来看看怎么实现 6A 的 7f climbhop cancel 吧, 这个似乎不是一下子就能调出来的.

利用 ClimbHop 机制使得 StNormal 下也能抓住速度过快的移动块 <https://discord.com/channels/403698615446536203/51928>

基于机制, 抓住 CassetteBlock 向上爬, 然后 CassetteBlock 消失, 也会触发 ClimbHop. 但如果不向上爬就没有.

6 冲刺

基本机制: beforeDashSpeed 与冲刺速度的关系.

各类交互中重要的两项: DashDir, DashAttacking

Ultra 的机制在源码里是 Dash Slide, 乘子是 DodgeSlideSpeedMult = 1.2f. Dash Slide 以两种情形出现, 一是 onCollideV 里, 二是 DashCoroutine 里. DashCoroutine:

当你在冲刺时, 并且在地面上, 并且 DashDir 如图, 那么你就可以 DashSlide. 但是这会“消耗”掉你的 DashDir, 使它变成水平方向的. onCollideV:

我翻不到源码, 但根据现象, 可能是: 正常情况下, 冲刺只能在第 5f 输入方向作为 dir. 而超级冲刺则是 dashAttackTimer 不为 0 时, 在冲刺中的每一 f 都可以输入方向作为 dir. 反编译出来了, 超冲转弯是由 canCurveDash 控制的, 这项在 DashBegin 时设为 true, 在 onCollideH 或 onCollideV 调用时设为 false, 如果打开了超冲异变且 canCurveDash, 则可以超冲转弯那么对于在地面上开始的 R,D,X

那么在第 1 帧, 我们有 dashStartedonGround = onGround = true. 在冲刺的第 5 帧 (实际按下 R,D 时), 首先我们成功触发 DashCoroutine 里的 dash slide, 根据原速度 + 斜下冲刺算出冲刺速度, 然后水平速度 *1.2, Speed.Y=0, DashDir 变为朝右. 然后再检测碰撞, 合理认为我们没有和地面发生碰撞 (虽然我没完全看懂源码为什么表明没和地面碰撞...). 于是后面还可以再接超冲的拐弯. 因此如果是 5,R,D,X ; 1,R ; 1,R,D, 你就能在第 7 帧第 8 帧见到第二次 1.2 倍加速. 这次就是和地面碰撞了!

起手也可以是离地面 1px 的开始的冲刺 R,D,X. 后面完全一样.

(第 1 帧 onGround = false, 于是触发 Dash corner correction, 向下移动 1px. 会直接 return 掉, 而不是 onCollideV 一路跑到最末尾触发 collide event. 然后后面当然就完全一样了. 虽然按理来说这帧的 Speed.Y = 0 触发不了才对...姑且就当申必浮点数原因导致 0>0 吧...)

Fuck , 为什么还可以离地 2px 吸附下来...

呃, 冲刺第一帧的吸附应该是这条: (Player.update() 里, base.update() 后)

```
if (!onGround && DashAttacking && DashDir.Y == 0f && (CollideCheck<Solid>(Position + Vector2.UnityY))
{
    MoveVExact(3);
}
```

呃, DashCorrectCheck 应该是用来防止你被吸附到带刺的地板上致死的, 它就是检测伤害箱有没有和 LedgeBlocker 相撞. 你可以看到三个朝向的 Spikes 自带 LedgeBlocker 的 component. 圆刺也有.

我在 2023/01/03 优化 6B 的时候, 就是因为被这个坑了所以才没有多省 1f.

应用: 在机关上, 调整竖直亚像素 (你需要在这段时间取得 LiftBoost), 然后正/反 hyper, 1,R,J, 这 1,R,J 既触发向下 1px, 又触发 hyper, 向上移动 52.5/60, 在亚像素合适的情况下刚好还能抓住机关侧面, 然后 1,L,G,K, 这样就能吃到非常高的速度.

Roboboost = moving block 6f bunny hop subpixel hyperdash reverse cornerboost, 这个就是后面的这段 sphdrCb. 亚像素调整不一定是通过 mb6f 来搞的. Kevin 块也有调这个亚像素的案例.

似乎 Celeste studio 显示的 speed 是, 用于计算这一帧位置更新的那个 speed, 于是由于碰撞而改变的 speed, 实际上是留到下一帧用的, 因此也在下一帧显示. 但 DashCoroutine 里的 Dash Slide 自然是在位置更新之前就计算出来的.

所以 5,R,D,X ; 1,R ; 1,R,D ; 1,R , 会在第 5 帧看到第一次 1.2 倍加速, 在第 8 帧看到第二次 1.2 倍加速.

当然, 这玩意不会有 buffer 的问题, 所以 5,R,D,X ; 1,R,D ; 1 就可以在第 7 帧见到 1.44 倍速了. 中间不需要那个 1,R.

是的...1.44 倍加速的 true grounded ultra 在空间足够的情况下其实很好手操, 只要你是落地之后再按 6 帧 R,D,X 就行了. 空间不足差 1px 落地的情形就恶心了...

由于需要可以 unduck, 即使有跳跃次数也不能斜冲天花板然后变成横冲然后搓出 super.

正 super 可以 5 帧搓出, 反 super 需要 6 帧搓出.

第一帧速度是 0, 因此你可以在带上刺向右移动块的右边若干 px (取决于移动块速度), 上方 1px 处, 开始右下冲刺.

$240 > 240 * \text{Sqrt}(2)/2 * 1.2$, 因此在 XX 情形下应该 R,Z 而非 R,D,X

在冲刺状态下, 蹭墙跳优先于抓跳. 这和 StNormal 是反过来的.

可以 14,R,X; 1,R,J. 这第 15 帧实际上是用的 DashUpdate(), 但是 DashEnd(), NormalBegin() 了, 所以显示 StNormal. 实际上操作还是冲刺相关操作.

冲刺第一帧强制静止, 作为推论, 在带刺的单向板处向上冲会死 (如果脚没在刺以下).

因为 DashUpdate() 里允许的操作只有 SuperJump(Ducking or not), SuperWallJump, WallJump, ClimbJump. 所以你即使有无限跳跃, 也无法在斜向冲刺时用跳跃打断! 这是因为没有墙壁时只可能触发 SuperJump, 但 SuperJump 需要 $\text{Math.Abs}(\text{DashDir.Y}) < 0.1f$. 相对地, 空中横冲就可以用跳跃进入 SuperJump 来打断冲刺.

Hyper 和 wavedash 本质上并不是完全相同的机制.

DashCoroutine 和 onCollideV 都是重要的! 特别的, 在你 (没撞墙) 在角落“日门”的时候, 用的是 DashCoroutine 的 1.2 倍加速. 如果按 onCollideV 来算, 绝无可能吃到这个速度! (因为会先 onCollideH 速度归 0.)

冲刺只会水平转向, 不会竖直转向, 这里的转向指的是 DashDir 改变. 特别的, 斜上冲撞竖直墙后无法蹭墙跳. 呃, 撞天花板也不行. 并且由于 OnCollideV 只在 MoveV(Speed.Y * Engine.DeltaTime, onCollideV) 一处用到, 所以大概是没办法指望通过 Maddy 本身速度以外的位移来产生 1.2 倍加速之类的.

冲刺结束后会设置 AutoJump (无论冲刺方向!), 所以如果你有需求的话, 确实可以用 CutScene 来取消冲刺从而取消 AutoJump.

冲刺结束后会重置速度。特别的，如果你斜上冲但撞到了天花板，则第 15 帧重置速度时你还是会获得向上的速度，这使得你能更快的斜上冲反抓。

冲刺的角落修正也会使得超冲无法拐弯，因为这是在 OnCollideH/V 中的，它们在最开始就会设置 canCurveDash = false。但这并不会重置 DashAttackingTimer 所以仍可以撞碎砖块。

OnCollideH 里设置了，若发生了正常碰撞，且如果在冲刺状态或红泡泡状态，那么如果在地面（由于判定顺序，只需要这一帧开始的时候在地面上，并不需要紧挨墙壁的时候也 onGround 图 3）且前进 1px 后可以以蹲姿避开碰撞，那么就会进入蹲姿（注意，OnCollideH 在这里直接 return，不会再水平移动，也因此不会有 retain 之类的，水平速度也不会消失。但也因为一切发生在 OnCollideH 里，水平亚像素会被重置。）。因此 Maddy 在一些很矮的地方横冲时会自动进入蹲姿。（我称之为钻洞修正）如 图 2。此外，注意这也并不需要冲刺是水平方向。

冲刺的第一帧设置 DashDir = Vector2.Zero，DashUpdate 中有一些地方会检测 DashDir != 0，这使得第一帧无法进行一些操作（如抓水母）

按住跳 + 平 u 抓水母打断，可以起到利用 varJumpTimer 上飘一段的效果。



图 2: Maddy 是猫猫虫的证据



图 3: 注意 onGround 的判定时机

6.1 Ultra 机制

ultra 的 dashdir 在哪些情况下失去：注意，Rebound, Bounce 之类的并不。

6.2 各类取消冲刺的方法

正面撞墙（如果是门/下落块等，还可以利用 retention 搞出 ultra）

水母/Theo 水晶

剧情动画

斜向绕角 (撞墙取消冲刺, retention 恢复速度. 斜下的话, 直觉上斜下冲就是保留水平速度的, 虽然仔细一想还得是 retention, 但不违反直觉. 斜上绕角则略反直觉, 需要注意. 不过除了无抓谁会这么干 (呃, 还真有应用场景, 8A, 8B 天然有一些不得不无抓的地方)... 除非需要绕过去之后光速下落)

7 物理更新与固块碰撞

首先介绍 (onCollide 不固定) 的 MoveH/VEExact, 注意必须 move 不为零才会发生碰撞事件.

在 MoveHEExact 中, 如果朝着 moveH 方向移动 1px 会与固块产生碰撞, 那么去处理碰撞事件 (触发 onCollide, 但 onCollide 参数默认为空, 除非比如 player 物理更新里调用 OnCollideH, 或者移动块更新里调用 OnSquish), 并结束 MoveHEExact. 否则移动 1px, 然后 moveH-=1. 循环直至 moveH=0.

这是机制上不允许高速穿过固块.

然后再讲 OnCollideH/V.

怎样会产生蹲姿

7.1 MoveH

MoveHEExact, MoveV, MoveVEExact

7.2 OnCollideH / OnCollideV

竖直撞的优先级是 -1, -2, -3, -4, 1, 2, 3, 4, 水平撞是 1, -1, 2, -2, 3, -3, 4, -4, 但水平撞会受到 LedgeBlocker 的阻碍 (圆刺, 煤球, 尖刺都带有此组件)

7.3 Retention

Timer <= 0 时, 在 onCollideH, 会将 Speed.X 赋予 wallSpeedRetained, 并重置倒计时.

如果 Timer > 0, 无法更改 wallSpeedRetained.

倒计时: 如果 Speed.X 与 wallSpeedRetained 反向, Timer = 0, 不改变 Speed.X. 否则, 如果在 Maddy 的 wallSpeedRetained 方向 1px 处没有墙, 将 Speed.X = wallSpeedRetained, 并 Timer = 0. 否则, Timer -= Engine.DeltaTime.

```
if (wallSpeedRetentionTimer > 0f)
{
    if (Math.Sign(Speed.X) == -Math.Sign(wallSpeedRetained))
    {
        wallSpeedRetentionTimer = 0f;
    }
    else if (!CollideCheck<Solid>(Position + Vector2.UnitX * Math.Sign(wallSpeedRetained)))
    {
        Speed.X = wallSpeedRetained;
        wallSpeedRetentionTimer = 0f;
    }
    else
```

```
    {  
        wallSpeedRetentionTimer -= Engine.DeltaTime;  
    }  
}
```

与之形成对比的是 `LiftBoost`, 可以每一帧都更新数值.

移动块, 假设移动块每帧后撤 1px, 那么按照更新顺序, 抓跳, 撞墙, 移动块后撤, 倒计时更新 & 速度变为 `wallSpeedRetained` (判定为第二种情形). 反复循环.

在移动块并不矮且左下方有地面 (或者跟着它一起运动的移动块) 的情形, 甚至可以落地恢复体力于是多次抓跳 <https://discord.com/channels/403698615446536203/754495709872521287/1066488134608621728>

呃, 值得注意的是, 利用下蹲也可以恢复速度将 `retain` 拖很久 (会稍微吃一些阻力) <https://discord.com/channels/403698615446536203/754495709872521287/1066488134608621728>

8 实体碰撞

碰撞箱 vs 伤害箱: Maddy 实质上只有一个 Collider, 但是具体用的时候碰撞箱和伤害箱轮流上岗. 因此它们之间其实并不是完全隔离的, 而是稍有关联, 这也是碰撞箱 = 羽毛伤害箱能成立的原因之一.

由于更新顺序, 用到的实体的碰撞箱晚于实际位置 1 帧.

会先将 Maddy 的碰撞箱换成伤害箱, 然后开始碰撞检测.

目前见到的所有实体碰撞结果 (Action<Player> onCollide), 基本都形如

```
Collider collider = base.Collider;
...
base.Collider = collider;
```

这使得碰撞自始至终用的 Maddy 的伤害箱都是同一个, 而不会有先后顺序的问题.¹⁹

呃, 也有极端情形, 甚至会造成碰撞箱异常 (碰撞箱 = 羽毛伤害箱): <https://discord.com/channels/403698615446536203/7544>

一般来说不会多次碰撞, 由于伤害箱会重置回碰撞前

8.1 实际碰撞箱/实际伤害箱

Maddy 的伤害检测用的是 Actual Collide Hitboxes (紫色), 这玩意是用上一帧的位置 + 这一帧的 speed 算出来的 (再加上撞墙的种种修正), 不涉及风/物块推动/其他实体交互带来的位置改变等项. 这是由运算顺序导致的.

唯象地, 运动实体 (包括 spike 在内!) 的碰撞箱会晚于实际位置一帧. 这并不是 bug, 只是我们 Celeste Studio 所选取的观察时点的原因. 在运行逻辑上, 碰撞箱确实在这一帧到达了正确的地方. 但是在 Maddy 的视角下, 她 update 完之后, 其他实体的运动已与她毫无关联. 因此 Maddy 视角中的一帧, 看到的实体, 实际上在程序上的上一帧的位置处. 因此运动实体 (在与 Maddy 交互的意义下), 也有 Actual Collide Hitboxes.

BadelineBoost 的 BoostRoutine (StDummy 那段) 也会导致碰撞箱漂移. 因为这是实体 BadelineBoost 的更新过程导致的, 而不是 Maddy 自身的更新. 与之相比, FlingBird 带来的位置更新是写在 Maddy 自身的状态机里的, 因此不会碰撞箱漂移.

总结: 与实体交互 (包括圆刺, 尖刺, 弹簧等), 用 Maddy 的“实际伤害箱” (紫色) 与实体的实际碰撞箱. 与固块交互 (判定落地, 抓墙等), 用红色碰撞箱.

蓝色碰撞箱可以用于显示被物块推动之前的位置, 绿色伤害箱... 可能没啥用, 但大部分时候它与紫色伤害箱一致.

8.2 各类弹跳

- OnDashCollide 相关:

¹⁹但也存在非常奇怪的反例, 似乎表明 Maddy 的伤害箱会变, 原因不明. 不过至少原版似乎还没反例. <https://discord.com/channels/403698615446536203/854300051404095498/1029863643522801684>



图 4: 被大风推动, 因此虽然绿色伤害箱已经和刺重合, 且没有向左速度, 但不死. 因为 Actual Collide Hitboxes (紫色) 还没和刺重合.



图 5: 紫色伤害箱与刺重合, 但速度 (Speed.X) 向左, 不死.

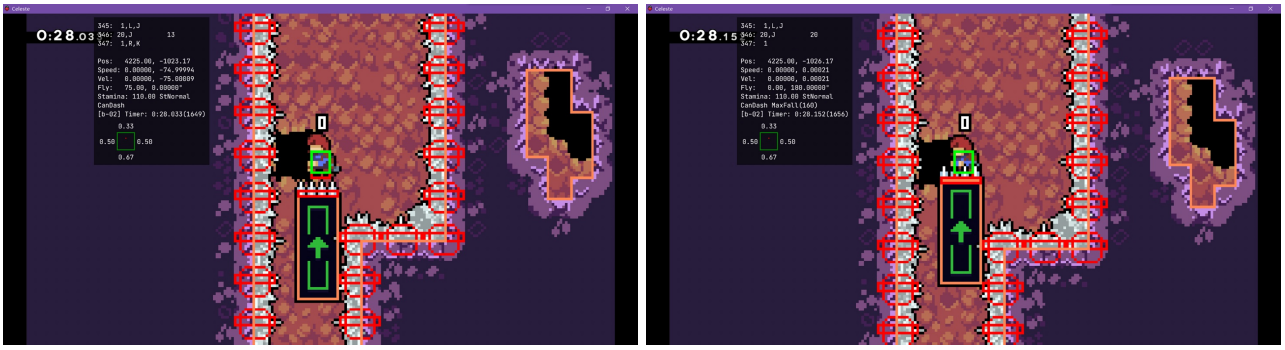


图 6: 由于延时 1 帧, 刺的碰撞箱和移动块的上边缘的距离变为 1px-2px



图 7: 因此可以在合适的时机实现, 你的脚碰到移动块, 而伤害箱不与刺重合.



图 8: 在下一帧, 刺将上移. 由于实际伤害箱不计物块推动, 因此会被向上抬升 1px 并死亡.

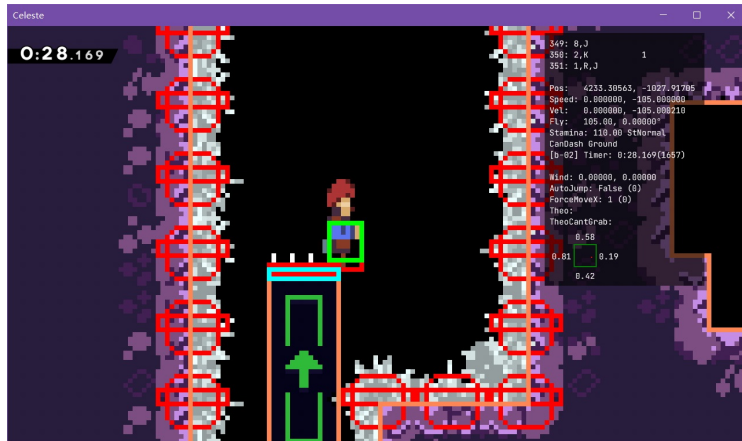


图 9: 如果改为跳跃, 那么 Maddy 有自己主动抬升, 实际伤害箱就更高, 伤害箱与刺不重合就不会死.



图 10: 由于大风, 实际伤害箱在左侧 1px, 恰好不死亡. (呃, 应该说不死亡本质上和风没有任何联系. 风的作用只是将玩家移动到指定位置, 使得能够进行相应的操作. 不死亡还是因为速度同向/碰撞箱就没接触.)



图 11: 因此你可以在下一帧获得 onGround, 并走下平台进入狼跳时间.



图 12: Coyote Hyper

- 撞击可撞碎的砖块 (DashBlock) 是 Rebound
- 撞击 3A 的果冻 (MrOshiroDoor) 是 ReflectBounce
- 弹簧:
 - 向上弹簧是 SuperBounce, 重置竖直速度为 -185, 水平速度为 0. 当速度向上时不会触发弹簧.
 - 水平弹簧是 SideBounce, 重置竖直速度为 -140, 水平速度为 ± 240 . 当速度同向且 > 240 时不会触发弹簧.
 - 原版没有向下弹簧.
- 踩 Seeker/踩雪球/踩 Oshiro/踩河豚是 Bounce, 重置竖直速度为 -140, 不重置水平速度
- 被河豚炸/弹球/被 Seeker 炸是 StLaunch 中的 ExplodeLaunch, 其中超级弹由 `Input.MoveX.Value == Math.Sign(Speed.X)` 则 `Speed.X *= 1.2f` 给出. 新版还额外使用了 `explodeLaunchBoostTimer = 0.01f` 来增加容错, 再加上冻结帧有 6f, 一共有爆炸 1f+ 冻结 6f+Timer 1f=8f 的时间来输入超级弹. ExplodeLaunch 在某几个特定方向上会有方向校正. (注意, Seeker 炸也有超级弹)
- 蓝心 / 气泡 (比如 6A 的含羽毛气泡) / 侧面碰 Seeker, 是 PointBounce, 机制比较奇怪, 不会正上下左右弹.

<https://xminty7.github.io/bumpertool.html>

Bounce, SuperBounce, SideBounce 都会强制跳转回 StNormal.

弹簧移动 Maddy 的机制 (强制回到 StNormal, 狼跳帧清零, ...)

向右弹簧弹的瞬间, 会试图将 Maddy 的碰撞箱左下角 (尽管与弹簧碰用的是伤害箱!) 移到弹簧的最右的中心高度处 (但纵向位移被 ± 4 界住). 由于碰撞箱位置是整数, 弹簧位置也是整数, 因此水平位移整数个像素 (`MoveH(Spring.Right - Player.Left)`), 而弹簧中心的纵坐标是半整数, 因此竖直位移半整数个像素 (`MoveV(Calc.Clamp(Spring.CenterY - Player.Bottom, -4f, 4f))`)(除非超过 ± 4 的情形).

电箱: 如果你在撞电箱的前一帧恰好处在电箱上方, 使得撞电箱的那帧能判定为 `onGround`, 那么你可以获得狼跳帧以及冲刺恢复.

向上的刺: 如果脚底比刺低, 也不死.

尽管大部分实体不像 Actor 类那样有 `movementCounter` 用来产生亚像素, 但 `Entity.Position` 本来就是 float 类型!

只是各类实体的位置在设置关卡的时候, 似乎都被设置成了整数, 而大部分与 Maddy 调整自身亚像素有关的实体, 它们的运动写的也恰到好处, 使得位置依然始终是整数. 不过也有部分, 比如雪球, BladeTrackSpinner (5B 的那个), 位置就可以不是整数.

写的恰到好处: 移动块/移动单向板基于 `MoveHEXact/MoveVEXact(int move)`, 位置始终是整数; 弹簧以及各类随移动块/移动单向板运动的, 基于 `StaticMover` 这个 Component, `StaticMover` 会检测对应方向上的 Solid 与

JumpThru, 而 Solid 和 JumpThru 都会在 MoveHEXact/MoveVExact 中 MoveStaticMovers(Vector2.UnitX * move)/MoveStaticMovers(Vector2.UnitY * move), 其中 move 是 int 类型! 所以弹簧以及各类随移动块运动的, 位置也始终是整数.

于是三类会改变位置的 (呃, 移动块在单独的章节了, 这里就不提):

横向弹簧 SideBounce, 上面已经说了, 水平位移整数像素, 竖直位移半整数个像素 (除非 ± 4).

朝上弹簧 SuperBounce: MoveV(Spring.Top - Player.Bottom). 这依然是整数.

雪球/Seeker Bounce: MoveVExact((int)(base.Top - Player.Bottom)). 虽然雪球/Seeker 等自身位置不是整数, 但用的是 MoveVExact, 所以不影响.

一个细节: 各类弹跳除了 PointBounce 和 ExplodeLaunch, 都会重置 dashAttackTimer 且重置 StateMachine.State, 而 ExplodeLaunch 会提前结束 State 却不改变 dashAttackTimer. 特别的, 如果你在冲刺的第一帧触发 ExplodeLaunch (比如趁弹球刚刚恢复), 那么你就获得了 dashAttackTimer + 结束了冲刺 + DashDir = (0, 0), 然后再碰 DreamBlock 就会有平常难见的情况. PointBounce 虽然也不重置 dashAttackTimer 且会将 StDash 切换至 StNormal, 但是它可能的调用往往都是 DashAttacking 就进入某分支, !DashAttacking 才调用 PointBounce, 所以没法用 PointBounce 来做.

如果你不是在冲刺第一帧触发, 那么也是有应用的, 例如你需要往右走很远, 但只能向左进入果冻.

<https://discord.com/channels/403698615446536203/598977992957624343/1083304329894699108>

另一个细节: 理论上, 当你的水平整像素和 Puffer 的 Position (不带亚像素的那个) 完全一致时, 你可以被它往下炸. Puffer 的 ctor 中会对水平位置加上 idleSine.Value * 3f, 必须要这个随机数恰好随到整数才行... 很苛刻.

但也不是总那么苛刻, 注意到这个过程是浮点数加法, 所以位置足够远时, 对随机数的精度要求没那么高. 在 $[0, \pi)$ 上 (随机数实际范围是 $[0, 4\pi)$), 只需以同样精度靠近其中 5 个精确值, 或者以一半的精度靠近 $\pi/2$, 那么就足够了.

Farewell 足够远, 所以居然能找出两个例子来. 稍加估算, 发现在这个距离 (72000) 的浮点数加法精度是 $a = 0.004f$. 我算了下, 满足条件的随机数占比是 $p \approx 1.232a + 0.520\sqrt{a} \approx 3.7\%$ (呃, 这个公式是 Taylor 展开算的, a 越小越精确). 距离 4100 处 0.8%, 这对于一张地图算是很短的距离了 (1A 达到了 4000, 2A 3000, 3A 8000, 4A 15000, 5A 9000, 6A 25000, 7A 25000, 8A 14000), 因此这种现象应该意外的还不算过于罕见.

<https://discord.com/channels/403698615446536203/429775260108324865/909329896873025546>

9 移动块 / 移动单向板

注意, 与固块碰撞一节相区分. 那里是 Maddy 撞固块. 这里是固块推 Maddy. 两者的更新顺序不同.

Platform 分 Solid 和 JumpThru 两种 (或者说我还没找到第三种).

9.1 骑乘

骑乘单向板: 不违反以下任一条件, 且玩家不与单向板碰撞但向下移动 1px 之后碰撞.

- 不为 StClimb.
- 不为 StDreamDash.
- 竖直速度不朝上.
- IgnoreJumpThrus = false. (在原版中等价于不为 StReflectionFall)

骑乘固体的条件是 (由上到下依次判定, 符合前提条件则直接返回之后计算的结果):

- StIntroMoonJump 时, 不骑乘.
- StDreamDash 时, 与固体碰撞. (注意 JumpThru 在此状态下永不骑乘)
- StClimb 或 StHitSquash 时, 向面朝方向移动 1px 后与固体碰撞.
- climbTriggerDir 不为 0 时, 向 climbTriggerDir 方向移动 1px 后与固体碰撞.
- 默认情形, 向下移动 1px 后与固体碰撞.

9.2 穿墙

玩家不能主动穿过固体, 固体可以穿过玩家, 也可以作为隧道使玩家穿过别的固体 (隧穿).

- 玩家不能主动穿过固体: 玩家自身更新引起的移动, 每移动 1px 都会检测是否与固体碰撞.
- 固体可以穿过玩家: 固体在自身移动之后, 才会将碰撞到的玩家推动到其前端. 因此若固体速度过快, 有可能不与玩家碰撞而直接穿过.
- 隧穿: 固体在自身移动之后, 如果碰撞到玩家, 则会将其逐 px 地推动到前端. 在这个过程中, 如果玩家碰撞到了其他的固体, 则会调用 Player.OnSquish(CollisionData data). data 里记录了如果没有其他固体时, 玩家最终所应该到达的目标位置. 在 OnSquish 中, 会尝试以下几种方式的组合, 来试图避开碰撞:
 - 进入蹲姿;
 - 瞬移到原位置的 $[-3, +3] \times [-5, +5]$ 范围内;
 - 瞬移到目标位置的 $[-3, +3] \times [-5, +5]$ 范围内;

当然, 组合是有一定条件的, 也是有优先级的, 此略.

总之,

- 如果不能避开碰撞, 则死亡.
- 如果避开碰撞, 且前面采用的组合是: 进入蹲姿 (原先在蹲姿不算), 并且瞬移到原位置/目标位置的 $[-3, +3] \times [-5, +5] \setminus \{(0, 0)\}$ 范围内 (即原位置/目标位置处不可行), 那么无视当前的这一个移动块, 尝试解除蹲姿. 这可能会导致玩家卡进这个移动块中. (Koral Clip)

现在, 注意到 OnSquish 中的避障方式是瞬移, 因此是存在穿过其他固体的可能性的. 由于固体普遍至少有 8×8 的碰撞体积, 穿墙只能是通过瞬移到目标位置的 $[-3, +3] \times [-5, +5]$ 范围内. 此时, 有以下手段可以使得目标位置和原位置相距很远, 从而达成隧穿:

- 移动块足够快. 显然这使得目标位置和原位置相距很远.
- 玩家 **原先已经** 卡进移动块的内部, 且移动块足够长. 此时目标位置和原位置的距离的主要贡献来自于移动块的体长.

如果用 Koral Clip 卡进移动块内部, 则这一次 OnSquish 不满足“原先已经”. 想要利用 Koral Clip 进行隧穿, 一般是利用一个斜向移动的移动块. 根据运算顺序, 移动块首先横向移动, 此时需要触发 OnSquish 做出 Koral Clip. 然后移动块纵向移动, 试图将玩家推动到前端, 路径上遇到障碍物, 再次进入 OnSquish, 判定为移动到目标位置的 $[-3, +3] \times [-5, +5]$ 范围内, 成功隧穿.

此外也可利用关卡边界卡进移动块, 此略.

9.3 Solid

移动块相关的方法是写在基类 Solid 里的.

在 Solid 的 MoveHEXact 方法, 如果 Maddy 的 Input.MoveX (不是 moveX), Speed.X 与 move 都同方向, Maddy 没有骑乘在它身上, 且将 Solid 水平移动 move, 竖直向上移动 1px 后会与 Maddy 碰撞, 那么将 Maddy 向下移动 1px. 如 [图 13](#).

然后会对碰撞到 (且不是 TreatNaive 的) 的 Actor 执行 entity.MoveHEXact(moveH, entity.SquishCallback, this), 查询可知 Actor 类的 SquishCallback = OnSquish, 其中 Maddy 的 OnSquish 重载过. 注意, 在此期间会关掉 Solid 的碰撞, 因此 OnSquish 的致死必须要有另一个固块, 只有推动的固块本身是无法致死的 (哪怕 Maddy 卡在其中). (这里发生的事: 固块给了玩家预备的一个移动, 然后关掉了自身的碰撞, 任凭玩家自己再去碰撞别的固块 (但 TargetPosition (i.e. 如果除了此固体以外没有其他固体, 则 Maddy 应该到达的位置) 的信息被刻入 OnSquish 的参数, 使得固块假装还在).)

对于没有碰撞到的 Actor, 如果它是 Rider, 那么让它与移动块运动同样距离. 理所当然的不会调用 OnSquish.

(在 StDreamDash 下, Maddy 会被临时设置成 TreatNaive. 从而会被判定成 Rider.)

在这两种情形, 最后都会给予 Actor 以 LiftSpeed. 注意, 不是只给一个分量, 而是两个分量都给.

注意, Solid 一次性移动 move 而非每次 1px. 特别地, 注意到 Solid 对玩家的移动也是移动 move 后再一次性结算的, 因此当移动块速度足够快时, 它甚至可以穿过玩家. 不过这需求的速度基本得上千, 原版中除了加载瞬间

的瞬移钥匙门这类, 应该是见不到的. 更容易见到的现象是固体作为通道使玩家穿过别的固体, 见 OnSquish 章节.

同理 MoveVExact. 但是没有“向下 1px”的类似机制.

MoveHExact 和 MoveVExact 都可以触发 OnSquish, 因此一个移动块有可能产生两次 OnSquish (e.g. 第一次 TrySquishWiggle 纵向抬升, 第二次 TrySquishWiggle 横向移动. 我这里没有说反顺序)



图 13: Maddy 每帧向下移动 1px. 为了在横冲的时候触发而选择在水中进行.

向下 1 px 的机制有两方面作用: 1. (大概率是设计意图) 让你能从移动块顶端的边缘走下去. 2. 在移动块侧边的时候能够更快下落. (呃, 这可以让玩家试图躲避被移动块撞死的时候, 更加容易一些. 但这或许不是开发者有意为之.)

关于 1., 这有利于一些机关块反 cb, 如果你斜下冲之后正好在移动块上方 1 格, 那么这个机制能将你往下拉 1 格然后再横推, 从而下 1 帧可以顺利做出反 cb, 如 图 14.

关于 2., 那就是更快下落. 参考 图 15.

9.4 OnSquish

在 OnSquish 中, 有以下三步骤:

- (1) 如果不在攀爬状态且不在蹲姿, 那么先切换到蹲姿. 如果此时已经不与固块碰撞, 则 OnSquish 直接结束 (结果: 通过进入蹲姿避免了挤压). 否则, 按照碰撞结果试图将 Maddy 移动过去. 如果目标处没有障碍物, 则 OnSquish 直接结束 (结果: 通过进入蹲姿 + 被推动避免了被挤压). 否则, 还原 Maddy 的位置, 继续后面的判定. 以上三种结果都会设置 `data.Pusher.Collidable = false`.
- (2) 不管是否经历步骤 (1), 判定 `TrySquishWiggle(data, 3, 5)`, 这会:

- 重新设置 `data.Pusher.Collidable = true`;



图 14: Ultra Difficult 中的案例

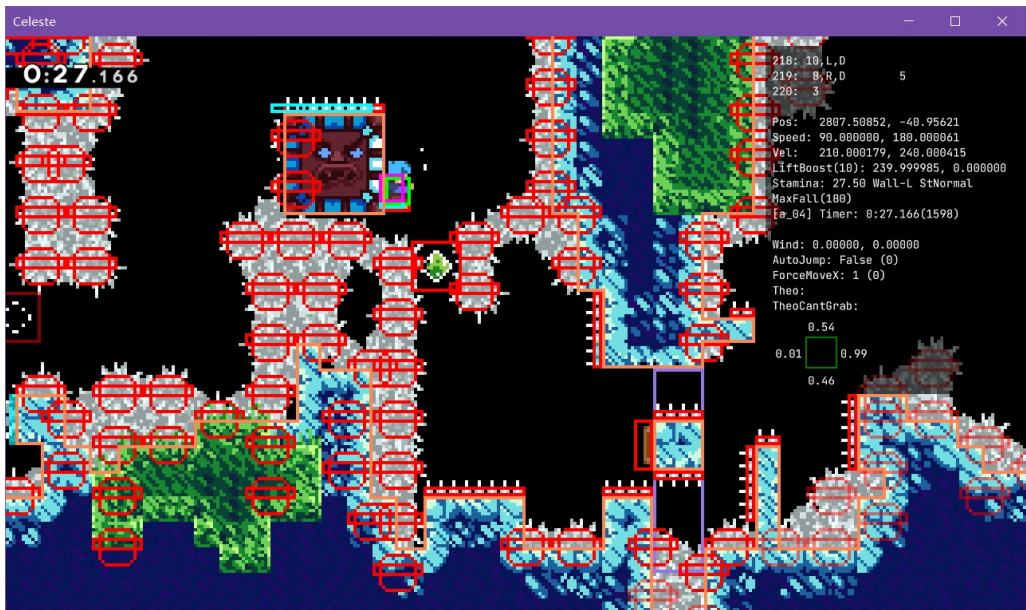


图 15: 日峰山中的案例

- 然后看 Maddy 在原位置处, 轻微扭动一下位置, 是否能避开这次碰撞; 如果成功, 那么这就是 Maddy 的位置, 结束.
- 然后看 Maddy 在目标位置处, 轻微扭动一下位置, 是否能避开这次碰撞; 如果成功, 那么这就是 Maddy 的位置, 结束.
- 所有位置都尝试失败后, 则判定为躲避失败, 结束.
- 无论以何种方式结束, 都重新设置 `data.Pusher.Collidable = false`.

详见 [第 9.4.2 小小节](#).

如果这个扭动没能成功避开碰撞, Maddy 死亡, OnSquish 结束. 否则继续后面的判定 (注意此时 `data.Pusher.Collidable = false`).

- (3) 如果经历过步骤 (1), 且 `TrySquishWiggle(data, 3, 5)` 判定成功, 且位置允许, 那么解除蹲姿 (由于 `data.Pusher.Collidable = false`, 可能因此卡进 Pusher 里). 否则什么都不做.

以上所有改变 Maddy 位置的行为, 都是直接设置位置, 而不是再去调用 `MoveH/V`, 因此是有机会穿墙的! 以下称这种利用 `OnSquish` 穿墙的动作作为隧穿, 详细的讨论见 [第 9.4.1 小小节](#).

能够在 (1) 直接结束的情形, 包括上下挤压至蹲姿, 水平挤压但蹲姿不被推动, 水平挤压蹲姿被推动但不挤压等 (见 [图 16](#), [图 17²⁰](#), [图 18²¹](#)).

步骤 (2) 主要体现为卡脚修正, 还有挤压死 Maddy.

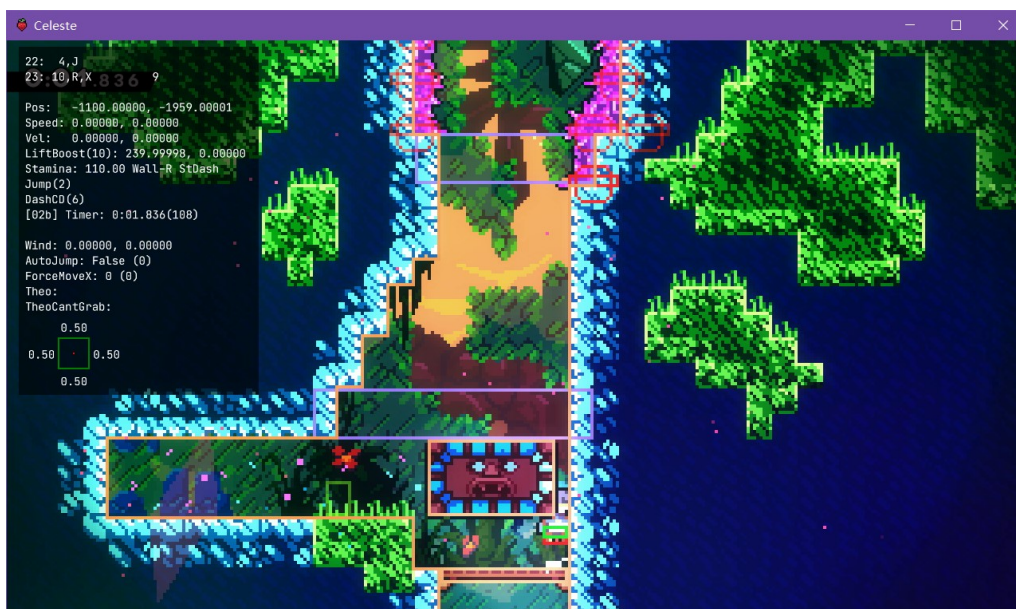


图 16: 蹲姿不被推动

于是, Maddy 与固块的交互 (准确来说是移动块的更新带来的, 而非 Maddy 的更新涉及到移动块带来的) 带来了四类移动: (a) 被推动 (`MoveHExact`) (b) 被挤压 (`MoveHExact` 跳转到 `OnSquish`) (c) 被带动 (`Ride`) (d) 被修正

²⁰演示 TAS: [SquishToCrouch2.tas](#)

²¹演示 TAS: [SquishToCrouch3.tas](#)

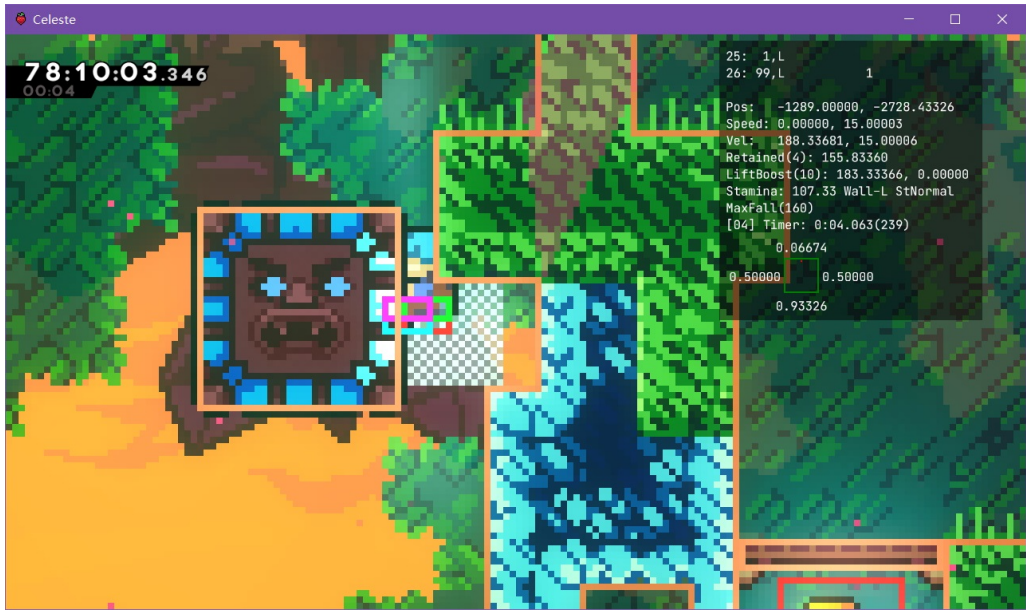


图 17: 蹲姿被推动但不挤压



图 18: 蹲姿被推动后仍挤压, 再修正

(向下 1 px). 此外还获得了 `LiftSpeed`(`Actor` 在设置 `liftspeed` 时, 会自动产生一个 10f 的 `liftSpeedTimer`).

9.4.1 隧穿

常规固体的碰撞箱至少有 8×8 , 仅靠 $[-3, +3] \times [-5, +5]$ 的瞬移是不能越过的. 而使用 `TargetPosition`, 则能产生更多的位移, 有机会隧穿.

- 第一种情形是移动块仅水平/竖直位移起作用, 此时需要速度足够高 (至少 410), 才能前一帧还在障碍物与 `Maddy` 之前, 这一帧已经盖到 `Maddy`, 而前端能越过障碍物 (或者加上 3 px / 5 px 的修正之后能越过).
- 第二种情形是, 在移动块移动前就将 `Maddy` 卡进移动块内部, 则此时移动块的长度是越过障碍物的主要动力. 例如先在移动块 `MoveH` 中, 迫使 `Maddy` 进入蹲姿并 `Wiggle`, 然后解除蹲姿卡进移动块内部. 然后在移动块 `MoveV` 时完成隧穿 (需要移动块竖直速度不能太大, 否则无法在第一次 `MoveH` 进入蹲姿并 `Wiggle` 的情况下, 保证第二次 `MoveV` 时进入蹲姿后还与该移动块碰撞). 或者利用边界 `EnforceBounds`.

移动块就像一个入口被封死的单向隧道. 正常情况下你没法利用它越过障碍物. 但一旦你卡进了它的内部, 则可以顺着隧道移动到出口处. 或许也有一点像利用虫洞进行跃迁, 不过这好像不是一个很好的比喻...

9.4.2 `TrySquishWiggle` 的细节

具体来说, 是检测 `Maddy` 原位置的 $[-3, +3] \times [-5, +5] \setminus \{(0, 0)\}$ 范围, 以及 `TargetPosition` 的 $[-3, +3] \times [-5, +5] \setminus \{(0, 0)\}$ 范围内, 是否有一点使得 `Maddy` 不与任何固体碰撞 (包括 `data.Pusher`). 如果有, 则将 `Maddy` 的位置直接设置为这个点.

`TrySquishWiggle` 的优先顺序参考 图 19.

`TrySquishWiggle` 的范围抹去了 $\{(0, 0)\}$, 可能是考虑到步骤一中已经检测过了. 即使没检测过, 但对于 `Maddy` 原位置处检测碰撞也是没有必要的. 而 `TargetPosition` 处确实有可能需要检验一下. 不过对于移动块, 除非速度足够快 (至少要 410, 或者原先就已卡在移动块中且不进入步骤一 (对于隧穿的 `MoveH` 蹲姿 `Wiggle` 法, 这不可能不进入步骤一 (除非死亡 图 20). 利用边界 `EnforceBounds` 可以), 否则我们不太能见到这个问题.

9.5 `JumpThru`

单向板的移动则是写在 `JumpThru` 里的. 大体写法与 `Solid` 差不多, 但是没有横向碰撞, 也就是 `MoveHEXact` 里只对骑乘的对象进行移动, 且不产生 `LiftSpeed`. 而纵向则是若单向板上升, 则计算骑乘与碰撞. 若下降, 只计算骑乘. 上升或下降都产生 `LiftSpeed` (依然, 是两个分量都有的).

呃, `Kevin` 块能把人物挤过 `JumpThru` 应该也与这有关. 不, 就是因为 `onSquish` 只检测 `Solid` 不检测 `JumpThru`.

移动块每帧移动整数个像素 `Platform` 类有 `movementCounter`, 但没有 `Speed`. `Solid` 类有 `Speed`, `Solid.Update()` 是 `MoveH(Speed.X * Engine.DeltaTime); MoveV(Speed.Y * Engine.DeltaTime);` `Solid/JumpThru` 和其他实体交互体现在 `MoveHEXact/MoveVEXact`, 因此对实体产生的推动是整数个像素. `JumpThru` 类没有 `Speed`, `JumpThru` 类底下有 `RotatingPlatform`, `SinkingPlatform`, `MovingPlatform` 等, 都各自重载了 `Update()` 以实现不同的移动.

`Platform.Update()` 会将 `LiftSpeed` 归零.

	-3	-2	-1	0	1	2	3
-5	76	54	32	10	30	52	74
-4	72	50	28	8	26	48	70
-3	68	46	24	6	22	44	66
-2	64	42	20	4	18	40	62
-1	60	38	16	2	14	36	58
0	56	34	12	N/A	11	33	55
1	59	37	15	1	13	35	57
2	63	41	19	3	17	39	61
3	67	45	23	5	21	43	65
4	71	49	27	7	25	47	69
5	75	53	31	9	29	51	73

图 19: TrySquishWiggle 的优先顺序, 越小越优先.

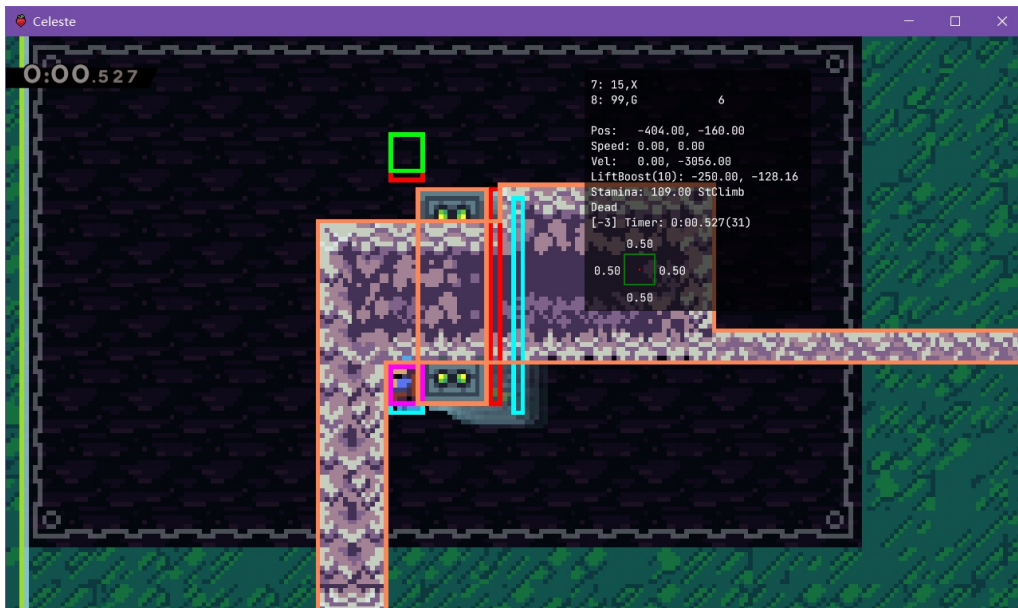


图 20: Maddy 以死亡为代价, 表明 TrySquishWiggle 的位移范围不包含原点.

对于使用 Platform.MoveH/MoveV 来更新的, $LiftSpeed.X/Y = moveH/V / Engine.DeltaTime$. (注意 Platform 类本身在更新时并不使用 MoveH/MoveV). 因此对于 Solid 类, $LiftSpeed = Speed * Engine.DeltaTime / Engine.DeltaTime = Speed$.

JumpThru 类本身不产生 LiftSpeed, 只重载了 MoveHEXact/MoveVEXact, 使得后者能将 LiftSpeed 赋予 Player. JumpThru 下属的 MovingPlatform 有一个 tween 组件, 这个组件更新时会调用 MovingPlatform.MoveTo = Platform.MoveTo, 而 MoveTo 就是包装了的 MoveH + MoveV, 于是会产生 LiftSpeed. 又由于 MoveV 下面是 MoveVEXact, 因此能将产生的 LiftSpeed 赋予 Player. SinkingPlatform 本身的更新会用到 MoveV, 于是既产生也赋予 LiftSpeed (下降段会因为 $LiftSpeed < 0$ 而 $LiftBoost = 0$, 信息面板对此是不显示的).

简言之, Platform.MoveH/MoveV 赋予固块 LiftSpeed, Solid 的 MoveHEXact/MoveVEXact 和 JumpThru 的 MoveVEXact 再将自身的 LiftSpeed 赋予 Player, 各类 Solid 和 JumpThru 的移动都以直接或间接的方式调用了 Platform.MoveH/MoveV, 因此确实产生 LiftSpeed.

Moving jumpthrough platforms don't give the player any lift boost speed when they're only moving horizontally. More specifically, they give the player both horizontal and vertical lift speed in MoveVEXact(), but nothing at all in MoveHEXact(). This means that you'll only get a horizontal boost from moving jumpthrus if the platform is also moving vertically, which feels quite inconsistent.

In particular, you will get a boost from a horizontally moving platform in Resort/Ridge if you jump right as you land on the platform, but not if you wait a moment before jumping - this is because the platform moves down slightly when you step on it, and you'll only get boosted during this short period. This also affects some very particular TAS scenarios involving jumpthrus attached to moon blocks, where the lack of a lift boost while standing on a jumpthru prevents some optimizations from being possible.

呃呃, 正是由于常规情况下横向移动不给 liftboost, 但踩下去导致 MoveVEXact 会产生 liftboost, 使得横向移动的单向板的手感很怪.

9.6 Koral clip

官图 6A.tas [02b]

在第 11 帧, 在 Kevin 块更新 (水平运动) 的时候触发 OnSquish, 依次判定:

1. 步骤一, 原先不在蹲姿且不在攀爬, 因此进入蹲姿, 挤压依然发生, 继续判定.
2. 步骤二, TrySquishWiggle, 向下移动 1px 后在蹲姿的状态下成功避开, 此时 Pusher.Collidable = false.
3. 步骤三, 经历过步骤一, 且由于 Pusher.Collidable = false, 可以 UnDuck, 因此 UnDuck.

于是卡进了 Kevin 块但没死.

第 12 帧, 还在 StDash, 在 Maddy 的竖直运动判定 OnCollideV, 判定为 Kevin 块改为向上运动, Maddy 结束 StDash 进入 StNormal.

由于在 OnSquish 我们说过, Pusher 本身不能致死, 因此一直没死.

在向上冲刺的第 7 帧, Kevin 块开始运动, 按照正常计算出的 moveV, 应将 Maddy 恰好推至 Kevin 块顶上, 于是就这么干了. 此过程没有与其他固块发生碰撞, 因此没有调用 OnSquish.

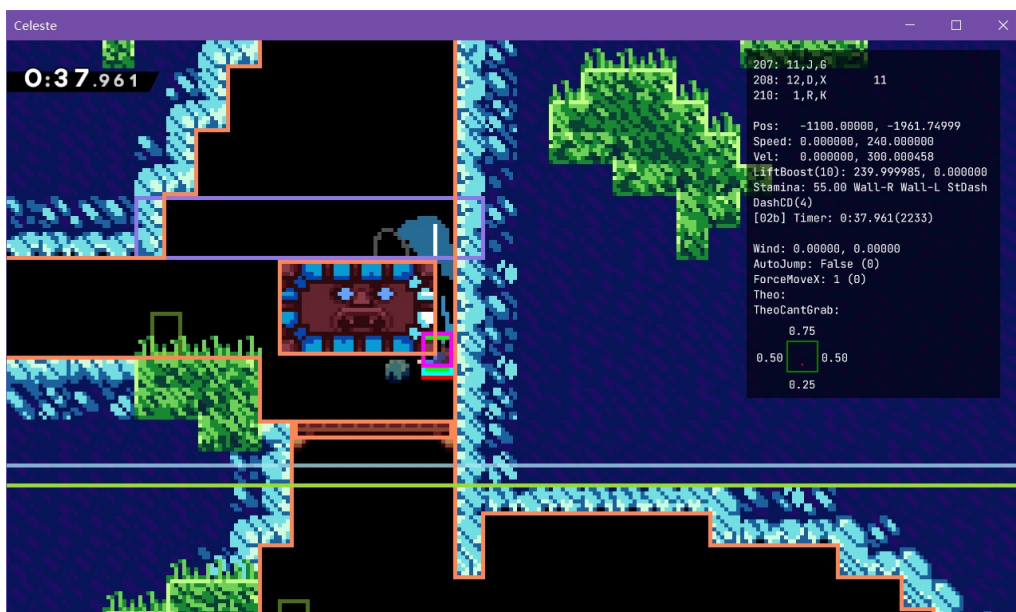


图 21: 没死

类似的, 在图 22²² 中, 卡好位置使得进入 OnSquish 的第三阶段, 并成功解除蹲姿, 卡进 MoveBlock 里. 下一帧则 OnGround 而恢复冲刺, 而面对 MoveBlock 的继续挤压通过 OnSquish 第一阶段的进入蹲姿而结束. 这样就在不可能的地方恢复了冲刺!

需要注意的细节: 因为要进第三阶段, 所以必须经历第一阶段, OnSquish 那一帧不能是 StClimb, 且进入蹲姿后不能已经结束. 此外, 还需要注意在 OnSquish 之前还可能会有移动块同向运动的 MoveV(1f) 修正.



图 22: OnSquish 恢复冲刺

²²演示 TAS: 4A jumpleless.tas

10 LiftBoost

可归结为三步: 移动块自身获得 LiftSpeed, Player (被动/主动地) 从移动块获得 LiftSpeed, Player 从 LiftBoost 获得速度加成.

10.1 移动块自身获得 LiftSpeed

见上, 或许我们应该把那个板块的这部分内容移动过来?

10.2 Player 从移动块获得 LiftSpeed

大部分情形下, 都是由移动块将 LiftSpeed 赋予 Player (即来自于移动块本身的更新)

(以下是针对 Maddy 简化过的机制)(非 Maddy 的 Actor 兴许或有 LiftSpeedGraceTime = 0f, 则机制会略有区别.)

与 Platform 碰撞/骑乘获得 currentLiftSpeed, 见 Solid/JumpThru 章节. / 传送带 (学名 wallbooster (不要和 wallboost 机制搞混!)/俗名 conveyer) / walljump (以前版本有 walljump 朝向的 bug, 现在应该没了). 如果这些都没发生, 则 current = 0 (Actor.update 中).

如果 current 不为 0, 则将 LiftSpeed 设为 current 并重置 liftSpeedTimer. 对于移动块碰撞/骑乘, liftSpeedTimer = 10f. 但对于传送带/walljump, liftSpeedTimer = 9f (这是因为更新顺序是: 状态机 → liftSpeedTimer → 移动块更新).

liftSpeedTimer ≤ 0 时 LiftSpeed 设为 0.

LiftSpeedcap 住得到 LiftBoost. 范围是水平 [-250, +250], 竖直 [-130, 0].

但除此之外, Player 还有一种极为特殊的方式获得 LiftSpeed, 那就是 WallJump. 在 WallJump 中, 有

```
private void WallJump(int dir){
    ...
    if (base.LiftSpeed == Vector2.Zero){
        Solid solid = CollideFirst<Solid>(Position + Vector2.UnitX * 3f * -dir);
        if (solid != null){
            base.LiftSpeed = solid.LiftSpeed;
        }
    }
    ...
}
```

这里, 踢右墙则 dir = -1, 踢左墙则 dir = 1. 这个机制是 CassetteBoost 成立的基础. (也适用于其他移动块!)

10.3 生效: Player 从 LiftBoost 获得速度加成

在 liftSpeedTimer 期间, 通过以下操作, 会获得 LiftBoost: 冲刺 (仅限于从 StNormal 和 StClimb 进入. 特别的, StDash 里 (超冲) 是吃不到的), 跳跃 (包括普通跳 jump/抓跳 climbjump, 踢墙跳 walljump, 蹭墙跳 wallbounce

(源码里叫 SuperWallJump), wallboost, 以及 super/hyper), 以及通过松开抓键/体力耗尽退出 StClimb, 走下上升移动块的边缘, 离开传送带, 以及其他一些情形 (以上操作是所有列在 Maddy 自身的更新里的, 其他的比如岩浆块第四阶段会将 Maddy 速度设为 LiftBoost, 这些我尚未一一翻遍).

(walljump 获得 LiftSpeed 在判定 LiftBoost 之前, 所以可以一帧内直接吃到在移动块上 walljump 的 LiftBoost. 当然, 也可以是原先就有 LiftSpeed, 然后这帧 walljump 吃到 LiftBoost)

LiftBoost 的叠加方式基本可以总结为: 先确定基础速度 (对于 jump/climbjump, 是原水平速度 +40*moveX, 对于 walljump/wallbounce/wallboost, 是直接设置为固定值. 而竖直速度则都是设置为固定值. 对于松抓/体力耗尽退出 StClimb, 速度不变.), 然后再加上 LiftBoost. 但例外是 hyper, 冲刺以及走下上升块的边缘.

对于 hyper. Hyper 的速度是通过在 super 的基础上乘以因子 (1.25, 0.5) 得到的, 看源码可知这在 LiftBoost 的后面! 因此 hyper 时 LiftBoost 也会乘以因子.

对于冲刺, 由于冲刺第一帧速度为 0, 因此是将 LiftBoost 叠加到原先速度 beforeDashSpeed 上. 然后第 5 帧用 beforeDashSpeed 来做冲刺速度的判定²³. 这会影响一些边界情况: 比如原先速度为 220, LiftBoost 为 40, 进行水平冲刺, 那么得到的速度是 $220 + 40 = 260$ 而非 $240 + 40 = 280$. 由于 beforeDashSpeed 不影响冲刺竖直速度, 因此 LiftBoost 的竖直分量无法叠加至冲刺.

对于走下上升移动块的边缘, 具体的判定条件是: StNormal, LiftBoost 朝上, 速度不朝上, 上一帧在地面而这一帧不在. 则会把竖直速度设置为 LiftBoost.Y.

```
private int NormalUpdate(){
    if (LiftBoost.Y < 0f && wasOnGround && !onGround && Speed.Y >= 0f)
    {
        Speed.Y = LiftBoost.Y;
    }
}
```

对于离开传送带, 具体的判定条件是

```
if (!CollideCheck<Solid>(Position + Vector2.UnitX * (float)Facing))
{
    if (Speed.Y < 0f)
    {
        if (wallBoosting)
        {
            Speed += LiftBoost;
            Play("event:/char/madeline/grab_letgo");
        }
    }
}
```

以上可知竖直 LiftBoost 没法叠多个.

如图, 将在蹭墙跳瞬间获得 $170(\text{wallbounce})+250(\text{LiftBoost})$ 的速度. 这里也应用了刺的伤害箱会晚 1f 从而能

²³因此在移动块上进行上/下冲的水平速度为 0.

吃到 LiftBoost.

将在踢墙跳的瞬间获得 $130(\text{walljump})+250(\text{LiftBoost})$ 的速度

Spring moon boost (国内 TASer 戏称为春月加速):

$130(\text{jump})+250(\text{月球块瞬间弹回给的 LiftBoost})+$ 后续可能的冲刺还能吃 $250(\text{LiftBoost})$

似乎是带弹簧的月球块在弹簧被击中的瞬间, 会回到初始位置再开始被击退. 于是首先自己冲向月球块或弹簧, 使之往后退. 在合适的时间点再碰弹簧 (对于斜上冲向月球块的, 基本上是往上爬碰弹簧; 对于冲向弹簧的, 基本上是再横冲碰弹簧 (由于冲刺 CD, 这要求前一个冲刺在较远处开始)), 则碰弹簧瞬间将月球块瞬移回原点 (实际上是设置了月球块的 dashEase 与 dashDirection, 真正的月球块移动还要等到月球块自身更新时才发生), 然后月球块与 Maddy 发生碰撞给出 LiftBoost, 然后就获得 $130+250$ 的速度. 注意弹簧给的速度始终是 240, spring moonboost 的最终速度与之无关. 只不过需要利用弹簧使得月球块瞬间回弹.

月球块最大回弹 8px, 弹簧宽 6px, 触发弹簧时月球块瞬间移动 7px, 要想被月球块推动, 必须恰好把月球块推到最深处. 假如还有风 (且晚于 Maddy 更新), 则还必须亚像素 + 被风推动 $< 1\text{px}$.

11 蹲姿

源码中 Duck 和 Crouch 相关.

Ducking 改变的同时改变碰撞箱.

如果 Maddy 在蹲姿中, 且位置无法取消蹲姿, 但却没按下, 似乎她会试图自动挤出来? (至少在墙角边缘的时候....)

Demodash 必须在开始前处于不在蹲姿中, 因此连续 dd 穿刺的时候必须想办法在中间解除蹲姿. (例: 9.tas 的 lvl_g-02, 2022/11/19 版)

在 Player.Update() 中 (在 base.Update() 之后, 移动位置之前), 需要没有狼跳时间且竖直速度严格大于 0 才会解除蹲姿.

在 DashBegin() 中可以进入蹲姿, 大部分半空中的斜下冲会在第 5 帧由于竖直速度大于 0 且狼跳时间消失而在 base.Update() 之后立刻解除蹲姿. 然而, 在一些特殊的情况, 你可以稍微保留这个蹲姿一会儿: 见 7A [d-10] 2023/01/09 图 23. 这里利用了进入蹲姿后, ExitBlock 会显形的特性, 使得蹲姿斜下冲了几帧! (第 6 帧由于运算顺序, 此时还不能解除蹲姿, 然后再是移动位置 (视觉上可以解除蹲姿). 在第 7 帧才真正解除. 同理, 如果在掉落块等向下移动的方块的下方 (斜) 下冲, 那么在你被掉落块压死期间, 也会保持蹲姿. 如 Kevin 块 240 的速度就能一直压着你. 当然, 你也可以在狼跳时间内进入斜下冲, 这样在狼跳期间也会保持蹲姿.



图 23: 蹲姿斜下冲

在 ClimbBegin 与 ClimbUpdate 中, 蹲姿不会解除. 且如果不向下爬的话, 也不会触发 Player.Update() 里最常用的那个蹲姿解除. 因此如果你很奇特的以蹲姿进入了 StClimb, 你可以在此期间一直保持它. 在 NormalUpdate() 里进入蹲姿的分支是无法进入 StClimb 的, 但是 OnCollideV 里 DashDir.Y > 0f 的那个进入蹲姿是在 NormalUpdate() 外面的, 且 OnCollideV 在 NormalUpdate 之后. 因此可以 NormalUpdate 进入 StClimb 并 OnCollideV 进入蹲姿. 案例: 用 Spinner Stun 路线以前的 7B [e-01].

在 StNormal 中, 当你在地面上处于蹲姿却未按下键时, 会尝试 UnDuck. 如果成功则 UnDuck, 否则检测是否微

移后可以 UnDuck, 可以的话就先微移.

```
if (Ducking){
    if (onGround && (float)Input.MoveY != 1f){
        if (CanUnDuck){
            Ducking = false;
            Sprite.Scale = new Vector2(0.8f, 1.2f);
        }
        else if (Speed.X == 0f){
            for (int num = 4; num > 0; num--){
                if (CanUnDuckAt(Position + Vector2.UnitX * num)){
                    MoveH(50f * Engine.DeltaTime);
                    break;
                }
                if (CanUnDuckAt(Position - Vector2.UnitX * num)){
                    MoveH(-50f * Engine.DeltaTime);
                    break;
                }
            }
        }
    }
}
```

参考 MyTASRef 演示视频/UnDuck50(1).mp4 与 MyTASRef 演示视频/UnDuck50(2).mp4

Delayed ultra 可以用来在这种地形 图 24 恢复冲刺. 下一帧通过 wallboost 或者右跳来离开刺.



图 24: 用 delayed ultra 恢复冲刺

如 图 25, 即使有狼时间, 下一帧 LJ 也会因为 wallSlide 而解除蹲姿, 改为 LDJ 可保持蹲姿. 如果是 J 或 RJ 则不会触发 wallSlide, 蹲姿正常地保留着.



图 25: wallSlide 导致的蹲姿解除

12 AutoJump, forceMoveX

JumpGraceTime = Coyote frames = 6f

VarJumpTime = 最普通的跳跃后可以按住跳键保持纵向速度不变的时间 = 12f

这种机制在源码里叫 Variable Jumping, 只在 NormalUpdate 里面出现. 用 varJumpTimer 控制时长, 可以被 AutoJump / 按住跳键触发. 如果一旦在 NormalUpdate 中却没成功触发, 则计时归零.

大部分情况下, AutoJump 机制很简单, 就是将 AutoJump 设为 true, 设置 varJumpTimer, 并令 AutoJumpTimer = 0. 这种情况下, varJumpTime 就是一段长度固定的保持着最高竖直速度的跳跃过程.

并且, 我们知道 -40~40 的时候按住跳键, 则重力减半 (变为 7.5). 实际上 AutoJump 也同样会有重力减半的效果. 并且一般来说 (AutoJumpTimer = 0), AutoJump 并不会自己消失 (除非有另一个跳 / 落地 / 抓墙等). 因此不被打断的情况下, 在达到 MaxFall 之前, AutoJump 引起的竖直方向上的运动是完全确定的, 无法被玩家输入改变的.

唯一例外是 Bounce, 此时 AutoJumpTimer = BounceAutoJumpTime = 6f, 而 varJumpTimer = BounceVarJumpTime = 12f. 也就是说有 6f 速度为-140, 剩下 6f 可以手动操作选择保持在-140, 也可以不按跳让其减速. 在 varJumpTimer 结束后, 也可再手动操作选择是否要 -40 40 的半重力效应.

如果在 varJumpTimer > 0 时冲刺, 由于冲刺过程不覆盖跳跃的数据, 而冲刺时长会小于部分 varJumpTimer (例如蹭墙跳 15f), 因此可以有 DashBounce, 在冲刺结束后重新获得跳跃的竖直速度!

也可以选择打断冲刺回到 StNormal, 这样就可能在冲刺后还有 varJumpTimer.

StPickup 是一个 12f 的过程. 在这个过程中, 速度和 VarJumpTimer 会被冻结住 (第一帧还是会走的), 结束后恢复, 竖直速度如果向下则重置为 0. 但其他 Timer 在这个过程中都是走的, 例如游戏计时器, DashCD, etc.

因此可以在还有 4 帧跳跃时冲刺 (消耗 2f), 然后抓物体打断冲刺 (消耗 1f), 最后回到 StNormal 还有 1f 跳跃, 就可以恢复竖直速度.

e.g. 得到了水平冲刺速度 240+ 弹簧竖直速度-185



图 26: 呃暂时懒得起名了

(这里不需要是 AutoJump, 按住跳键也同样能触发)

forceMoveX 是一个类似 AutoJump 的机制, 顾名思义它会将输入的方向覆盖成 forceMoveX. 只不过它总是只在 forceMoveXTimer > 0 时生效, 并不会像 AutoJump 那样可能一直持续下去. 在例如 WallJump 时, 如果同时键入的方向 MoveX 非零, 那么 forceMoveX 就会设置成与墙相对的方向, 时长跳跃 1f + forceMoveXTimer 的 $\text{Ceil}(0.16*60f) =$ 共 11f.

13 Holdable

Holdable 并不是只有在 `minHoldTimer <= 0` 才可以扔, 在一些条件下也可以蹲跳甩掉. 后者可以使得水母平 u 的应用场景更加广泛

13.1 平 u 接水母并瞬间丢弃水母

假设在合适的地面上有水母, 我们容易写出这样的操作:

```
14 | L,D,X
    |
  1 | R,J
    |
  6 | R,D,X
    |
  1 | G
    |
 12 |
    |
 99 | R,D,J
```

这获得了 (421.67, -105.00) 的速度并瞬间丢弃水母, 无视了 `minHoldTimer`, 在一些需要立即再接 ultra 的情形很有用.

其原理是

```
private int NormalUpdate(){
    ...
    if (Holding != null){
        ...
        if (!Ducking && onGround && (float)Input.MoveY == 1f && Speed.Y >= 0f && !holdCannotDuck){
            Drop();
            Ducking = true;
            Sprite.Scale = new Vector2(1.4f, 0.6f);
        }
        ...
    }
    ...
}
```

但是, 有的情形下, 我们会发现这样并不能成功, 需要把 12 改成 13 (或者更好一点, 12; 1,R), 这是为什么呢? 关键在于 `holdCannotDuck` 变量. 这个变量的作用是, 当你手持可以缓降的手持物品, 按住下, 速降至地面时, 不会自动将其下放 (原版中水母可缓降, Theo 水晶不能缓降). 这个变量并不是实时计算的, 而是在以下三种情况下发生赋值:

- (1) `NormalUpdate()` 中, 在地面上, 有手持物²⁴, 不按下, 且 `holdCannotDuck = true`, 且要么不是蹲姿要么速度严格朝上. 那么 `holdCannotDuck = false`; (简单说, 按住下落到地面被锁在 `holdCannotDuck` 这个状态后, 一旦不再按下, 就“解锁”. 最后一个条件是避免触发其他 `Drop()`.)

²⁴时机是 `NormalUpdate()` 最开始. 因此可以在触发 `Throw()` 的同时触发这一项.

(2) NormalUpdate() 中, 不在地面上, 且有可以缓降的手持物品²⁵. 那么 holdCannotDuck = (float)Input.MoveY == 1f;

(3) PickupCoroutine 的最后一帧, 在地面上, 手持物品可缓降, 按下. 那么 holdCannotDuck = true.

明显看出以上都是在有手持物品的情况下才会发生的. 因此, 若是上一次手持物品以 holdCannotDuck = true 的状态离开 (比如空中下放水母的前一帧也按着下), 那么新一次手持物品的时候, 就需要 StPickUp 结束后, 额外在 NormalUpdate() 多等一帧 (不能按下), 然后才能蹲跳甩开水母.

13.2 另一种 NormalUpdate() 中的 Drop()

承接上文,

```
if (!Ducking && onGround && (float)Input.MoveY == 1f && Speed.Y >= 0f && !holdCannotDuck){
    Drop();
    Ducking = true;
    Sprite.Scale = new Vector2(1.4f, 0.6f);
}
else if (onGround && Ducking && Speed.Y >= 0f){
    if (CanUnDuck){
        Ducking = false;
    }
    else{
        Drop();
    }
}
```

后面这个 Drop() 的用意不明. 看起来主要起作用的反而是 Ducking = false, 使得你斜下冲抓水母打断撞地后, 只能维持一帧蹲姿 (从而渲染效果不太奇怪?) Drop() 看起来只是为了保证不出 bug 而设置的保险.

我们是有办法避开它, 以实现蹲姿抓水母的: 斜下冲, 抓水母打断, 撞地的前一帧速度朝下且正好在地面上, 撞地这帧即判定 onGround = true 并获得 jumpGraceTimer, 同时这帧由于移动块等被带离地面, 再下一帧狼跳.

其他的蹲姿抓水母方法: 注意到 StClimb 下不需要丢掉手持物品就可以直接冲刺, 而 DreamDash 下可以直接以进入 StClimb 结尾, 因此直接带着水母进果冻反抓, 进入 StClimb, 再 dd 进果冻, 出来再反抓, 则蹲姿抓水母且 StClimb.

13.3 Offgrid Holdable

Holdable 在搬运期间, 位置由 Player.UpdateCarry() 中 Position + carryOffset + Vector2.UnitY * Sprite.CarryYOffset 给出位置.

²⁵时机是 NormalUpdate() 中所有丢弃物品的操作之后.

- carryOffset 常态下是 Vector2(0f, -12f), 而在 StPickUp 下则是一段动画 (给 Player 加上了一个名为 tween 的 component), 不是整数. 投掷 Drop/Throw 可以通过 NormalUpdate, Die, BoostBegin, BadelineBoost-Launch, DoFlingBird, StartCassetteFly 实现. 因此要打断 StPickUp 来获得 offgrid Holdable,
 - 可以通过除了 NormalUpdate 的方式实现.
 - 可以通过其他手段打断 StPickUp 进入 StNormal (比如跳过剧情) 然后再扔. 注意 tween 这个 component 还在, 并不随着 PickupCoroutine 结束而结束.
 - * 如果等待 minHoldTimer 结束再扔, 则投掷时 tween 已经播放完毕. 无法达成预期效果.
 - * 其他两种 NormalUpdate 中 OnGround 情况下的 Drop() 都是可以达成的.
- Sprite.CarryYOffset 则是这样的, PlayerSprite 会有一堆元数据, 给出在哪个动画状态下的第几帧的 PlayerAnimMetadata, 这其中标记了 int CarryYOffset, 而 Sprite.CarryYOffset 则是由当前对应的 PlayerAnimMetadata 的 CarryYOffset 再乘上 Scale.Y 给出. 因此 off grid 依靠的是 Sprite.Scale.Y 不是整数. Player 在 UpdateSprite 中会

```
Sprite.Scale.X = Calc.Approach(Sprite.Scale.X, 1f, 1.75f * Engine.DeltaTime);
Sprite.Scale.Y = Calc.Approach(Sprite.Scale.Y, 1f, 1.75f * Engine.DeltaTime);
```

而平常的一些跳跃, 落地等, 则会改变 Sprite.Scale, 两者结合起来使得 maddy 有一种果冻般的质感. 最常见的 Jump(..) 会设置 Sprite.Scale = new Vector2(0.6f, 1.4f), 这给出 13 帧非整数的 Sprite.CarryYOffset, 只需要在这段期间扔出 Holdable, 就可以获得 offgrid Holdable.

然后我们知道 offgrid 的东西对 SolidTiles (Grid) 的表现, 与对其他 Solid/JumpThru 的表现, 是不一样的, 这就导致了 <https://discord.com/channels/403698615446536203/519281383164739594/1122942945247641683>.

具体来说, Rect 碰撞 grid 是先将 Rect 转化为由一个个单元格 (cell) 组成的矩形 (SolidTiles 用的是 8*8 的 cell), 然后再看这个每一个 cell 上是否是砖块. 游戏在转化过程中, 计算右侧/底部时先 -1f, 使得紧贴右侧/底部并不算碰撞. 这样做, 实际上等效于, 将 Rect 的宽/高都 -1 + eps, 然后将 cell 视为 8*8 的矩形, 检测两个矩形是否相交 (相触不算). 不过若是 Rect 本身都是整数坐标, 则无需 -1 + eps, 就用直观的整点矩形是否碰撞即可.

而 Hitbox 碰撞 grid, 则是用 Hitbox.Bounds 给出的矩形来检测 rect collide grid. 其中, Hitbox.Bounds 的左侧, 顶部, 宽度, 高度是由 Hitbox 的数据直接转成 int 给出的可惜不是右侧, 底部被强转成 int, 否则我们会穿过 7px 的竖直缝, 5px 的水平缝. 这就导致, offgrid 陷入卡进墙里的情况随坐标而变化 (以 CelesteTAS 修复过的 Draw.HollowRect 为标准来讨论. 原版是不会渲染出卡进墙里的效果的). 当 AbsoluteLeft < 0, 则修复版 HollowRect.x 固定向下取整, 而 int 强转效果上是向上取整, 所以导致卡进左墙. 当 AbsoluteLeft > 0, 则 HollowRect.cw 比往常宽 1px, 而 Hitbox.Bounds.Width 正常, 因此卡进右墙. 当 AbsoluteTop < 0, 卡进天花板. 当 AbsoluteTop > 0, 卡进地板.

CelesteTAS 中稍微修复了一下 Draw.HollowRect, 使得 offgrid Hitbox vs normal Hitbox 的表现是符合直观的 (呃, 当时可能主要是 player normal hitbox vs puffer offgrid hitbox 之类的?)(具体来说, fx = Floor(x), cw = Ceiling(width + x - fx), 同理 fy, ch, 然后 orig(fx, fy, cw, ch, color)). 不过这样修复的话, 对 offgrid Hitbox vs offgrid Hitbox 应该也会有 bug? 但这种情形也太呃呃了.

13.4 水母

反扔 + 70

速度机制

在 IsTired 情况下不能抓 ($\text{IsTired} \Rightarrow \text{CheckStamina} < 20f$, 这里 $\text{CheckStamina} = \text{Stamina} + (\text{wallBoostTimer} > 0f ? 27.5f : 0f)$). 特别的, 你可以最后一次使用抓跳, 然后抓取水母, 然后 wallBoostTimer 结束开始闪红.)(IsTired 同样用于渲染闪红, 这样写可能是为了保证渲染效果)

单水母无限冲刺最快向上爬升 (要求每个循环之后的水平位置大体不变):

垂直: 34,G; 11,D; 10,U,X 比较接近最快, 效率是 $86.25 \text{ px} / 52 \text{ f} = 1.659$.

左右斜上: 比较慢

14 运算顺序 Order of Operation

14.1 深度机制

actualDepth 是如何影响更新顺序的

14.2 EntityList.UpdateLists()

- (1) 将 toAdd 加入列表. 会同时对 TagLists, Tracker 添加此实体, 调用 entity.Added(Scene scene) 使得其 Components 被添加入 Tracker, 并设置 entity 的实际深度. 注意, Components 只是加入 Tracker, 而 Scene 里本身并不存储这些信息. 如果 toAdd 非空则标记 EntityList 为 unsorted.
- (2) 将 toRemove 移出列表. 会调用 entity2.Removed(Scene) (将 Components 从 Tracker 移除并将 entity.Scene = null), 再对 TagLists, Tracker, Engine.Pooler 移除此实体.
- (3) 若 unsorted 则对实体依照 actualDepth 排序.
- (4) 对 toAdd 中的实体调用 entity.Awake(Scene scene). 这一般是需要用到场景信息, 或者需要再往场景加入实体的一些初始化.

注意: 我们并没有按照实际深度来加入 Tracker, 而是加入 Tracker 的同时设置实体的实际深度. 因此, Tracker.GetEntities<T>() 里面的顺序, 只能保证, 若实体 a, b 都是 T 的实例 (它们有可能实际上来自于 T 的一个子类, 取决于是否标记 Tracker[false]), 且 a, b 具有同样的深度 (特别的, 具有完全相同的类型), 那么它们在 Tracker.GetEntities<T>() 中的顺序与它们 **初始** 的 actualDepth 的先后顺序相同.

于是下面这些在 Tracker 中的顺序我们是很难知道的:

- Tracker.GetEntities<T>() 中的 a, b, 其中 a, b 深度不同.
- Tracker.GetComponents<T>() 中的 a, b, 其中 a, b 来自实体 A, B, 且 A, B 深度不同.

特别的, 对于

```
foreach (PlayerCollider component2 in base.Scene.Tracker.GetComponents<PlayerCollider>()){
    if (component2.Check(this) && Dead){
        base.Collider = collider;
        return;
    }
}
```

这并不是按照实际深度的顺序来排列的!

根据 [第 17.30 小节](#), 如果都是在 Level.LoadLevel(..) 里 foreach (EntityData entity in levelData.Entities) 那个大循环里被添加进来的话, 它们的顺序就是.bin 文件中的顺序.

14.3 深度

Monocle.EntityList.UpdateLists() 里, 进行了 entities.Sort(CompareDepth), 因此会按深度更新/渲染. CompareDepth 比较的是 double 类型的 Entity.actualDepth.

实体自带一个 int 类型的私有字段 depth. 每当对公共属性 Depth 进行赋值, 若场景非空则会调用 Scene.SetActualDepth(this) 而 Entity.Added(Scene scene) 也会调用 Scene.SetActualDepth(this). 而 Entity.Added(Scene scene) 会在 EntityList.UpdateLists() 中在添加实体时用到.

Scene.SetActualDepth(Entity entity) 的机制是, Scene 有个字典 actualDepthLookup, 查询 entity.depth (int 类型的那个), 得到一个 double. 每次查询, 都会给当前这个 int 对应的值 value 增加约 1E-6. 而实体的 actualDepth = (double)entity.depth - value. 第一次查询该 int 的时候, actualDepth = entity.depth. 此外, 设置完实际深度后, 还会将 Scene.Entities 标记为 unsorted, 这样下次 UpdateLists() 的时候就必须进行再次排序.

因此, 对于具有同样 Depth 的实体, 它越晚被设为此 Depth/被加入场景, 它的实际深度就越负, 也就越晚更新.

注意到 actualDepthLookup 随着 Scene 建立就再也不能通过其他方式变动了. 除非 Scene 被替换掉 (比如通过一些重新开始, 金草莓死亡之类的手段 LevelExit), 否则 actualDepthLookup 只会越垒越高. 甚至可能高到明明 a.Depth > b.Depth, 却 a.actualDepth < b.actualDepth.

14.4 风与玩家的更新顺序

根据 第 17.31 小节, 我们知道这取决于进入房间时, LoadLevel(..) 和 Add(player) 哪个先调用. 结果是只有 TransitionRoutine 或者参数为 Transition 的 TeleportTo 会先调用 Add(player) (或者压根没 Remove(player)). 后者不太会遇到, 因此只有切板时才会玩家先更新, 其余的如在房间内死亡, 重试等, 都会使得玩家后更新.

上面说的只是进入房间瞬间的更新顺序. 但不少如 Player.IntroJumpEnd (例如 7A 3000m 被 Badeline 抛上去), Player.DreamDashEnd 都会设置 Depth = 0, 这也会使得玩家的更新晚于风的更新.

14.5 使用 TAS Helper 来研究运算顺序

Order of Operation

经典案例:

- 标准的一套时序 (MoveH, MoveV, EntityCollide, MoveBlock, CelesteTAS command, ...)
- OnGround, RefillDash, RefillStamina, 各类 Timer 的时序.
- 风与玩家
- 移动块多 cb, 以及移动块高速情况下无法 cb
- Roboboost
- Spring-moon boost
- 1A ClimbHop

15 Fling Bird

Fling Bird, 即 Farewell 里那个会把你甩飞出去的鸟, 它的猛扔动画分为以下步骤:



15.1 机制

(1) 等待, 直至 Maddy 碰到鸟的触发区域.

(2) 一阶段 (鸟接住 Maddy):

Maddy 进入 `StFlingBird`(以 Celeste Studio 显示的为判定标准).

鸟的初始水平速度 = Maddy 的水平速度 $\times 0.4$, 鸟的初始竖直速度 = 120.

加速度 13.33 (单位: px/s f)²⁶, 方向朝着现速度与目标速度 (0, 0) 的连线.²⁷

速度达到 (0, 0) 时进入下一阶段.

(3) 二阶段 (鸟左移蓄力):

鸟的速度重置为 (-140, 140).

加速度 (-13.20, 13.20), 经过共 9f 变为 (-21.21, 21.21).

进入下一阶段.

(4) 三阶段 (鸟转身甩出 Maddy):

鸟的加速度变为 80, 方向朝着现速度与目标速度 (380, -100) 的连线, 即加速度为 (76.58, -23.14).

在第 9 = 1+3+5 帧变为 (380, -100), 其中 2-4f 是冻结帧. 然后还会再持续 3 帧.

结束 `StFlingBird`.

(5) Maddy 回到 `StNormal`, 速度被设为 (380, -100). 有 `AutoJump`, 12f 的 `varJump` (速度 -100), 以及 12f 的 `forceMoveX` (朝右). 同时鸟完成剩下的动画.

在 `StFlingBird` 阶段, Maddy 以速度 (± 200 , ± 200) 朝着鸟的正下方 17px (蹲姿则 14px) 运动.

- 如果 Maddy 的初始水平速度不是太高谱²⁸, 也没有墙壁等阻碍 Maddy 运动, 则在第一阶段结束时, Maddy 能成功移动到鸟的正下方 17px (14px). (*)

²⁶在 `StFlingBird` 中, 有 `TimeRate = 0.8`. 原本加速度应为 $1000 * \text{Engine.DeltaTime} \approx 16.67$, 再乘 0.8 就得到 13.33. 鸟的速度虽然还是这么多, 但是位移变为 $\text{Speed} * \text{Engine.RawDeltaTime} * \text{TimeRate}$. 这里及后文只根据 `TimeRate = 0.8` 折算了加速度与位移, 没有折算鸟的速度.

²⁷第一帧也会减速, 加速度为 16.67.

²⁸速度在 1000 以下都是安全的. 更高一些则取决于第一帧的位置. 再高就怎么也不可能.

二, 三阶段不受初始条件的影响, 因此鸟的运动, 以及 Maddy 相对鸟的位置, 都是完全确定的.

15.2 StFlingBird 的时长, 水平位移, 竖直位移

根据机制, 在 **条件 *** 满足的情况下,

StFlingBird 全过程只由 Maddy 的初始水平速度决定.

Maddy 不同的初始水平速度会造成鸟的不同初始速度. 而第一阶段的鸟是匀减速运动, 加速度固定. 因此不同的初始水平速度会造成减速时间不同, 加速度方向不同, 也就导致竖直方向加速度不同.

以下,

- StFlingBird 的时长指的是 Celeste Studio 显示为 StFlingBird 的帧数.
- 位移指的是鸟从初始位置到 StFlingBird 结束后的第一帧 StNormal 时的位置之间的位移. 我们只关心不同初速度下 Maddy 最终的位移差, 而这等于鸟在不同初速度下的位移差.
- Vel.X 指的是碰到 FlingBird 的这一帧, 假如此处没有鸟, Maddy 所应该具有的 Speed.X. 在没有弹簧, 移动块推动等的情况下, 这等于显示 StFlingBird 的第一帧时 Maddy 真实的 Vel.X. 由于进入 StFlingBird 的第一帧会将 Speed 重置为 0, 观察 Vel.X 是更加方便的.

15.2.1 时长

若 $0 \leq \text{abs}(\text{Vel.X}) < 71.2027$, 则第一阶段共 9f. 于是 StFlingBird 共 30f (含 3f 冻结帧). 若 $\text{abs}(\text{Vel.X})$ 更大, 则第一阶段时长更长, 进而整个动画时长也 longer, 参考 [附录-表格 3](#).

实际上比较有用的是临界值:

71, 163, 225, 277, 324, 368, 410, 451, 490, 529, 567, 604,
641, 678, 714, 750, 786, 821, 857, 892, 927, 962, 997, 1032 ...

往往可以尝试在碰到鸟之前, 提前减速 (但仍在同一帧碰到鸟), 如果速度减到越过了临界值, 那么就省下了一帧动画的时间.

假设需要额外减速 δ 就能跨过临界值. 若相同 δ 且加速度相同 (比如都是空气阻力下每帧额外减速 6.5), 则碰鸟前损失的水平位移一样. 因此碰鸟时溢出的水平位移越多, 则允许更多的额外减速 δ . 显然在高速情形下更有可能大量溢出 (但也还是看运气). 因此

- Vel.X 较小的时候, 一般只能在速度比较接近临界值时, 做额外减速.
- Vel.X 较大的时候, 可以尝试大量减速来越过临界值.

15.2.2 水平位移差

Vel.X 越大则最终位置越靠右. 区间 $-71 < \text{Vel.X} < 71$ 对应了最终水平位置约 3px 的区间.

在高速情形下, 我们有如下近似:

Vel.X 每快 1 px/s, 造成的水平位移差相当于 Maddy 以 Vel.X 走了 0.01 帧²⁹.

考虑这样的实战情形, 被鸟扔飞后, Maddy 经过空中减速, 最终碰到某个能重置速度的物体 (e.g. 弹簧, Badeline 球). 这个过程中耗时来源于水平位移. 假设这个过程最低水平速度为 $v_{末}$, Maddy 进入 StFlingBird 的速度为 Vel.X, 且如果额外减速 δ 就能跨过临界值 (但减速不能使得进入鸟动画的时间更晚), 省下鸟动画的一帧. 那么,

如果

$$0.01 \times \delta \times \text{Vel.X} < v_{末},$$

就应该选择减速. 反之则不.

虽然少一帧鸟动画的位移, 但后面多走了一帧, 这在水平位移上是赚的. 运气足够好的话, 后面甚至可能不需要多走一帧 (也就是没减速前, 最后一帧的水平位移溢出得比较多). 则这样就可以通过提前减速赚到 1 帧.

- 当 Vel.X 较小的时候, 额外减速 δ 如果能成立, 则一般 δ 也较小 (否则基本都会更晚撞鸟), 这时不等式往往成立, 能省一帧.
- 当 Vel.X 较大的时候, 如果 δ 较小, 则一般还是能省一帧.
- 当 Vel.X 较大的时候, δ 较大, 且 $v_{末}$ 较小, 则需要根据不等式进行分析.

例: Vel.X = 472, $v_{末} = 140$, 松开右键在空气阻力下每帧额外减速 6.5, 查表知 $\delta = 26$. 得到

$$0.01 \times \delta \times \text{Vel.X} = 122.72 < 140 = v_{末},$$

因此减速有利.

但赚到的水平位移并不是很多, 因此实际情况中, 这样也并没有成功省下一帧...

15.2.3 竖直位移差

abs(Vel.X) 越大则最终位置越靠下. 区间 $0 \leq \text{abs}(\text{Vel.X}) < 71$ 对应了最终竖直位置约 0.18px 的区间.

在较高速情形下, 我们有如下近似:

$$\text{Vel.X 每快 } 1 \text{ px/s, 造成的竖直位移差为 } 0.0015 \times (T - 23),$$

这里 T 是 StFlingBird 的时长.

15.3 补充

一般来说 StFlingBird 无法打断.

²⁹Vel.X > 200 的时候比较准 (0.01 帧这个参数有 ± 0.0005 帧的误差). 在临界值附近参数有微小的突变, 但仍在误差范围内. 而 Vel.X = 150 时对应约 0.012 帧.

16 一个一个 State 全写下来...

16.1 Coroutine, StateMachine

todo: 这里的理解应该还有不少问题, 需要仔细再理解一下这堆 IEnumerator 是怎么运作的.

Coroutine 每帧的更新基本如下: 如果 `waitTimer > 0f`, 则 `waitTimer -= Engine.DeltaTime`, 然后结束. 如果 `waitTimer <= 0f`, 则调用 `enumerator.MoveNext()` 使得迭代器往前运行直到下一个 `yield`, 如果是 `yield return X f` 则 `waitTimer = X f`, 结束; 如果是 `yield return null`, 则这帧到此为止; 如果是 `yield break`, 则返回一个非空的 `count` 为 0 的 `IEnumerable` 集合, 效果是中止了此迭代器.

如果它 `set` 了 `StateMachine.State` (且状态确实改变了), 则会调用 `currentCoroutine.Replace/Cancel` 使得此 `Coroutine.ended = true`, 原 `Coroutine` 也就不再工作. 但如果替换的 `Coroutine` 非空, 那么新的 `Coroutine` 仍然会更新. `Replace` 例如 `StNormal -> StDash`, `DashCoroutine` 仍更新一次. `Cancel` 例如 `hyper: StDash -> StNormal`

`StateMachine` 这个 component 的 `Update()` 比较简单, 基本就是调用当前状态和 `Coroutine` 的 `Update()`. 其中状态的 `Update()` 会给 `State` 赋值, `State = updates[state]()`, 因此这些 `Update()` 都是 `Func<int>` 类型的.

状态转换靠的是 `StateMachine.State` 的 `setter`: 在 `locked` 或者 `State` 并未改变的情况下, 什么都不做. 其余情况下, 调用前一个状态的 `end` (如果有), 然后调用当前状态的 `begin` (如果有), 然后如果此状态有 `coroutine`, 就 `currentCoroutine.Replace(coroutines[state]())` 将其取代为当前状态的 `coroutine`, 否则调用 `currentCoroutine.Cancel()`. 注意: 在 `currentCoroutine.Replace(coroutines[state]())` 中实际上通过 `()` 运算符调用了 `coroutines[state]!` 呃, 这个理解应该是错误的, 许多情况下第一帧 `Coroutine` 被调用, 是因为 `StateMachine.Update` 里的更新顺序问题.

举例: `DashCoroutine` 大体上是 `yield return null`, 设置初速度然后 `yield return 0.15f`, 最后 `StateMachine.State = 0`. 而 `DashUpdate()` 则是超冲拐弯, 再次超冲, `PickUp`, `JumpThru` 冲刺修正, `SuperWallJump`, `ClimbJump`, `WallJump`, 以及这些操作对应的状态转换.

因此第一帧, `X End` 以及 `DashBegin()`, 通过 `currentCoroutine.Replace` 而调用了 `DashCoroutine()`, `yield return null`. 第 2-4 帧, 因为第一帧的 `Celeste.Freeze(0.05f)` 而处在冻结. 第 5 帧, `DashCoroutine` 设置初速度, `yield return 0.15f`, `waitTimer = 0.15f`. 第 6-14 帧, `waitTimer -= 0.0166667f`. 第 15 帧, 因为 `waitTimer <= 0` 而往下运行, `StateMachine.State = 0`.

因此冲刺时长 (不含冻结, 从 `DashBegin()` 到 `DashEnd()` 含头尾计) 共 $1 + 1 + \text{Ceil}(0.15 / \text{DeltaTime}) + 1$ 帧, 常规情况下是 12 帧. 同时我们知道跳跃是 `varJumpTimer = 0.2f`, 常规情况下也是 12f. 因此在引擎减速情况下, 可以跳跃后冲刺, 冲刺结束以后仍有 `varJumpTimer > 0f`.

说起来 `private IEnumerator FlingBirdCoroutine()yield break;` 完全意义不明, 或许 `FlingBird` 的 `DoFlingRoutine` 原本是写在这里的?

16.2 各 State 一览

StNormal (为了避免遗漏什么, 这次要把 NormalUpdate() 完整读一遍)(值得注意的细节, 空气阻力的顺序. 这导致若冲刺, 则这一帧不会受到空气阻力的影响. 若是跳跃, 则会先计算空气阻力, 再计算跳跃, 导致在地面速度 0 按一帧 R,J 的速度是 16.6+40.)

StClimb (同理)

StDash (同理)

StBoost (泡泡重置位置, bubble super/hyper 等的原理, 蹲姿进会高 2px)(泡泡重生时间)(进泡泡如果不快启则会以 80 px/s 的速度 approach 到指定位置, 这个位置 = $\text{Booster.Center} - \text{player.Collider.Center} + \text{Input.Aim.Value} * 3f$. 但是 BoostEnd() 时, 会移动到 $(\text{Booster.Center} - \text{player.Collider.Center}).\text{Floor}()$, 注意 StateEnd 是没法像 Coroutine 那样打断的. 呃, 应用场景: 假如有个倒 T 字形的空间, 墙壁全是刺, 泡泡在交叉口, Maddy 在左侧, 开关在泡泡中心右侧 3px 处, 道路在上方, 那么由于 Maddy 没法越过泡泡, 且基本上只有一次触发泡泡的机会, 就必须用这一段动画移动到开关处触发, 然后再上冲.)(由于用的是 MoveToX/Y, 因此泡泡的重置位置会有微小的浮点误差. 这个 FNA/XNA desync 最后怎么修的来着...)(你有可能在进入泡泡后无法 uncrouch, 这样就可以在红泡泡中一直保持 crouch, 哪怕之后的位置允许 uncrouch)

StDummy

StPickUp (在 StPickUp 结束时, 如果手持实体且实体带有 SlowFall 属性, 那么若 gliderBoostTimer > 0f 且 gliderBoostDir.Y < 0f 则将纵向速度重置为 $\text{Math.Min}(\text{Speed.Y}, 240f * \text{gliderBoostDir.Y})$, 否则若... 这个 gliderBoostTimer 一般都比较长, 使得接水母的容错相当大.)(在且只能在 StNormal 投掷手持实体时, 会触发实体的 OnRelease(force) 来赋予实体速度. 而对于 Player, 取决于是否按下以及朝向, $\text{force} \in \{0, \text{Vector2.UnitX}, -\text{Vector2.UnitX}\}$ 对于水母, Theo 水晶, 以及绝大部分 mod 实体, OnRelease 基本都是 $\text{force.Y} = (\text{force.X} == 0f) ? 0f : -0.4f$, 然后 $\text{Speed} = \text{force} * N$, N 取决于实体, 对于水母是 100f, 对于 Theo 水晶是 200f. 此外除了水母的实体, 在 $\text{Speed} \neq 0$ 的情况下还会获得 noGravityTimer.)(StPickUp 结束时, 若没有 gliderBoost 且 $\text{Speed.Y} < 0$, 则重置到速度至少为 -105, 而双水母足够 cannotHoldTimer 循环, 因此可以双水母升天. 一般好像是用 21-22, G; 1, D 这样循环)(于是 StPickUp 会有三种速度重置: gliderBoost (至少向上 $240 * \text{Dir.Y}$), 速度重置至至少向上 105, varJump 的速度重置)(在超冲的情况下, 5RX 1RUG 会导致 $\text{gliderBoostDir.Y} = 0f$ 但 $\text{Speed.Y} < 0f$ (超冲的转弯在抓取之前), 于是速度变为 -105f.)

StFlingBird

StLaunch (弹球/Puffer 的不应期)(完全没法自己控制左右上下速度, 当二维速度小于 220 时才会回到 StNormal. 此外可通过冲刺提前结束, 可通过 StPickUp 打断.)

StRedDash 可跨版, 红泡泡重生时间, 红泡泡不能 wavedash 和 hyper, 呃从红泡泡斜下冲再跳跃是不是有 ultra 的效果但这个速度比较低

StSwim (可以 climbhop 上岸)(+40 * 10)(在水面静止不动时, Vel.Y -10, -20, +30 循环. 以此为例解释水面附近速度的沉浮机制)

StHitSquash (可以按跳无条件地进入 StNormal, 否则会有一段 hitSquashNoMoveTimer)

StDreamDash (进入方式: onCollideH/V, 在果冻方向 dir (正上下左右) 上过 DreamDashCheck)(DreamDashCheck: DashAttacking 且 $(dir.X = \text{sign}(\text{Speed}.X) \parallel dir.Y = \text{sign}(\text{Speed}.Y))$, 因此你可以在侧面通过 $dir.Y = 0$ 来进入, 哪怕你的 $dir.X$ 是 0 甚至反向! (怎么在 dashAttack 期间将 Speed.X 扭过来并撞果冻就是你的事了.. 比如 corner glide 将 $dir.X$ 重置为 0))(进入后 $\text{Speed} = 240 * \text{DashDir}$)(如果手持物品的话, 虽然你没法冲刺, 但你可以先冲刺再抓物品, 然后利用 DashAttack 进去)

StStarFly (不输入的话, 羽毛会保持之前的方向, 这大概是 Input.GetAimVector 相关?)(feather boosts happen because the feather doesn't cull inputs to circle on the first frame, so any square gate input outside the circle will get additional speed)

StCassetteFly (时长是完全固定的, 不依赖于初始位置. 中间的飞行段的曲线 cassetteFlyCurve 由三个参数控制, 分别是起点, 终点和控制, 其中终点和控制都是被触发它的对象唯一决定的 (如 Cassette/BubbleReturnBerry), 起点则就是玩家位置. 由跟随时间变化的参数 cassetteFlyLerp 来控制飞行到曲线上的何处, 这个变化过程是被写死的 (仍受到 TimeRate 影响). 在结束时, 会重置位置到终点, 但是位置 = Position = 整数坐标, 因此不重置亚像素! 简言之, 不能调节时长, 不能调整整数坐标, 可以调整亚像素.)(值得注意的是, 飞行段 $\text{Position} = \text{cassetteFlyCurve.GetPoint}(\text{Ease.SineInOut}(\text{cassetteFlyLerp}))$ 是浮点数, 因此如果 cutscene, 会使得整数坐标变得不再是整数.)(此外 CassetteFly 还会改变 Sprite.y, 因此如果打断了 StCassetteFly 会影响渲染. 参考春五全收集 Kokodoko <https://www.bilibili.com/video/BV1X24y1f7Gu>)

StIntroWalk/Jump/Respawn/WakeUp

StAttract/StTempleFall/StReflectionFall/StIntroMoonJump/StIntroThinkForABit (据说深度不一样的话, StAttract 会表现的稍不一样)

StFrozen (State 17)

17 拾遗

17.1 冻结帧

冻结帧的机制应该是, 一切属性继承上一帧 (Pos, Speed, Vel, 冻结帧以外的状态, Timer (包括游戏计时器和其他实体的计时器)(除去 FreezeTimer), etc), 且停止根据 Vel 更新位置. 就是字面意思的暂停.

注意, 这使得 Celeste Studio 中 Selected 显示的帧数, 和实际上游戏计时器走的帧数, 不一样.

冻结帧期间, 背景音乐不停. 由于 Cassette Block 需要和背景音乐对齐, 因此 Cassette Block cycle 在冻结帧也是走的. 因此等待 Cassette Block cycle 时每次冲刺都能节省 3 帧.

呃, 好像长的冻结帧会覆盖短的来着, 因此不会叠加 (如果这些冻结在同一帧触发).

17.2 Cassette

这里主要说原版 Cassette. 我们以 TAS Helper 处理过后的信息来讲解.

- 每帧 timer += 1f, 如果 tempo 不等于 1, 则是 timer += tempo.
- 每当 timer 满 10f, (beat)Index ++.
- 现在有 beatsPerTick = 4, ticksPerSwap = 2. (原版情况下似乎都是这个配置)
- 于是我们可以导出一个概念, beatsPerSwap := beatsPerTick * ticksPerSwap = 8. 并令 beat := Index % beatsPerSwap. 原版情况下基本就是 beat = Index % 8.
- 每当 beat % beatsPerTick = 0, 响起一声较轻的嘀嗒声.
- 每当 beat 满 beatsPerSwap = 8, 这帧的较轻嘀嗒声改为较重的嘀嗒声, 同时颜色就切换到下一种. 源码里是 currentIndex 变化, 我们不妨把 currentIndex 重命名为 colorIndex.

切版时, Index, timer 等都不变. 颜色数, tempo 可能会发生改变. 而 colorIndex 会重置. 重置规律如下: 一般来说, 取决于 Index % beatsPerSwap 是否大于 beatsPerSwap / 2. 如果是的话, 那么颜色变为第 (-2) 种, 否则第 (-1) 种. 因此当 beatsPerSwap = 8 时, 同余类为 0,1,2,3,4 时为第 (-1) 种, 同余类为 5,6,7 时为第 (-2) 种. 可见它们之间并不是对称的!

特殊情形是, 如果 MapMeta 里有 CassetteModifier 且 mapMetaCassetteModifier.OldBehavior = true, 那么 colorIndex 取决于 Index / beatsPerTick 再模去颜色数, 也就是说主要取决于在第几个嘀嗒. 如果颜色数是 ticksPerSwap 的倍数, 那么就是取决于这是一个 swap 里的第几个嘀嗒 (否则还受这是第几个 swap 的影响!)(比如 9A 会有颜色数为 3 的情形).

SJ cassette 是一坨.

17.3 切版

切板移动机制 (这里指的是 EnforceBounds 部分, TransitionRoutine 更会晦涩就留到后面了)

bubs drop

纵向冲刺切板蹭墙, StDash 没了但 DashAttack 还有, 并且横向速度也没了 (竖直速度好像还有). 因此需要速度的话得切板后再 wb.

DashBlock (可撞碎的砖块) 在销毁时会有 3 帧冻结帧. 特别的, 切版也会销毁它. 因此有的时候切版时你需要等 43f. 呃, 如果有多个 DashBlock 会需要等待 $40 + 3 * n$ 帧吗, 应该不会.

呃, 为啥 DashBlock 被撞碎之后再切版回来, 它不会再出现? 同理草莓? 这应该与 DashBlock.RemoveAndFlagAsGone() 有关.

切版会重置亚像素, 因此可以使一些操作变为可能.

由于切板与碰撞箱有关, 因此部分情形下上切板时解除蹲姿会快一帧.

不同方向切板的速度重置机制. 位置重置机制. AutoJump, DashCooldownTimer 等机制.

5A a-01 -> a-13 的切板是 59 帧, 可能是因为亮度变化?

17.4 各类方块被触发

从 NormalUpdate 得到的按抓触发方块, 条件主要是 $Speed.Y \geq 0$, Holding = null, 按抓, (如果速度与 Facing 同向则偏移 2px 判定碰撞, 碰撞则进入 StClimb. 进入 StClimb 会修正位置至贴墙, 水平速度归零)/(否则, 若 moveX 不与 Facing 反向, 不按下, 向 Facing 偏移 1px 判定碰撞, 碰撞则获得 climbTriggerDir = Facing), 此外还有蹲姿, ClimbBlocker, wallSlideTimer > 0f (也就是抓墙直至耗尽体力再耗尽完 wallSlideTimer 后, 无法触发 climbTrigger) 等. 这些判定完之后是抓跳等.

这两种会影响 bool IsRiding(Solid solid). 如果在 StClimb/StHitSquash, 那么判定的是 CollideCheck(solid, Position + Vector2.UnitX * (float)Facing); 如果在 StNormal 且 climbTriggerDir != 0, 那么判定的是 CollideCheck(solid, Position + Vector2.UnitX * climbTriggerDir); 如果在 StIntroMoonJump 则 return false; 如果在 StDreamDash 那么判定 CollideCheck(solid); 否则判定 CollideCheck(solid, Position + Vector2.UnitY).

ZipMover 等方块, 在自身的更新中通过检测 IsRiding 来被触发. 因此触发方式是: 两类按抓 + DreamDash + 脚踩.

在速度与 Facing 反向的情形, 注意 climbTrigger 并不是直接触发方块, 而是改变了 IsRiding 的判定, 因此若这一帧速度太快, 移动至方块 1px 以外, 在方块更新的时候, 仍然无法触发 IsRiding, 就没法触发方块.

简言之, 需要水平速度 + 水平位置合理. 这里允许的组合是: 水平速度与 Facing 同向且水平位置 2px 以内; 或水平位置更新前后都在 1px 以内.

其余 State 的暂且还没看.

注意到在同一帧内, Maddy 只能有一种判定方式. 因此在合适条件下, 可以站在 ZipMover 上但不触发它. 比如 StClimb, 比如 climbTriggerDir.

17.5 BadelineBoost

我们这里讨论那种不是切版的 BadelineBoost. 切版的也差不多.

BadelineBoost 的深度是负的. 接触 Maddy 时给自身加上 BoostRoutine, 在自身更新到 BoostRoutine 的时候, 会首先设置 Maddy 为 StDummy, 且速度强制归零.

整个 Coroutine 包含进入 StDummy 这一帧在内的 12f 微移 (12 个 yield return null) + 1f (播放音效) + 6f 发呆 (yield return 0.1f) + 下移 5px 的 1f + 6f 发呆 (yield return 0.1f) + 1f 这帧进入 StLaunch.

取决于接触 BadelineBoost 时的位置, 决定水平朝向 num. Coroutine 开始时临时生成一个 BadelineDummy (badeline 而非球体), 然后这 12f badeline 做出一个接住 Maddy 微微后退的动作. 而以 Maddy 开始的位置 playerFrom = player.Position 为起点, playerTo = BadelineBoost.Position + new Vector2(num * 4, -3f) 为终点, 进行线性插值. 但是第 1 帧是仍在原位置, 这导致第 12 帧结束时未完全到达终点, 而是差了 1/12. 此外需要注意, player.Position 是整数坐标! 因此亚像素会被完全吞掉. 你必须真正往下移动满 1px, 才能最终位置移动 1/12 px, 单纯移动亚像素是没用的.

12f 之后, Coroutine 里不再有强制设置 Maddy 位置的. Maddy 本身尽管速度为零且 StDummy, 但并不是完全没法动, 还可以利用 varJumpTimer 微调位置, 但 12f 的间隔导致这只能是蹭墙跳. (这种技巧叫 jump timer abuse)

进入 StLaunch 时, 会将 Player.launchApproachX 设置为 BadelineBoost.CenterX. 注意, 是球体而不是 BadelineDummy, 因此不受前面 1/12 的影响, 是一个整的坐标. 而本身也并不改变 Maddy 的位置.

Player.LaunchUpdate 中, 会逐渐将 Maddy 的水平位置移动到 launchApproachX.

因此最终结论, 接触 BadelineBoost 时会导致亚像素归零. 在不考虑蹭墙跳 varJumpTimer 的情况下, 接触 BadelineBoost 时整数坐标每移动 1px, 进入 StLaunch 时的水平/竖直位置就移动 1/12 px, 而 StLaunch 将结束时则水平位置和 BadelineBoost 初始位置对齐 (这会导致整个过程中有 4 ~ 5 px 的位移, 相当显著!), 竖直调整依旧. 如果是改变了接触 BadelineBoost 时的水平相对位置的正负, 那么水平位置受的影响就更大了.

17.6 Puffer

Puffer 的 PlayerCollider 在 Player 更新时触发, 而自身的半圆碰撞箱则是在自身更新时使用. 它并不是严格的一个半圆, 而是一个名为 detectRadius 的 Circle collider (半径 32) + 检测时对 Player 的纵坐标的检测, 所等效出来的一个弓形. 由于写的比较微妙, 导致 Player 蹲姿/站立情况下, 这个弓形的大小是不一样的. 爆炸时, 会调用 Puffer.Explode().

Puffer.Explode() 使用 pushRadius (半径 40 的圆), 对范围内的 Player 检测是否被 Solid 阻挡后, 调用 Player.ExplodeLaunch(..

在 Puffer.Update() 中, Explode 只能由弓形碰撞箱内不被 Solid 阻挡的 Player 触发, 且在 Puffer 为 States.Gone 期间不触发. 然而, Explode 还可以由 OnSquish 触发, 这不受到 States.Gone 的影响. 因此 Puffer 在被移动块挤压的情况下可以短时间内触发多次 Explode().

如果是 Player 碰撞弓形碰撞箱, 那么必然导致 ExplodeLaunch() 的方向不可能向上. 然而, 如果由 OnSquish 触

发, 则有可能. 正如 Puffer 向正下炸的情形, 向正上炸在这种情形下也是可能的. 这需要 Player 的中心在 Puffer 的位置的正上方.

如果是单个移动块挤压触发 Puffer, 由于 Solid.MoveH/V 在触发 Actor.OnSquish 前, 会关掉自身的碰撞箱, 因此它们无法起到阻挡爆炸冲击波的效果.

噢, 说漏了, Explode() 居然还有一种触发方式, 那就是 OnPlayer 中 `player.Bottom > lastSpeedPosition.Y + 3f`, 这个高度限制和 ProximityExplodeCheck 的弓形范围并不一样! <https://discord.com/channels/403698615446536203/5192813>

17.7 JumpThru

<https://discord.com/channels/403698615446536203/615402712011636777/886535997507125278>

JumpThru 修正 (-40) 不是什么时候都有的. 下落时候就没有. 此外这也会受到 LedgeBlocker (CrystalStaticSpinner/DustStaticSpinner/Spikes/TriggerSpikes) 影响.

有时候需要避免此修正 (e.g. 从云下面上冲但前面不好调整高度), 可能需要 dd 来尽量减少此修正.

此外在冲刺时, 若 Maddy 脚底在板子顶部底下 6px 以内, 还会直接修正上去.

即使是水平移动的 JumpThru, 也会有上下浮动, 坑啊

17.8 Bubble Corner Glide

呃, 我不太知道叫什么, 但 Retention tech gym 里应该有写. 就是首先撞墙, 然后碰到泡泡, 速度虽然归零但下一帧又因为 retention 机制获得速度. 此时若直接冲刺, 则可以在泡泡形态下以高速开始冲刺 (如果原先也是高速). 如果不冲刺, 则是具有速度的同时, 又有 StBoost 下被泡泡拉回的 Velocity.

现在我们已经可以在 StBoost 下获得了速度, 此时触发一个交互:

- Bounce (雪球)/ SuperBounce (向上弹簧), 那么会调用 CurrentBooster.PlayerReleased (尽管这使得 cannotUseTimer 归零, 但 respawnTimer = 1f, 因此短时间内无法再与之交互), 置空 CurrentBooster, 然后再设置 StateMachine.State = 0, 但 BoostEnd 里用到的 Vector2 boostTarget 并没有重置, 导致会瞬移到泡泡中心, 并获得 Bounce/SuperBounce 对应的向上速度.

现象: 泡泡消失, 瞬移到泡泡中心, 速度同正常触发雪球/弹簧等.

- SideBounce (左右弹簧), 那么同向弹簧不触发, 反向弹簧触发. 由于 SideBounce 是先进行关于弹簧位置的像素水平垂直重置, 再 StateMachine.State = 0, 且不调用 CurrentBooster.PlayerReleased, 不置空 CurrentBooster, 更不会重置 boostTarget. 因此会瞬移回泡泡中心, 泡泡仍在但由于 cannotUseTimer 还在所以无法交互 (由于这个时间比泡泡自动进入 StDash 的时间更长), 获得照常的侧边弹簧速度. 由于 CurrentBooster 未置空, 因此此时冲刺则是泡泡形态的冲刺.

现象: 泡泡仍在, 瞬移到泡泡中心, 速度同正常触发弹簧, 下一个冲刺是泡泡形态.

- 碰到第二个泡泡. 这种情况约等于两个泡泡离的很近的情况下, 自然的碰到了第二个泡泡. 则第一个泡泡不会被释放 (贴图还在, cannotUseTimer 还在). 此时若从第二个泡泡冲出去, 则可以直接穿过第一个泡泡.

现象: 第一个泡泡仍在, 约等于两个泡泡离的很近的情况.

- 碰到羽毛. 没有任何对泡泡的处理就进入 StStarFly. 因此先触发 BoostEnd 的瞬移回泡泡, 然后就是常规的羽毛启动流程.

现象: 泡泡仍在, 瞬移到泡泡中心, 羽毛正常启动.

- PointBounce (带气泡盾的羽毛). 由于 PointBounce 调用 CurrentBooster.PlayerReleased 但不置空 CurrentBooster, 也不改变 StateMachine.State. 因此在 StBoost 下获得 PointBounce 的速度. 此时泡泡贴图消失, 进入重生期. 但由于玩家还在 StBoost 中, 因此一段时间后 (或者提前按冲刺触发) 会瞬移回泡泡中心, 开始冲刺. 在冲刺的第 5 帧, Player.CallDashEvents() 被调用, 由于 CurrentBooster 没有被置空, 这使得它瞬间复活 (至少贴图上看是的), 使得玩家的冲刺为泡泡形态. 由于这段期间我们都是在 StBoost 中, 没有 BoostBegin 的速度重置, 因此开始冲刺时, 如果原先速度较高, 则可以以较高的速度进行泡泡冲刺.

现象: 泡泡消失, 速度同正常触发气泡盾, 一段时间后瞬移回泡泡中心, 并开始泡泡冲刺.

以上的瞬移回泡泡, 实际上是 MoveH + MoveV, 因此遇到障碍物时情况更加复杂.

17.9 Cutscene

不同的剧情对应的 CutScene (CS) 都是单独写的, 因此它们的效果不一样. 可能的操作包括但不限于把状态变为 StDummy (具体形态各异)/StNormal, 重置 (某一方向上的) 速度/位置, 各种各样的动画效果... 以下是部分可能有的效果及其应用

部分 Cutscene 会将 StDash 变为 StNormal, 因此可以有水母 ultra 一样的效果.

部分 Cutscene 会阻止 camera 移动, 阻止 Level bounds. (因为这需要在 player.Update() 里触发, 但 level.Frozen = true 时不触发不进行 player.Update().) 如果能移动的话可以借机到一些地方去, 比如 Madeline in China 的 any%.

部分 Cutscene 会移动 camera. 可以利用这一点把镜头拉到特定位置, 然后再跳过动画, 从而使 Maddy 离镜头非常远并借机到一些地方去. 不过这应用场景可能不是很多.

部分 Cutscene 会重置 (一些方向上的) 速度, 但是利用 retention 你可以再把速度拿回来.

草莓种子 CutScene (CSGEN_StrawberrySeeds) 不重置 State.

17.10 望远镜, TalkComponent

更新: 注意 TalkComponent 是在 Lookout 等实体上的. Player 更新里是检测 TalkComponent.PlayerOver == null || !Input.Talk.Pressed 来判定是否是在与 TalkComponent 交互的 (也会用类似的东西来检测 CanDash). PlayerOver 是在 TalkComponent 自身更新的时候设置的, 是 TalkComponent 类的静态字段.

depth 不同的 TalkComponent, 它们的触发时间节点会很恶心. 如果场景里邻近的地方不止一个 TalkComponent, 那可能情况就更糟糕了. 这种情况下何时按 X 算冲刺, 何时算对话, 很恶心, 但是出于职业道德, 我们要把它探讨完.... 之后再写吧

```
Lookout.LookRoutine(Player player);
```

原地 1,X; 2,R,J: 在交互的那一帧, 望远镜将 LookRoutine 这个 Coroutine 加给 player, 而 Coroutine 是 Component, 自然是下一帧才更新的. 所以在下一帧才变为 StDummy. 大概由于 StateMachine 这个 component 比 LookRoutine 先被加进去的原因, 因此先 StNormal 更新, 获得 (56.6, -105) 的速度, 然后变为 StDummy. 然后再是位置更新 (此外注意这一帧不能按对话键, 否则不满足跳跃的触发条件). 呃, 第三帧及之后呢, 还得看 StDummy 和 LookRoutine...

TalkComponent.Update(), 这应该是触发相关.

```
if (flag && cooldown <= 0f && entity != null && (int)entity.StateMachine == 0 && Input.Talk.Pressed)
{
    cooldown = 0.1f;
    if (OnTalk != null)
    {
        OnTalk(entity);
    }
}
```

我还没完全解明望远镜相关, DTS 前面 NoControl 那段的原理到底是什么:

2,L,J. 在第一个 L,J 的时候, 是从 StNormal 进入 StDummy, 当然是可以的. 第二个, StDummy 并非完全不允许操作, varJumpTime 是可以用 Input.Jump.Check 的! 呃, 为什么要这么写.

arguably 3 types of bino storage

bino interaction - this only has an effect when leaving a room - easiest to get

bino control storage - allows you to control the camera, has the same effect as bino interaction storage when leaving room - also makes you invisible if you leave whilst it's active - requires being able to cancel the dummy state of a bino

bino dummy storage - abuses the fact that being in a bino disables room bounds - requires specific setup to make work

特别的, 你可以利用切板回到 StNormal 这一点, 来做出一个类似 cutscene ultra 的东西.

例: 9-9 skip 的 Bino control storage 用完了之后, 还榨取了退出望远镜会重置为 StNormal 这一点, 将冲刺取消掉了, 从而实现了水平速度 *1.2 & 竖直速度 =-105.

<https://discord.com/channels/403698615446536203/519281383164739594/1087431377034682460>

<https://discord.com/channels/403698615446536203/519281383164739594/1087509447569592462>

TalkComponent 的判定条件是 StNormal 加上 OnGround(), 以及其他如碰撞箱, 冷却时间, 按下对话键之类的要素. 注意, 这里是 OnGround() 而非 onGround, 前者只判定位置, 后者则是 Player 每帧更新的, 大部分情况下

需要竖直速度不朝上的. 因此, 望远镜可以在有向上速度的情况下触发.

跳跃打断望远镜: 红框部分通过跳跃离地打断 DummyWalk, 蓝框部分通过离地来中止与望远镜的交互, 并进入 StNormal.

```
private IEnumerator LookRoutine(Player player)
{
    Level level = SceneAs<Level>();
    SandwichLava sandwichLava = Scene.Entities.FindFirst<SandwichLava>();
    if (sandwichLava != null)
    {
        sandwichLava.Waiting = true;
    }
    if (player.Holding != null)
    {
        player.Drop();
    }
    player.StateMachine.State = 11;
    yield return player.DummyWalkToExact((int)X, walkBackwards: false, 1f, cancelOnFall: true);
    if (Math.Abs(X - player.X) > 4f || player.Dead || !player.OnGround())
    {
        if (!player.Dead)
        {
            player.StateMachine.State = 0;
        }
        yield break;
    }
    Audio.Play("event:/game/general/lookout_use", Position);
    if (player.Facing == Facings.Right)
```

图 27: 跳跃打断望远镜

由于 Player.DummyWalkToExact 的第一帧如果恰好水平位置相等, 则直接 yield break, 因此 1RN 2RJ 假如在跳跃的第二帧水平位置恰好与望远镜相同, 则可以让速度归零.

17.11 按钮

DashSwitch, 就是 Farewell 那个可以被水母撞开的按钮.

是 Solid, 可以抓跳.

对于朝上的, 它可以被 Maddy 踩下去一点点, Maddy 离开后会恢复. 如果 Maddy 手持物品的话则会直接踩塌触发按钮.

三类触发: 更新节点所在的对象是 Maddy, 水母, 水晶, Seeker, 自身.

第一类: 水母, 水晶, Seeker, 各自写在这些对象的更新当中.

第二类: 在 Player 的更新中. DashSwitch 本身有 OnDashCollide = OnDashed(Player player, Vector2 direction), 而 Player 类在 OnCollideH/V 会调用 CollisionData data, data.Hit.OnDashCollide(this, data.Direction), 因此触发按钮的开门效果.

第三类: 在 DashSwitch 的更新中. 如果 Maddy 手持物品的话则会直接踩塌向上按钮.

OnDashed 会先让 DashSwitch 移位, 然后再返回 DashCollisionResults.NormalCollision. 但是如果是近乎竖直的冲刺的话, OnCollideV 里对 NormalCollision 的结果, 是要试图移动 (X, ±1) 以避免碰撞的 (± 取决于速度方

向, X 从 -4 到 +4 但不包括 0)。由于按钮已回缩, 导致横向移动 1px, 纵向移动 1px, 就已避开。因此很奇怪地产生了 1px 的水平位移。这也是 5B [c-02] 那里贴墙上冲按钮掉下来会被刺扎死的原因。

17.12 WindMove

我们知道, 关于 WindMove 的水平部分在玩家上的作用机制, 大体有以下要素:

- 冲刺时不受影响。
- 在地面上蹲姿 (onGround && Ducking) 则不受风的影响。
- 风的反方向上 3px 内有墙则不受影响。
- 不会使你切板。(严格来说, 如果 WindController 的深度比玩家低, 那么可以玩家更新导致切板, 然后 WindController 再把玩家移回板面内 (仍然切板, 只是位置变为恰在板边))
- 刚复活, noWindTimer, 以及一些其他情形, 也不受影响。

但这里面有个很 tricky 的地方, 这是部分代码:

```
if (move.X != 0f && ...){
    ...
    if (Ducking && onGround){
        move.X *= 0f;
    }
    if (move.X < 0f){
        move.X = Math.Max(move.X, (float)level.Bounds.Left -
            (base.ExactPosition.X + base.Collider.Left));
    }
    else{
        move.X = Math.Min(move.X, (float)level.Bounds.Right -
            (base.ExactPosition.X + base.Collider.Right));
    }
    MoveH(move.X);
}
```

乍一看, 就是三类互不重合的情况, 地面蹲姿则 $move.X = 0f$, 左风则避免左切板, 右风则避免右切板。但仔细一品, 发现地面蹲姿 $move.X = 0f$, 仍会触发避免右切板的这个分支, 哪怕原本的风是左风!

17.13 云

云 Cloud, 由 bool fragile 来决定它是白云还是粉云。

云在 Cloud.Speed $\geq -100f$ 且 playerRider2.Speed.Y $\geq 0f$ 时, 会设置 playerRider2.Speed.Y = -200f (如果是粉云, 粉云会碎裂)。于是甚至可以云上平 u 速度却竟然是 (390f, -200f)。

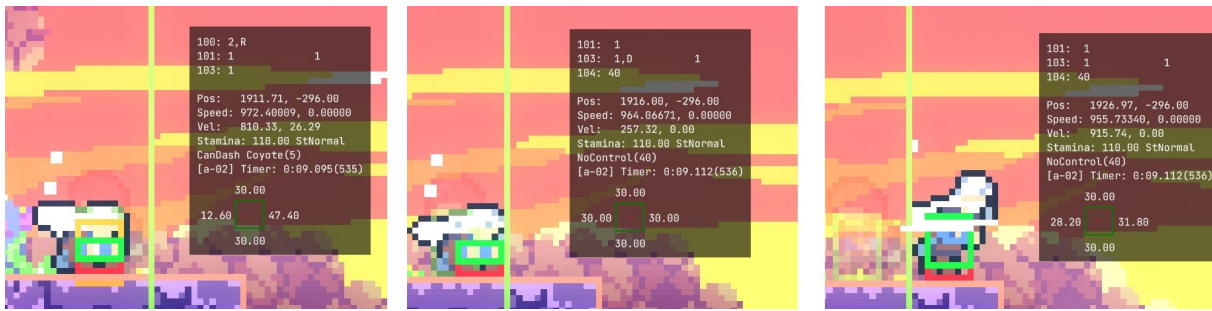


图 28: 在有左风的情况下, 利用蹲姿将其算成“右风”, 触发了“避免右切板”的机制, 使得人物被移回了板面内 (由于时序, 仍然切板).

因此在平 u cb 的时候, 这是有助于 cb 的. 几个可能的纵向速度: $-200f$, $-117.5f$ (云 liftboost 下的 hyper), $-235f$ (云 liftboost 下的抓跳). $-200f$ 最重要的是可以应用于没撞墙前那段, 因此这可以很大程度上降低地形要求.

(底下没有刺的) 白云上可以充能平 u,

```

13 | L,D,X
   |
  1 | R,J
   |
  1 | J
   |
 14 | R,D,X

```

从落到云的第二帧 (即从无冲刺恢复到可以冲刺的那一帧) 开始写. 需要第一个冲刺的第 10 帧在云上 (调整垂直高度). 亚像素调整的容错约有 $0.5px$, 相对好调.

云能接住掉落块, 似乎还能弹起刺? (参见 Madeline in China, 不知道是原版云还是扩展云?)

由于云在上升, 使得你可以获得比地面上的兔子平 u 更快的速度. 比如 $10LZ + 1RJ + 3R + 1RJ + 1RK + 14RDX$, 速度 $409.4B$ 中用到了这一点.

17.14 Kevin 块

某些情形下, 无法被撞击触发. e.g. 6A 的大 Kevin. 由 `CrushBlock.CanActivate` 控制, 这里面具体分为: 首先 `giant` 类只能往右, 然后由 `canMoveHorizontally`, `canMoveVertically` 控制.

`CrushBlock` 的攻击行为由 `attackCoroutine = AttackSequence()` 给出. 速度上限 240 , 某些会减速刹车, 可以触发 `FallingBlock`. 在攻击结束后会堆栈地回程, 每次起步速度 0 , 最高速度 60 . `Solid` 类的 `MoveH/V` 并不做 `Solid` 与 `Solid` 的碰撞检测, 只在撞击阶段会有 `CrushBlock.MoveH/VCheck`, 因此回程阶段 `MoveTowardsX/Y` 是可以穿墙的. 一般来说没机会见到, 比较容易见到的情形是有两只 Kevin 块, 这在一些 mod 图里容易遇到.

呃, 好像也不完全是堆栈地回程. 例如 6A/02 的那只 Kevin, 先引到右侧, 再引到左侧, 就直接停了.

Kevin 的回程是, 按 `moveState.Direction` 决定朝哪个方向回程到原先的分位置, 而垂直方向上实际上不进行移动. 因此 Kevin 并不会记录下它的那些位置修正. 只需撞击 -> 位置修正 -> 回程 -> 堆栈清空 -> 循环, 就能改变 Kevin 的”起点”.

17.15 果冻双跳

在 `StDreamDash` 下, 我们无法像常规情况一样获得 `onGround`, 进而刷新 `jumpGraceTimer` (`jumpGraceTimer` 的流逝依然有), 而是在 `StDreamDash` 结束时, 若冲刺方向不竖直才给予. 除此之外, 若冲刺方向不竖直, 则游戏允许你在 `DreamDashUpdate()` 中判定为离开 `StDreamDash` 的这帧里, 进行一次跳跃. 如果这里不允许跳跃的话, 那么果冻跳的输入窗口将变为 9 帧而不是 13 帧 (注意冻结帧期间 `buffer` 也会流逝).

因此果冻双跳就是: 最后一帧的 `DreamDashUpdate()` 中跳跃 -> `DreamDashEnd()` 又获得狼跳时间 -> `StNormal` 里再次跳跃.

上面提到的给予狼跳时间或跳跃, 都是需要冲刺方向不竖直才行的, 很明显 `Celeste` 并不想让我们竖直上下冲之后能跳起来 (这样可能显得过于怪异了).

17.16 Wall Slide

`wallSlideTimer` 在 `wallSlideDir` 非零期间, 每帧会减少 `Engine.DeltaTime`, 一旦冲刺, 跳跃, 有体力情况下按抓等, 就会重置 `wallSlideTimer = 1.2f`.

`wallSlide` 就是努力贴墙但不按抓, 也会减慢下落. 由于前述, 意味着你可以分段地使用 `wallSlideTimer`.

目前没找到啥应用, 可能用来调亚像素吧. 那么应该是可以用来调 `cp` 的. 感觉基本就是常规抓的下位替代, 无抓情况下的替代品罢了.

<https://discord.com/channels/403698615446536203/519281383164739594/1062862292611498004>, `wall slide` 和按跳重力减半的效应结合, 尽可能慢地下落.

17.17 钥匙机制

`LoadLevel` 期间, `Key` 的两个不同构造函数被调用, 其一就是正常的加载地图上的 `Key` (以下称第一类构造), 其二则是构造并将其加入 `Player` 的 `Follower` 中 (以下称第二类构造).

17.17.1 LoadLevel 相关

`OnPlayer` 时, 会调用

```
session.DoNotLoad.Add(ID);
session.Keys.Add(ID);
```

这里 `ID` 是 `Key` 的 `EntityID`.

这导致 `LoadLevel` 时,

- 地图上的实例: `DoNotLoad` 会使得 `LoadLevel` 期间, 第一类构造不被触发. 此外, 已有玩家实例的 `Follower` 也会使得第一类构造不被触发. 这应该是一个双重保险?
- 玩家携带的实例:

- 切板: 由于 GainFollower 方法会使得 Key 带有 Tags.Persistent. 因此切板后 Key 实例仍然在 Scene 当中, 玩家仍然带有 Key. (少部分 Follower 实例会设置 PersistentFollow = false, 此时就不会在拾取时自动带有 Tags.Persistent, 例如 (Generic)StrawberrySeed 类. 这导致切板后它们消失.)

- 非切板:

* 以 Level.TeleportTo 方法为例 (注意是原版的, 而不是 mod 的. 例如 Farewell 开头的奶奶暴毙剧情 CS10_Farewell. 或者 CS02_DreamingPhonecall, 虽然并没有调用 Level.TeleportTo, 但内容类似.), 大体内容是

```
Leader.StoreStrawberries(player.Leader);  
level.Remove(player);  
level.UnloadLevel();  
Session.Level = nextLevel;  
LoadLevel(introType);  
Leader.RestoreStrawberries(player.Leader);
```

StoreStrawberries 使得携带的草莓被存储到 Leader 类中的一个静态容器中, 并赋予这些草莓 Tags.Global. 然后 UnloadLevel 会导致所有不带有 Tags.Global 的实体卸载 (而 TransitionRoutine 则是 Tags.Persistent 也可逃过一劫). 然后调用 LoadLevel,

· LoadLevel 中, 在大部分实体都被加入后, 由于 IntroTypes 不是 Transition, 此时会调用 LoadNewPlayerForLevel, 这导致玩家实体是新创建的实例. 在构建好新的 Player 实例之后, 此时第二类构造会被调用, 使得玩家能够自动获得 session.Keys 中存储的那些 Key.

然后 RestoreStrawberries 再将草莓也归还给玩家, Tags.Global 也被移除.

* 死亡重生的例子: Player.Die 会使得调用 Leader.LoseFollowers(), 使得 Follower 不再 Persistent 且与 Player 解除绑定. 然后 PlayerDeadBody 在 DeathRoutine 结束时调用 PlayerDeadBody.DeathAction = Level.Reload, 使得关卡重启.

Level.Reload 会调用 Level.UnloadLevel(), 然后 Level.LoadLevel(Player.IntroTypes.Respawn). 仍然和之前一样, 会获得 session.Keys 中存储的那些 Key.

因此表现为, 草莓丢失, 但钥匙保留.

Key.RegisterUsed() 会使得钥匙被移除出 Followers 列表, 并移除出 session.Keys. (然而它仍然在 DoNotLoad 中) 因此之后对它的第一类和第二类构造都不会被调用.

Key 有 TransitionListener, 使得正在使用中的钥匙会被重新标记为 StartedUsing = false, 并重置 sprite.Rate 等. 不过它 (理所当然的) 不会重置 sprite.animationTimer (除非之前不在 idle 态, 在比较好的 cycle 下这是可能的). 所以虽然我们不能借此来获得一个恒定在 sprite.Rate = 3 下运作的 key, 但是依然可以借助 LockBlock 加速调整 key 的 cycle.

17.17.2 开锁

LockBlock 有个半径 60f 的 PlayerCollider, 玩家与之碰撞后会 (在碰撞的每一帧) 尝试对玩家的 Follower 中的尚未开始使用的 Key 调用 LockBlock.TryOpen(Player player, Follower fol).

TryOpen 检测 LockBlock 中心与玩家中心的连线是否与自身之外的 Solid 碰撞, 如果否, 那么将 Key 标记为 StartedUsing = true, 并给自身加上 UnlockRoutine(Follower fol).

UnlockRoutine 的第一帧就给自己加上 key.UseRoutine(Center + new Vector2(0f, 2f)), 并 yield return 1.2f. 之后也将自身 (LockBlock) 加入 DoNotLoad, 并调用 key.RegisterUsed(), 以及等待

```
while (key.Turning){
    yield return null;
}
```

这之后, 就有 LockBlock.Collidable = false, 以及再执行后续的 yield return sprite.PlayRoutine("open") 等.

因此, 在 yield return 1.2f 之后直接自杀, 那么重生后钥匙与钥匙门都会消失.

Key.UseRoutine 很长, 开始时设置 Turning = false, 结束时设置 Turning = true. 开锁的时间瓶颈就在于它的 UseRoutine.

此外, LockBlock 继承基类的 Platform 具有 Depth = -9000. 由于 Add(new Coroutine(key.UseRoutine(Center + new Vector2(0f, 2f)))) 发生在 LockBlock.Components.Update 中. 按照 ComponentList.Update, 更新开始前会设置 LockMode = LockModes.Locked, 然后对 components 逐个更新, 最后 LockMode = LockModes.Open.

在 LockModes.Open, 那就是直接添加/删除. 在 LockModes.Locked, 则是加入到 toAdd/toRemove 中, 然后 LockMode 的 setter 被调用时会再去处理这些 toAdd/toRemove.

因此按照时序, LockBlock.TryOpen 成功的那一帧, key.UseRoutine 并不运作.

Tween 类, 在我们这里用到的配置下 (Reverse = false, Mode = OneShoot), 在 Start 时会将 TimeLeft 初始化为 Duration. 然后每次更新 TimeLeft -= Engine.(Raw)DeltaTime (取决于参数配置). Percent = Max(0f, TimeLeft)/Duration. 然后 Eased = Easer(Percent). 当 TimeLeft <= 0f 时, 设置 Active = false.

完整的时序, 以下用 Depth 0 / -9000 / -1000000 来表明这分别是 Player / LockBlock / Key.Update:

- (1) 第 1 帧, Depth 0, LockBlock.TryOpen 成功. UnlockRoutine 加入到 LockBlock.Components 中.
- (2) 第 1 帧, Depth -9000, LockBlock.UnlockRoutine 将 key.UseRoutine 作为 Coroutine (以下记为 **Coroutine1**) 加入 LockBlock.Components 中.
- (3) 第 1 帧, Depth -9000, 由于时序, key.UseRoutine 并不运作.
- (4) 第 2 帧, Depth -9000, LockBlock.Update 中, key.UseRoutine 将 tween1 加入到 Key.Components 中. yield return tween1.Wait().

Tween1 是 CubeOut, Duration = 1f, Tween1.OnUpdate 中有 `sprite.Rate = 1 + t.Eased * 2f`. 此后没有别的改变 `sprite.Rate` 的事物.

- (5) 第 2 ~ N_1 帧, Depth -1000000, 先进行 `sprite.Update()`, 然后 `tween1.Update()`. 在第 N_1 帧结束时, `tween1.TimeLeft <= 0`, `Active = false`.
- (6) 第 $N_1 + 1$ 帧, Depth -9000, 此时 `tween1.Wait().MoveNext()` 为 false, 从 `Coroutine1` 的 `enumerators` 栈顶弹出, `key.UseRoutine` 来到栈顶.
- (7) 第 $N_1 + 2$ ~ N_2 帧, Depth -9000, 等待 `key.sprite.CurrentAnimationFrame == 4`.
- (8) 第 N_2 帧, Depth -1000000, `key.sprite.Update()`, 使得 `key.sprite.CurrentAnimationFrame == 4`.
- (9) 第 $N_2 + 1$ 帧, Depth -9000, `yield return 0.3f`.
- (10) 第 $N_2 + 2$ ~ N_3 帧, Depth -9000, 等待.
- (11) 第 $N_3 + 1$ 帧, Depth -9000, `key.UseRoutine` 将 `tween2` 加入到 `Key.Components` 中. `yield return tween2.Wait()`.
Tween2 Duration 为 0.3f, 负责 `sprite` 的旋转.
- (12) 第 $N_3 + 1$ ~ N_4 帧, Depth -1000000, `tween2.Update()`.
- (13) 第 $N_4 + 1$ 帧, Depth -9000, `tween2` 弹出, `key.UseRoutine` 回到栈顶.
- (14) 第 $N_4 + 2$ 帧, Depth -9000, 会启动一个 Duration = 1f 的 Alarm, 在倒计时结束后会有一个 Duration = 1f 的 tween, 负责设置钥匙的顶点光的渐变暗效果, 完成后还会将 Key 从场景移除 (不过此时早已开门). 并 `yield return 0.2f`;
- (15) 第 $N_4 + 3$ ~ N_5 帧, Depth -9000, 等待.
- (16) 第 $N_5 + 1$ 帧, Depth -9000, `key.Turning = false`. 由于 `UnlockRoutine` 此时已经更新过, 需要等到下一帧.
- (17) 第 $N_5 + 2$ 帧, Depth -9000, `UnlockRoutine` 更新, 使得 `LockBlock.Collidable = false`.

在 `Engine.TimeRate = 1f` 的情况下, $N_1 = 60 + 1$, $N_1 + 1 \leq N_2 \leq N_1 + 27$, $N_3 = N_2 + 18 + 1$, $N_4 = N_3 + 18$, $N_5 = N_4 + 2 + 12$. 于是,

总共 $(115 + \text{cycle})$ 帧, 这里 $\text{cycle} = N_2 - N_1 - 1 \in \{0, 1, \dots, 26\}$.

`cycle` 的具体数值取决于 `sprite.animationTimer`. 在 `sprite.Rate` 达到 3f 之后, 由于 `idle` 态的 `currentAnimation.Delay = 0.1f`, 因此每个 `CurrentAnimationFrame` 都是 2 帧. 而 `idle` 态总共 14 帧 (0 ~ 13), 因此最多因动画而耽误 26 帧.

现在我们用

$$t := \left(\text{sprite.animationTimer} \times 10f + \text{sprite.CurrentAnimationFrame} \right) \bmod 14$$

作为 `sprite` 的 (约化) 整体计时器. 有 $t \in [0, 14)$. 那么每帧 t 增加 $\frac{\text{Rate}}{6}$.

设

- TryOpen 的那一帧 (第一帧)(在这一帧更新后, 也即信息面板上看到的) 的约化时间为 t_1 ;
- 最早的判定 `key.sprite.CurrentAnimationFrame == 4` 的那一帧 (第 63 帧) 的约化时间为 t_2 (在判定时, 也即实际上是第 62 帧时的信息面板上的);

那么 $t_2 - t_1 = 25.3320 = -2.6680 \pmod{14}$. 当 $t_2 \in [4, 5)$ 时, 能立即开门. 对应于 $t_1 \in [6.6680, 7.6680)$ 时, 钥匙在最快的 cycle.

由于原先 `sprite.Rate = 1`, 在等 cycle 段 `sprite.Rate = 3`, 所以之前每快 3f = 少 1f key cycle. 不过, 如果使用过 Key 的第二类构造, 那么 `animationTimer` 自然被重置了; 反之, 如果是 Transition, 则保留.

17.17.3 杂项

OnPlayer 还会调用 `session.UpdateLevelStartDashes()`, 使得 `session.DashesAtLevelStart = session.Dashes`. 这里 `session.Dashes` 是实时统计的冲刺数, `session.DashesAtLevelStart` 是每一面开始时 (以及例如获得钥匙, 磁带时) 同步自 `Dashes` 的. 设置这两项的用意好像是, 关卡内死亡的时候, `Dashes` 会重新回到 `DashesAtLevelStart`. 因此这样可以使得 `Dashes` 数不计入那些失败的尝试? 虽然这还是有点意义不明.

OnPlayer 也设置 `key.Depth = -1000000`.

17.18 岩浆块

泡芙佬写了份文档.

<https://github.com/lnf-24/BounceBlockExplanation>

在泡芙佬说的第四阶段 (停留 2f), 会通过 `BounceBlock.ShakeOffPlayer` 将 Maddy 强制退回 `StNormal`, 速度设为 `bouncelift`, 并给予狼跳时间. 注意这需要此时 Maddy 仍骑乘在 `BounceBlock` 上, 所以这第一帧必须按着抓键. 否则只会触发松抓键吃到 (上一帧产生的) `LiftBoost`, 而不会获得狼跳帧.

此时水平速度为 (这一帧的) `LiftBoost` (我没完全翻阅过, 且只观察了向右的例子), 并回到 `StNormal`, 且有狼跳帧, 那么再在狼跳帧跳就可以再获得 $40 + \text{LiftBoost}$.

与之对比, 如果在岩浆块还没完全碎裂之前跳, 那么至多就是 $130 + \text{LiftBoost}$. 考虑到 `LiftBoost` 往往能达到 150, 这样做亏了不少速度.

17.19 漂浮块

或者叫月球块.

`FloatySpaceBlock` 的运动: 它们是成组运动的, 运动跟随它们组的主宰者 (`bool MasterOfGroup`). 对于主宰者, 垂直方向上暂且不提. 水平方向上一般不运动, 除非被冲击. 在 `dashEase <= 0.2f` 时, 在 `OnDash` 中可以获得 `dashEase = 1f`, `dashDirection = direction`. 在 `Update()` 中, 会 `dashEase = Calc.Approach(dashEase, 0f, Engine.DeltaTime * 1.5f)`, 也就是一般来说每帧 $-0.025f$, 然后 `MoveToTarget()`. 在 `MoveToTarget()` 中, 会根据 `dashEase` 确定 `vector`, 然后将组里的每个平台, 移至它的原始位置 + `vector`.

这导致 `dashEase` 和 `dashDirection` 会不依赖于历史地, 完全决定漂浮块的水平位置.

先移动有 rider 的,再移动没 rider 的. (呃,这可能是为了避免 rider 被不合理地夹死?). $vector = Calc.YoYo(Ease.QuadIn(dashDirection * 8f, \text{这里 } QuadIn(x) = x^2,$

```
public static float YoYo(float value){
    if (value <= 0.5f)
    {
        return value * 2f;
    }
    return 1f - (value - 0.5f) * 2f;
}
```

这是一个分段线性函数. 节点是 (0,0), (0.5, 1), (1, 0).

于是当再次能冲击时 ($dashEase \leq 0.2f$), 漂浮块基本已回到原位. 况且多次冲击并不能带来新东西... 没什么用.

如果月球块上的弹簧被触发, 则会触发 `FloatySpaceBlock.OnStaticMoverTrigger()`. 对于水平弹簧, 这等效于产生了一次 `OnDash`. 特别的, 这会使得月球块的位置瞬间移动到几乎原位的地点. (由于 $dashEase = 1f - 0.025f$, 还会沿着冲击方向退 $0.79f$, 由于原位的亚像素是 0.5 , 这使得移动到原位再沿冲击方向退 $1px$ 的地方.)

可知月球块被冲击时, 最多回退 $8px$. 而弹簧被本身宽 $6px$, 因此触发弹簧时会被移动到第 $7px$ 处. 当月球块恰好回退 $8px$ 时触发弹簧, 则月球块瞬间回弹 $7px$, 恰好能推动 `Maddy`. 产生的 `LiftSpeed` 在 400 左右, 于是 `LiftBoost = 250`.

若有风, 且风晚于 `Maddy` 自身的更新 (也就晚于被弹簧改变位置), 那么被弹簧改变位置之后, 判定风的反方向 $3px$ 处没有 `Solid`, 因此会有风吹动. 若风吹动 + 自身亚像素 $> 1f$, 则 `Maddy` 多移动了 $1px$, 就无法触发 `Spring moon boost`.

17.19.1 Bubble storage 泡泡储存

Alaska [081].jpg

泡泡储存: 这里我们用的是 `FrostHelper` 的 `CustomBooster`, 注意它的实现方式和原版并不是一样的 (玩家状态机多添加了 `Custom Boost` 和 `Custom Red Boost`, 然后配套地做) 泡泡有一个 `bool` 字段 `StartedBoosting`, 在进入泡泡的时候 `StartedBoosting` 会设置成 `true`, 在类似冲刺 (i.e. 启动泡泡也算) 的事件发生时, 会找到 `StartedBoosting` 为 `true` 的泡泡, 认为是由它启动的, 将它的 `StartedBoosting` 设为 `false` 且一旦找到一个, 就不继续找了. 注意: 泡泡状态结束并不会设置 `StartedBoosting` 为 `false`. 也就是说, 常规情况下, 是通过在 `Custom(Red)BoostUpdate` 中启动泡泡, 达到了设置 `StartedBoosting = false` 并退出 `Custom(Red)Boost` 的效果.

现在发生了什么: 我们首先到中间的泡泡 (泡泡 1), 触发望远镜的同时接触泡泡, 由于望远镜的 `Coroutine`, 我们立刻回到 `StDummy` 并随后回到 `StNormal`. 这里唯一关键的是, 我们结束了 `CustomBoost`, 但没有发生冲刺事件. 此时泡泡 1 `StartedBoosting = true` 然后来到右边的泡泡 (泡泡 2), 泡泡 2 `StartedBoosting = true`. 常规地启动泡泡 2, 调用冲刺事件. 由于实际深度, 泡泡 1 比泡泡 2 更先被加入 `Tracker`, 因此先找到泡泡 1, 发现它

StartedBoosting = true, 于是认为是由泡泡 1 启动的泡泡 1 消失了所以我们的路径上不会再撞到它到达左边, 再次按冲刺, 发现泡泡 2 StartedBoosting = true, 于是认为是由泡泡 2 启动的, 因此不会使得 SwapBlock 响应

原版泡泡储存: 原理基本一致, 但注意原版标记泡泡用的是 Player 的 CurrentBooster 字段, 因此只能储存一个泡泡. 而不像 FrostHelper 可以储存任意个. 不过也有好处, 那就是原版的储存是可以跨房间的, FrostHelper 的这个不行

此外还有很神奇的一件事情, 向上弹簧会取消 CurrentBooster, 但水平的弹簧不会! 因此水平弹簧也能用来储存 (原版) 泡泡. 不过这些取消毕竟都需要在 StBoost 下, 而 StBoost 下很难离开太远, 所以对位置要求很高.

不知道谁写的:

First thing:

Cornerglide and Boost State Retention are two different mechanisms, as far as I am aware.

Boost State Retention (Bubble State Retention) I understand, Cornerglide I'm not sure on.

Boost State Retention: I'm going to analyze the frame-by-frame information for the Bubble State Retention. Technically, this isn't actually "Retaining" the boost state, rather, it's retaining the value of CurrentBooster in any event:

When the player touches the Booster, the following things occur.

The player's state is set to "BoostState" (in code - StBoost, or a constant = 4). This does a number of things. The Current Booster and the Last Booster is set to the touched Booster.

From there, we're gonna gloss over how the cornerglide works, but you cornerglide into the Spring. This is normally, you'd expect an Upwards spring to disable the CurrentBooster by setting it to null, which it does. So when hitting a side spring, it doesn't reset the CurrentBooster, and also sets you to Normal State (Normal). This also means that anything that resets your player state to normal that doesn't reset the Current Booster.

the next time you dash, the dash is one of the few things that cares about CurrentBooster, and as such it resets the CurrentBooster. Cornerglides

First off, there is a handy dandy variable called "wallSpeedRetained" which, i think we can all guess what it does. When you hit a wall (there's a little bit more specifics on what constitutes "hitting a wall" but whatever). When you exit the "normal gameplay state" into another state like dashing, the Timer is set to 0. This means that when you dash, the dash is one of the few things that cares about CurrentBooster, and as such it resets the CurrentBooster.

Cornerglides with Boosters

Boosters can be moved around in, assuming you can have the speed to maintain it. It pulls you back to the wall. However, assuming you're not exiting the "normal gameplay state" which sets the wallSpeedRetention time to 0. The distance travelled per frame is actually subtracted by 1.33px (1.33 * GameSpeed, 80px/s * TrueGameSpeed).

Another note: Boosters teleport you back to their center when you exit their state into another state, This is what I have 100% confirmed so far.

CG 接泡泡的情况下, 一方面 StBoost 下没有速度衰减, 另一方面由于泡泡有 80 的速度将 Maddy 拉向自身, 导致即使撞墙也会在下一帧触发 retention 而恢复速度, 因此 CG 接泡泡的水平速度可以一直储存至泡泡开始自动冲刺. 自动冲刺时, 如果前一帧面前没有墙, 则自然是基于此速度开始冲刺. 否则, 以 beforeDashSpeed = 0 开始冲刺. 因此有墙的时候这个速度能否保留至冲刺, 是看脸的, 一半概率能成.

17.20 Collider 相互碰撞的检测机制

注意 Rect 本身并不是 Collider. 以下都假定宽度, 高度, 半径等非零, 不然我需要重新细细读一遍代码...

Rect vs Circle: 等价于闭矩形与开圆盘是否重合.

Rect vs Point: 点是否在 $[x, x + w) \times [y, y + h)$ 中.

Rect vs Rect: 开矩形是否相交.

Hitbox vs Hitbox: 即 Rect vs Rect.

Hitbox vs Circle: 即 Rect vs Circle.

Grid vs Rect: 实现在 int Rect 的时候才比较合理. 大体是将 Rect 粗粒化为一个个 8*8 的矩形, 然后查询 VirtualMap<bool> Grid.Data. 这等效为将 Grid 变为一个个 8*8 的矩形, 然后看是否有像素重合.

Hitbox vs Grid: 先把碰撞箱转换成坐标都是整数的矩形, 然后 Grid vs Rect.

17.21 镜头

Camera

镜头移动速度?

哦镜头移动是写在 Player.Update() 里的.

暂停结束后的 NoControl 会移动吗

屏幕大小是 320f * 180f.

机制比较复杂, CameraTarget 需要监视以下内容:

Player.level.CameraOffset: Player.CameraAnchor: Player.CameraAnchorIgnoreX: Player.CameraAnchorIgnoreY:
Player.CameraAnchorLerp: Player.EnforceLevelBounds: Player.level.CameraLockMode:

此外 killbox 等也会影响.

在比较简单的情形下, CameraTarget 就是 $\text{Vector2}(\text{base.X} - 160f, \text{base.Y} - 90f) + \text{level.CameraOffset}$ 然后再被左右上下 bound 住.

然后在大部分 State 下, 镜头会位移 $(\text{CameraTarget} - \text{level.Camera.Position}) * 0.0739$.

由于拖曳, 这使得调整镜头的策略会比较微妙.

17.22 整数坐标不再是整数 (offgrid)

<https://discord.com/channels/403698615446536203/519281383164739594/1065511990480605184>

通过 `StCassetteFly` 过程中 `cutscene`, 可以使得整数坐标不再是整数.

这会产生一些奇妙的效应, 比如视觉上看碰撞箱变大 (实际上是因为其左右两端不再是整数位置). 由于 `Hitbox` 与 `grid` (`SolidTiles`) 的碰撞方式和与 `Hitbox` 的碰撞方式不一样, 所以神奇的事情会发生.

以下内容是对一个提问的回答:

不是我发现的, 我只是dc搬运工+原理解释

好像没正式名称, 大概是因为还没发现什么实际应用, dc 上大概会以 `offgrid` 为关键词讨论此事

玩家的位置 (`ExactPosition`) 分为整数部分 (`Position`) 和小数部分 (`movementCounter`)

整数部分 `Position` 处理了几乎所有游戏逻辑相关, 而小数部分则是使得运动看起来比较 "连续"

通常情况下, 游戏在移动玩家时会调用特定的方法, 这会同时处理整数部分和小数部分

然而, 在 `StCassetteFly` 中间的飞行段, 游戏用的是, 直接设置玩家的 `Position`

如果此时通过跳过剧情等手段打断 `StCassetteFly`, 我们就能得到非整数的 `Position` (以下称这种情形为 `offgrid`)

上面说了, `Position` 是游戏逻辑相关的, 因此变为非整数, 会出现一些问题

我们在游戏中通常见到的固体, 可分为两大类

一种是砖块/墙体, 它们的碰撞体是 `Grid`, 也就是视作网格上的一个个砖块

另一种就是剩下的那些, 它们的碰撞体是碰撞箱 (`Hitbox`), 也就是矩形

玩家的碰撞体则也是碰撞箱, 大部分实体的碰撞体也是碰撞箱

碰撞箱与碰撞箱的碰撞, 和碰撞箱与 `Grid` 的碰撞, 它们是不完全一样的

何时一样: 当碰撞箱的坐标都是整数的时候, 也就是日常情形

因此 `offgrid` 的时候表现会比较奇特

(具体细节: 碰撞箱 vs 碰撞箱 = 两个坐标都是实数的矩形, 是否重叠 (相切不算)).

碰撞箱 vs `Grid` = 先把碰撞箱转换成坐标都是整数的矩形, `Grid` 等效为一个个 `8*8` 的矩形, 然后看矩形是否重叠

那么真的穿墙了吗? 是或不是.

玩家并没有真正地卡进墙里, 但如果你将砖块简单地等同为一个个 `8*8` 的碰撞箱, 那么玩家确实卡进了这些碰撞箱

备注: 穿墙的视觉效果基于 `CelesteTAS`. `TAS mod` 修改了 `Hitbox` 的渲染,

使得 整数坐标的碰撞箱 vs 非整数坐标的碰撞箱 的视觉效果, 也是符合预期的 (一般情况下是玩家以外的实体卡在墙里). 在原版自带的 `debug` 模式中, 你并不会看到穿墙.

17.23 LastAim

`Input.GetAimVector(..)` 中给 `LastAim` 赋予八个主方向的值. 但是, 它实际上并不是将 360 度均等分为八份. 而是: 如果在上半平面, 则宽容度为 17.5 度, 如果在下半平面, 则宽容度为 22.5 度. 若与正上下左右这四个方向的夹角在宽容度以内, 那么 `LastAim` 就是这个方向. 否则是斜的四个方向. 这样做使得斜上冲的角度容错更大 (55 度的区间).

游戏内部并没有直接存储角度/弧度, 而是存储一个 `Vector2 Aim`. 不过在以上得到 `LastAim` 的过程中, 确实有将 `Aim` 用 `Atan2` 转换成弧度, 值在 $(-\pi, \pi]$. `Math.Atan2(double y, double x)` 表示的是与 x 轴正方向的夹角 (逆时针为正). 不过由于游戏的 y 轴与数学上的 y 轴相反, `Aim` 转换成的弧度是按顺时针方向的, 以正右为零点.

`Studio` 使用的 `AnalogMode` 则以正上为零点, 同样按顺时针方向, 采用角度制. 这样选取零点可能是因为 `Studio` 想要避免负数角度的使用, 同时又要尽可能地方便用户, 需要将右半平面的角度连续地放在一起.

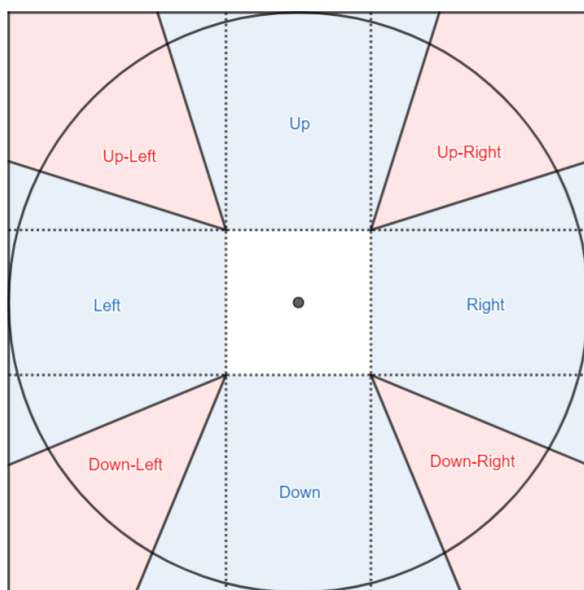


图 29: 八个主方向对应的范围

17.24 idu 和 didu

个人理解 `du` 与 `u` 的区别在于, 打断冲刺的时间点, 在触发 `u` 之前还是 `u` 之后 (没打断则视为打断时间为无穷远).

按照这个划分 `gu cancel` 显然不算 `du`, 常规 `ultra` 也不算 `du`.

`instant` 在于是否在理论上能够撞地的第一时间撞地 (e.g. 常规 `idu` 在速度 7.5 或 0 时撞地, 斜下冲抓 `Holdable StPickUp` 一结束就撞地). 呢不过有人会认为是横向发生移动的第一帧要撞地, 这样的话抓 `Holdable` 就不是 `idu`.

而 `idu` 和 `didu` 的区别在于, 撞的那个地板是否是你 `cb` 的那个地板. 当然这个概念分划只对以 `cb` 打断的 `du` 才有效.

但 `didu` 名字里 `instant` 就在于, 它也是符合之前所说的 `instant` 的特征.

不过命名不重要, 不影响交流就行.

17.25 Spinner Cycle

大坑.

最早的 Spinner Stunning 的案例 (发现于 2022/12/25)

<https://discord.com/channels/403698615446536203/598945702554501130/1056636810513612870>

它是那么地浑然天成, 要是需要再多一个暂停, 可能都不会有这个发现.

极端使用法: 不断暂停, 使得完全压制住处在某一个 spinner cycle 的刺, 让它们一直无法激活 (发现于 2022/12/25)

https://media.discordapp.net/attachments/519281383164739594/1056675884045717544/IMG_0101.png?width=

由以下决定:

```
if (base.Scene.OnInterval(0.05f, offset))
{
    Player entity = base.Scene.Tracker.GetEntity<Player>();
    if (entity != null)
    {
        Collidable = Math.Abs(entity.X - base.X) < 128f && Math.Abs(entity.Y - base.Y) < 128f;
    }
}

public bool OnInterval(float interval, float offset)
{
    return Math.Floor((TimeActive - offset - Engine.DeltaTime) / interval) < Math.Floor((TimeActive
```

取决于场景 (房间) 的加载时间, 以及 spinner 自身的 offset.

顺便扯下卸载, 分两种模式, 远距离 (128*128 以外) 分 3 组, 通过 base.Scene.OnInterval(0.05f, offset) 设置 collidable.

近距离通过 if (base.Scene.OnInterval(0.25f, offset) && !InView()) Visible = false; 然后下一帧 if (!Visible) Collidable = false; 来卸载. 分 15 组.

https://docs.google.com/document/d/1qV5o7789RxQSji_vpwGeKfSeua_yJXdc/

collidable 是通过影响 Collide.Check 来做到是否有伤害判定的. 在 PlayerCollider.Check(Player player) 中, 会检测 player.CollideCheck(base.Entity), 而 CollideCheck(Entity other) 即 Collide.Check(this, other).

```
public static bool Check(Entity a, Entity b)
{
    if (a.Collider == null || b.Collider == null)
    {
```

```

        return false;
    }
    if (a != b && b.Collidable)
    {
        return a.Collider.Collide(b);
    }
    return false;
}

```

呃,在某些极端的情形下,你会看到一些奇怪的加载情况. 但这是符合程序代码的.

<https://discord.com/channels/403698615446536203/519281383164739594/1059557245756653768>

由于非常经典的 Maddy 先于其他大部分实体更新的原因,这一帧激活的刺在下一帧才有伤害判定.

由于 visible 和!visible 是两个分支,因此可以在第 0 帧恰好从不可见变为可见,然后在第 3 帧才 Pause

17.25.1 Spinner Drift

Spinner Drift: 指颜色跳到另一个组的现象.

比较容易理解的情形: 由于浮点误差,相当于每次加的 DeltaTime 都偏大/偏小. 导致 0.05f 需要少一帧/多一帧的时间就可以达成. 也就跳到前一个/后一个组. 这种情况下,初末的 $\text{Floor}((\text{TimeActive} - \text{offset} - \text{DeltaTime}) / 0.05f)$ 只相差 1.

剩余情形: 初末的 $\text{Floor}((\text{TimeActive} - \text{offset} - \text{DeltaTime}) / 0.05f)$ 可以相差 0 或 2. 由于浮点误差,导致 $(\text{TimeActive} - \text{offset})$ 和 $(\text{TimeActive} + \text{DeltaTime} - \text{offset} - \text{DeltaTime})$ 不一样. 参考模拟器模拟的 Spinner Drift 图 30, 则一切自然明了. 喔, Minty 的论证似乎更加解明本质. 按它的说法, CycleLength = 1 - 5 全是某个单一的 positive/negative current/previous indexer drift 造成的结果. 那 CycleLength = 6 - 7 则是两者复合的结果?

值得注意, CelesteTAS 的 Actual Collide Hitbox 会对 collidable 与否的透明效果造成延迟一帧的影响. 平时做 TAS 当然可以开着. 研究机制的时候最好关掉.

顺带一提,圆刺的内部是实心的,人物碰撞箱当然也是实心的. 呃,据说 CelesteTAS 渲染出来的圆刺碰撞箱和 Debug 自带的碰撞箱是不一样的(后者不对). 那 CelesteTAS 渲染是基于啥逻辑? 另外,圆与点的碰撞,圆与矩形的碰撞,是分开来写的. 目前的渲染,是否确实起到了若有像素重合,则实际上碰撞到的效果?

防治措施: 如果只需要 stun 一个,那么 Spinner Cycle + n f 就够了,这里 n 大概不超过 9 (Pause 期间 TimeActive 走的长度)

17.25.2 Lighting Cycle

卸载

加载

```

Frame=9833f, TimeActive=163.9207611, offset=0.4207586944, CycleLength=1, AverageActualDeltaTime= 0.01666259766
frame, TimeActive, (TimeActive - offset - DeltaTime)/0.05, (TimeActive - offset)/0.05
0f, 163.9207611, 3269.666714, 3270.000048
1f, 163.9374237, 3269.999966, 3270.3333
=====
Frame=61479f, TimeActive=1024.46228, offset=0.04562517256, CycleLength=2, AverageActualDeltaTime= 0.01672363281
frame, TimeActive, (TimeActive - offset - DeltaTime)/0.05, (TimeActive - offset)/0.05
0f, 1024.46228, 20487.99977, 20488.3331
1f, 1024.479004, 20488.33424, 20488.66757
2f, 1024.495728, 20488.66871, 20489.00205
=====
Frame=1f, TimeActive=0.0666667968, offset=0.2544023097, CycleLength=3, AverageActualDeltaTime= 0.0166669548
frame, TimeActive, (TimeActive - offset - DeltaTime)/0.05, (TimeActive - offset)/0.05
0f, 0.0666667968, -4.088044241, -3.754710257
1f, 0.08333349228, -3.754710332, -3.421376348
2f, 0.1000001878, -3.421376422, -3.088042438
3f, 0.1166668832, -3.088042513, -2.754708529
=====
Frame=9376f, TimeActive=156.2892914, offset=0.5892819166, CycleLength=4, AverageActualDeltaTime= 0.01666259766
frame, TimeActive, (TimeActive - offset - DeltaTime)/0.05, (TimeActive - offset)/0.05
0f, 156.2892914, 3113.666855, 3114.000189
1f, 156.305954, 3114.000107, 3114.333441
2f, 156.3226166, 3114.333359, 3114.666693
3f, 156.3392792, 3114.666611, 3114.999945
4f, 156.3559418, 3114.999863, 3115.333197
=====
Frame=5160f, TimeActive=86.0539093, offset=0.08725241572, CycleLength=5, AverageActualDeltaTime= 0.01667022705
frame, TimeActive, (TimeActive - offset - DeltaTime)/0.05, (TimeActive - offset)/0.05
0f, 86.0539093, 1718.999804, 1719.333138
1f, 86.07057953, 1719.333208, 1719.666542
2f, 86.08724976, 1719.666613, 1719.999947
3f, 86.10391998, 1720.000017, 1720.333351
4f, 86.12059021, 1720.333422, 1720.666756
5f, 86.13726044, 1720.666826, 1721.00016
=====
Frame=8209736f, TimeActive=131071.9609, offset=0.04745628312, CycleLength=6, AverageActualDeltaTime= 0.01692708395
frame, TimeActive, (TimeActive - offset - DeltaTime)/0.05, (TimeActive - offset)/0.05
0f, 131071.9609, 2621437.936, 2621438.27
1f, 131071.9766, 2621438.249, 2621438.582
2f, 131071.9922, 2621438.561, 2621438.895
3f, 131072.0156, 2621439.03, 2621439.363
4f, 131072.0312, 2621439.343, 2621439.676
5f, 131072.0469, 2621439.655, 2621439.988
6f, 131072.0625, 2621439.968, 2621440.301
=====
Frame=4015431f, TimeActive=65535.94141, offset=0.7899410725, CycleLength=7, AverageActualDeltaTime= 0.01618303545
frame, TimeActive, (TimeActive - offset - DeltaTime)/0.05, (TimeActive - offset)/0.05
0f, 65535.94141, 1310702.696, 1310703.029
1f, 65535.95703, 1310703.008, 1310703.342
2f, 65535.97266, 1310703.321, 1310703.654
3f, 65535.98828, 1310703.633, 1310703.967
4f, 65536.00781, 1310704.024, 1310704.357
5f, 65536.02344, 1310704.337, 1310704.67
6f, 65536.03906, 1310704.649, 1310704.982
7f, 65536.05469, 1310704.962, 1310705.295
=====

```

图 30: 浮点误差与 Spinner Drift

呃, 我怀疑它可能是每次死亡重生/跨房间, 都会创建闪电. 这能解释为什么时间尽头然后重生/跨版之后所有闪电都有判定: 因为创建的时候 `Collidable = true` (Entity 类默认如此).

与之对比, `CrystalStaticSpinner()` 在创建的时候自带 `Visible = false`, 这在 `Update()` 的时候不需要 `OnInterval` 就会判定 `Collidable = false`.

17.25.3 DustStaticSpinner Cycle

卸载

加载

DustGraphic, Established

一帧最多加载 25 个动画

17.26 (位置) 浮点数操纵

参考文末的引用

Celeste: The Art of Floating Point Manipulation

17.27 浮点数

17.27.1 微小速度

1,D,J; 3,L,J; 3,R,J; 可以造出微小速度.

130 速度 + 按住反方向也有.

17.27.2 浮点误差与 Spinner Cycle

game uses 0.05s in code instead of 3 frames to check whether the game is on the cycle interval and toggle the spinner's collision, the `OnInterval` method is `public bool OnInterval(float interval, float offset) return Math.Floor((this.TimeActive - offset - Engine.DeltaTime) / interval) < Math.Floor((this.TimeActive - offset) / interval);`

`TimeActive` is time elapsed since the scene is activated (game speed change included, e.g. oshiro boss slowdown), `Engine.DeltaTime` is 0.0166667 if game runs at 100 percent speed and no slowdown in level, `interval` is 0.05 and `offset` is a random number between 0 and 1 when the room loads (so it's room name seeded)

呃, 注意到用的是 `DeltaTime` 而不是 `RawDeltaTime`, 所以改变 `TimeRate` 的行为应该也会改变 `spinner cycle`. (e.g. `BadelineBoss`)(这个确实是被验证了)

浮点误差的影响很大. 如果没有误差, 那么应该每 3f 产生偏差 $3 * 0.0166667 - 0.05 = 0.0000001 = 10^{-7}$. 但浮点数加法是有误差的, `TimeActive + DeltaTime` 的误差取决于 `TimeActive` 的大小, 基本上误差与 `TimeActive` 成反比 (负一次. 这大概是由于浮点数加法的原理). 当 `TimeActive` (以秒为单位的情况下) 是个位数时, 每帧产生的误差大约是 10^{-7} 级别. 两位数时 (一般 TAS 会达到的长度), $10^{-7} \sim 10^{-6}$ 级别. 三位数时, $10^{-6} \sim 10^{-5}$ 级别.

<https://discord.com/channels/510279444368457730/510606000454107148/1057291714995421194>

<https://discord.com/channels/403698615446536203/519281383164739594/1057358606955204658>

<https://discord.com/channels/403698615446536203/754495709872521287/1058740212433424394>

整体上来说, TimeActive delta 是增大的, 因此 TimeActive 比较大的时候整体的 spinner drift 的方向是统一的 (部分刺会有其他地方的浮点数误差造成的一些偏移)

17.27.3 时间尽头

由于浮点数加法的误差, 当 TimeActive 特别大时, $\text{TimeActive} + \text{DeltaTime} (=0.0166667) = \text{TimeActive}$, 导致 TimeActive 不再变化!

DeltaTime 受 TimeRate 的影响, 但 TimeRate = 1 - 1.8 时, 上限都是 TimeActive = 524288.0 (0x49000000)(219). 这大约是 118h. (由于误差, 并不是 $524288/3600=145\text{h}$)

如果 TimeRate < 1, 这个上限值会降低一些.

在时间尽头下, OnInterval 要么无法触发, 要么每帧都在触发. 因此圆刺要么一直试图加载 (指判定 collidable)(但你依然可以用超出镜头范围的方法避开它.), 要么一直保持之前的状态 (可能 collidable 也可能不).

呃, 似乎人们叫它 Spinner Freeze.

我不知道这个比例是不是 2/3. 人们大概被 (最初版本的) 颜色误导以为三种颜色是均匀的, 但是可能并不. update: 尽管现在是在 524288f 处, 但通过实验可以发现这个比例仍然是 2/3. (严格来说, 在任意的 TimeActive 处, 由于浮点数误差, 更新的那一批都不是占 1/3, 大概会是 0.3333332 - 0.3333334 这样的比例)

在时间尽头的情况下, 由于构造函数和更新函数的不同, (以下内容记得关掉 Actual Collide Hitbox), 对于 OnInterval(0.05f, toggleOffset) 不能触发的那一批, 关键时间点是切版 (这里暂不考虑在房间内呆到 524288 和房间内死亡的情形):

Lightning: 第一帧 (显示 NoControl (39)) 是 collidable and visible, 如果能触发 OnInterval(0.25f, toggleOffset) 则在 NoControl 结束后的第一帧起, 每一帧若 Collidable 则 Collidable 变为 InView(), 若 uncollidable 则永远 uncollidable; 否则永远 collidable.

DustStaticSpinner: 第一帧是 collidable and visible, 永远不变.

CrystalStaticSpinner: 第一帧 (NoControl (39)) 是 collidable and invisible, 由于带有 Tags.TransitionUpdate 标签因此第二帧 (NoControl (38)) 变为 uncollidable.

对于 OnInterval(0.05f, toggleOffset) 能触发的那一批, 切版不再那么重要, 讨论一般情形:

Lightning: 若它能触发 OnInterval(0.25f, toggleOffset), 则 Collidable 严格等于 InView() (因此还存着 Maddy 通过超出屏幕穿闪电的可能); 若它不能, 则永远 Collidable.

DustStaticSpinner: 初始 Collidable. 在 Sprite.Established 之后, Collidable 严格按照与 Maddy 的横竖距离是否都小于 128 px. 存在穿煤球的可能性.

CrystalStaticSpinner: 如果能触发 OnInterval(0.25f, offset), 则每帧更新 invisible/visible, collidable 由 InView() 和 128 px 共同控制, 比较复杂, 存在穿圆刺的可能; 如果不能触发 OnInterval(0.25f, offset), 则每帧 invisible and uncollidable 并更新 visible, 若 visible 则永远 visible 且这之后 Collidable 严格按照与 Maddy 的横竖距离是否都小于 128 px, 存在穿圆刺的可能.

我列了个表格, 描述了时间尽头之后, 切版之后, 各个 Hazard 的行为. 因为偷懒就先用插图代替了 图 31:

When TimeActive = 524288, after room transition:				
In the following, bool NearPlayer() = Abs(entity.X - Player.X) < 128f && Abs(entity.Y - Player.Y) < 128f				
Lightning	15f Check \ 3f Check	Yes	No	
	Yes	Collidable = InView()	Collidable = InView() in first NoControl frame, forever	
	No	Collidable forever		
Dust Bunny	Sprite.Established \ 3f Check	Yes	No	
	Yes	Collidable = NearPlayer()	Collidable forever	
	No	Collidable		
Crystal Spinner	3f Check No	Uncollidable forever		
	3f Check Yes	15f Check \ Visible	Yes	No
		Yes	Visible = InView(), Collidable = NearPlayer()	Uncollidable, and checks Visible = InView() every frame
		No	Visible forever, Collidable = NearPlayer()	

图 31: 电网, 煤球, 圆刺在时间尽头的行为

(这个表格有 bug, 看新版或者看 Minty 的)

如果不是切版进的, 还会需要更多讨论. 这里暂且不提.

17.28 暂停

暂停相关应该是在 `level.update()` 里面起作用的. 游戏计时器则是 `Level.UpdateTime()`. `TimeActive` 则是基类的 `Scene.BeforeUpdate()` 里更新. `UpdateTime()` 基本只要计时器开始了且没结束, 就会走. `TimeActive` 则是需要!`Paused`.

进入暂停是 `Pause()`. 它会创建一个叫 `menu` 的 `TextMenu` 类实体. `TextMenu` 类带有 `Tags.PauseUpdate`.

在 `Level.Update()` 中,

1. (大概) 首先会进行 `Scene.BeforeUpdate()`. 如果 `Paused = false`, 则 `TimeActive += Engine.DeltaTime`.
2. 若 `unpauseTimer > 0` (取消暂停后的那段时间): 则倒计时 -1, 并 `UpdateTime()`, 然后直接 `return`.
3. 检测是否按下暂停 (且 `CanPause`), 如果是的话就 `Pause()`. 这会产生一个叫 `menu` 的 `TextMenu` 类实体, 并使得 `Paused = true`.
4. `UpdateTime()`.
5. 如果 `Paused`, 则对带有 `Tags.PauseUpdate` 标签的实体进行 `Update`. 特别的, `menu` 带有此标签. `menu` 的更新里包括了按下取消则 `Paused = false`, 并设置 `unpauseTimer = 0.15f`.
6. 如果在前一步判定的时候, 不是 `Paused`, 那么进行 `base.Update()`. 这会将 `Scene` 里的所有实体更新.
7. 以上略去了 `SkippingCutscene`, `Frozen`, `Transitioning`, `RetryPlayerCorpse`(自杀尸体) 相关的, 它们与以上更新过程纠缠在一起. 参见源码.

`UpdateTime()`: 游戏计时器 (`Level.Session.Time`) 相关.

于是对于 1,S; 1,O; 9; 2

- 1,S: 先 `Scene.BeforeUpdate()`, 增加 `TimeActive`. 然后判定暂停, `Paused = true`. 最后因为暂停了所以判定不进行 `base.Update()`.
- 1,O: 由于 `Paused = true` 而没有进行 `Scene.BeforeUpdate()`, `TimeActive` 不变. 然后因为 `Paused = true`, 更新带有 `Tags.PauseUpdate` 标签的实体, 特别的取消了暂停. `Paused = false`, 并设置 `unpauseTimer = 0.15f`. 因为前面进了 `Paused` 分支所以不进行 `base.Update()`.
- 9: `Paused = false` 所以 `Scene.BeforeUpdate()`, 增加 `TimeActive`. 因为 `unpauseTimer > 0`, 而直接 `return`. 自然不会进行 `base.Update()`
- 2: `unpauseTimer <= 0`, 一切按正常流程. 所有实体都会更新.

在前 11 帧. 合计: `TimeActive` 增加 10 次. `IGT` 增加 11 次. 没有实体更新.

由于 `CanPause` 需要!`wasPaused`, 因此你无法连续暂停, 中间必须至少运动 1 帧.

由于 `NoControl` 期间也走 `TimeActive`, 因此每次 `Pause` 是会走 9f 的. 对于加载, 这基本不构成影响 (仍有可能触发部分 `Spinner Drift`), 对于卸载, 这有影响.

17.28.1 Spinner Stun

技巧: 部分情形下, 先故意远离以拉开镜头, 然后趁机提高速度 stun 回去, 会比直线路线更快. (对于从 0 速开始下落的情形比较显著)

17.28.2 重叠的刺

据说官图有不少地方有两个刺具有完全一样的位置. 这使得暂停调 spinner cycle 的恶心程度上升了.

17.29 Engine, Scene, Level, Session

Level 是指一个房间. (呃, 同一关下不同 Level 的 TimeActive 是怎么共享的? 这应该得看切换房间是怎么写的). Scene 大概就是 Level 的基类..? Session 大概是一整个关卡?

实体是在什么时候加载? 看起来像是 Level.LoadLevel() 的时候加入到 Scene 中的. (因此是一个房间一个房间加载的)

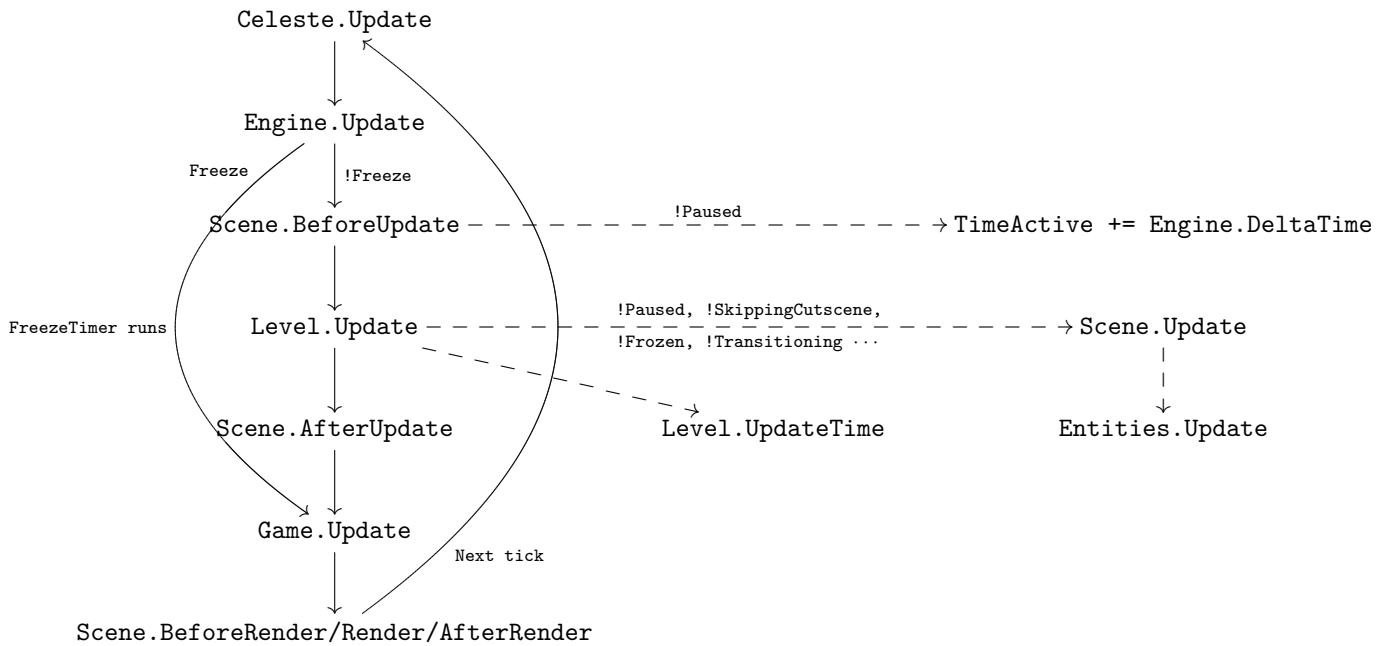
CelesteTAS 应该用 UnloadedRoomHitbox.cs 来显示未加载房间的. 用 GameInfo.cs 显示大部分信息.

使用的引擎 Monocle 是开发者自己写的, 包括蔚蓝在内的几个游戏都使用了它. Celeste 是 Monocle.Engine 类型的.

gameTime 是如何设置的: 在 Microsoft.Xna.Framework.Game 的构造函数 Game() 里, 有 TargetElapsedTime = TimeSpan.FromTicks(166667L). 游戏运行是 Game.Run(), 其中的主要成分是 Game.Tick(). Game.Tick() 会 gameTime.ElapsedGameTime = TargetElapsedTime 然后 Update(gameTime), 因此 RawDeltaTime = (float)gameTime.ElapsedGameTime.TotalSeconds = 0.0166667f. (Celeste.orig_ctor_Celeste() 中设置了 base.IsFixedTimeStep = true, 因此是固定帧率的这个分支). 在 Game.Tick() 当中关于 Update(gameTime) 的部分跑完了之后, 最后还有 Draw(gameTime), 完成渲染.

更新层次: 从 Celeste.Update() 开始. Celeste 是 Engine 类的, 并将 update() override 了. Celeste.Update() 里面主要是 Engine.Update(). Engine.Update() 里主要是要么冻结帧则除了冻结倒计时减一啥都不干, 要么就是 Scene.BeforeUpdate() + Scene.Update() + Scene.AfterUpdate(). 这里 Scene 为 Celeste.Level, 也 override 了. 大体上 Level.Update() 要么是处理暂停, 传送, 过场动画 (Cutscene) 等, 要么就是 Scene.Update(). Scene.Update() 在不暂停的情况下会顺利更新. Engine.Update() 做完还有 Game.Update().

层次: Celeste.Update() -> Engine.Update() (冻结) -不冻结-> Scene.BeforeUpdate() + Scene.Update() (-不暂停-> Level.Update()) + Scene.AfterUpdate() -> Game.Update().



实线：运行顺序，虚线：这个方法大致都做些什么。

(Level.Frozen 和 Engine 的 FreezeTimer 没啥关系...Level.Frozen 是吃心, 磁带的冻结, 草莓种子合成动画的 CS 等相关的.)

在 Scene.Update() 中, 有 RendererList.Update() (在 Entities.Update() 之后). 因此渲染确实是在更新之后.

根据上图, 冲刺的三帧冻结帧并不能增加 TimeActive. 因此蹲 SpinnerDrift 时冲刺没有用.

Level.UpdateTime() 也会在 Engine.Update() 的 DashAssistFreeze 分支中出现, 也就是说冲刺辅助是会增加游戏计时器时间的.

在 Pause menu 期间, TimeActive 不会变动而游戏计时器却在走, 所以一般来说 TAS 会 Pause 打开菜单后立马关掉.

在 Level.Render 中, 先进行 GameplayRenderer (里面是 Entities.Render, 然后再是 Entities.DebugRender, 因此深度高的 DebugRender 反而会在深度低的 Render 上面.), 然后依次是 Lighting, Background, Foreground, Hud, ScreenWipe, HiresSnow. 呃, 不过各个大项的深度并不是按照这个顺序排的, 可能跟 SetRenderTarget 有关?

此外, 冻结帧唯一还会更新 CassetteBlock.

Monocle.Entity 还有 PreUpdate 和 PostUpdate 两个字段, 不过原版中很少有用到, 似乎只有 CS07_Credits 有用到 PostUpdate, 而 PreUpdate 完全不见踪影.

顺带一提, 注意 DeltaTime 是在 Engine.Update 开头计算的, 这甚至在 Scene.BeforeUpdate 之前 (where TimeActive increases), 而之后 spinner 的 OnInterval 则用的也是这个 DeltaTime... 于是在这一帧发生的任何改变 TimeRate 的行为, 都实际上只能在下一帧才体现出来, 比如 SeekerEffectsController, BadelineBoost 等, 你可以在信息面板监视发现 TimeRate 和 DeltaTime 的这一帧延迟

17.30 地图

一张图的结构, 静态的看, 每一张图/关卡 (1A, 2A, etc) 是 MapData, 里面的每一面/房间是 LevelData. 而动态的看, 地图本身以及其上运行的音乐, 玩家触发的 Flags, 当前地图是否 FullClear 等信息构成了 Session, 一张图里的每一面及其中所有实体的状态构成了 Level.

Level 的基类是 Scene (场景), 里面存储着 Entities, TimeActive, Tracker 等信息. Session 里存储着 MapData, LevelData 等数据, 以及草莓, 钥匙, PlayerInventory, 磁带, 死亡数等信息. 而 LevelData 则存储着 Entities, Triggers 等的 EntityData, 房间名, WindPattern, 还有 Solids, BgTiles, FgTiles 等 string.

我们这里主要讨论”静态”的这一侧.

17.30.1 Mapdata

MapData 是怎样生成的: MapData.Load();

具体内容: 首先由 BinaryPacker.FromBinary(string path), 将指定文件路径上的二进制文件, 处理成 BinaryPacker.Element 类型的一个对象, 称为 root.

Element 类的成员有四个字段, string Package, string Name, Dictionary<string, object> Attributes, List<Element> Children. 这形成了一个树状的结构. 其中 Attributes 可能的类型有: bool, int, float, string. Package 字段只在初始的 FromBinary(string filename) 中会设置, 在后续的递归进行的 ReadElement(BinaryReader reader) 中并不设置. 因此这个字段只对 root element 有意义, 前面加上”Celeste/”后就等于 AreaData 的 SID (string ID).

第一层 (root) 的 Name 一般是 Map. 第二层的 Name 是”levels”, ”Filler”, ”Style”.

- levels 的每一个 child (第三层) 用于生成 LevelData levelData = new LevelData(child), 这些 levelData 构成 List<LevelData> Mapdata.levels;
- Filler 的每一个 child (第三层) 给出四个 int attributes (x, y, w, h), 构成一个矩形, 这形成了 List<Rectangle> MapData.Filler;
- Style 自身 (第二层) 给出 MapData.BackgroundColor (但不是必须要有). Style 如果有名为 Backgrounds 或 Foregrounds 的 Children (第三层), 那么 BinaryPacker.Element Mapdata.Background/Foreground 就由它们赋值;

在这些已经处理过的数据的基础上, 进一步得出 Mapdata 的其他字段.

以上是原版的情形. 如果是 mod 图, 在一开始还需要额外找到 root 的名为”meta”的 child, 用它来修改 AreaData, 并调用所有 EverestModule 提供的 EverestMapDataProcessor 来修改 root. 将处理的结果作为新的 root 再执行上面提到的操作.

<https://maddie480.ovh/celeste/map-tree-viewer> 展示了这树状的结构.

17.30.2 上面略过的一些细节

从 BinaryPacker.Element data 到 LevelData: Attributes 给出了 LevelData 里类型为 bool, int, float, string (及一些基于这些类型构成的简单类型) 的基础字段的值 (第三层). 而 data.Children 里面, 名为"solids", "bg", "fgtiles", "bgtiles", "objtiles" 的 child, 用其 Attributes 里唯一存着的 string 给 LevelData.Solids, Bg, FgTiles, BgTiles, ObjTiles 赋值 (第四层). data.Children 里名为"bgdecals", "fgdecals" 的 child (第四层), 它的每一个 child 直接生成一个 DecalData (第五层), 合起来得到 List<DecalData> LevelData.BgDecals/FgDecals. data.Children 里名为"entities", "triggers" 的 child, 它的大部分 child 各生成一个 EntityData (第五层), 合起来构成 LevelData.Entities/Triggers. 此外还有"entities" 的部分 child (第五层), 负责直接生成 LevelData 里的 List<Vector2> Spawns, int Strawberries, bool HasCheckpoint 等字段.

从 BinaryPacker.Element 到 EntityData: Attributes 给出 EntityData 的大部分字段 (第五层), 而 Children 给出 Vector2[] EntityData.Nodes (第六层).

从 Mapdata.Background/Foreground 到其他类型: 并不是立刻的, 而是后续在 LevelLoader.LoadingThread() 中进行. Mapdata.CreateBackdrops 将 data (第三层) 变成 List<Backdrop>. 原版情况下到第五层就全部完成了, 部分 mod 可能会需要更多层.

从 Solids, BgTiles, FgTiles 等到实际存在: 这些 string 是由矩阵形的 char 构成的, 它们会在 LevelLoader.LoadingThread() 被切割成 char 的二维数据, 形成 VirtualMap<char>, 再根据这些 virtualMap 去生成 level.SolidTiles, level.BgTiles 等. 对 SolidTiles 来说, char 的具体值表明了所在位置的砖块的样式.

从 EntityData 到实际存在: 在 Level.LoadLevel() 中作为实体构造函数的参数.

17.30.3 最初的一步: BinaryPacker 的工作方式

正如 System.IO.BinaryReader 可以 ReadString() (因为 string 有内存限制 2GB, 导致其长度不可能超过 int 的最大值 (实际上界比这小), 所以存的时候先存一个 int 表明长度就行, 实际运作中还用了 7-bit 整数压缩算法来节省空间), 读取到 Attributes, Children 等的时候, 会有固定格式的开头, 告诉你后面应该怎么读. 所以一步一步地读取即可. 作为约定的结果, Attributes 最多存 256 个 key-value pair. Children 最多存 32767 个.

17.30.4 AreaData

原版的一张图可以有 B-side, C-side, 它们当然是不同的地图, 但却又同属于一片区域, 这就是 Area. 并不存在 Area 这一类型, 但确实是有这样一个概念的. AreaData 主要负责处理在 3D 雪山这个场景中的事情, 包括这个 Area 的图标, ID, 名字, 切换到此 Area 时雪山摄像机应该处在的位置, 标题栏颜色, 内部的 JumpThru, Spike 等的样式, 关卡完成画面, 死亡重生时 ScreenWipe 的样式, 从雪山大地图进入关卡时的 ScreenWipe 的样式等.

17.31 进入房间的具体机制

各类进入一个房间的方式, 都归结于 Level.LoadLevel(Player.IntroTypes playerIntro, bool isFromLoader = false). LoadLevel 会

(1) 读取 Session 中的 LevelData.

- (2) 几乎最早地加入 WindController.
- (3) 用 levelData.Entities/Triggers (类型为 EntityData) 给场景加入实体/触发器.
- (4) 如果 playerIntro 不是 Transition, 除非已有非空的 level.NextLoadedPlayer 作为候选, 否则就会创建一个新的 Player, 并将其加入场景. 如果 playerIntro 是 Transition, 那就不会加入新的 Player.

Level.UnloadLevel() 则相当简单, 它会移除所有不带 Tags.Global 且不是 Textbox 类型的实体, 并调用 base.Entities.UpdateList(). 注意, Level 本身仍会照常更新. 因为 level (= Engine.scene) 并不会被设为空, 因此 level.Update() 依然会在 Engine.Update() 中被调用.

然后我们看 LoadLevel 在哪些地方以怎样的形式用到.

- (1) Level.Reload(), Level.TransitionRoutine(...), Level.TeleportTo(...).
- (2) 涉及场景跳转的 Cutscene.
- (3) PlayerDeadBody.DeathAction.
- (4) LevelLoader.StartLevel().

第一类的用法与第二类基本相似, 都是基本以 level.Unload(); ... level.Session.level = "XXX" (下一个房间的名字, 以此去 Session 里查找对应的 LevelData 等); ... level.LoadLevel(...); 的形式存在. (用到 LoadLevel 的 Cutscene 在 Level.Unload() 之前都会 level.Remove(player), 且对应的 IntroTypes 不是 Transition.

TeleportTo 如果参数是 Player.IntroTypes.Transition 则会 Remove(player), UnloadLevel(), Add(player), LoadLevel(Player.IntroTypes.Transition). 否则 Remove(player), UnloadLevel(), LoadLevel(introType).

Reload() 只做 UnloadLevel(), LoadLevel(Player.IntroTypes.Respawn), 但它本身只可能作为 PlayerDeadBody.DeathAction 而被调用.

TransitionRoutine 比较怪异, 它依次做这些事 (略去了大部分琐碎事情):

- (1) 生成一个实体列表 toRemove, 范围是所有不带标签 Tags.Persistent | Tags.Global 的实体. (玩家具有 Tags.Persistent)
- (2) player.CleanUpTriggers().
- (3) 指定下一个房间的名字.
- (4) LoadLevel(Player.IntroTypes.Transition).
- (5) 计算出切版目的地 playerTo.
- (6) 将玩家位置临时设置为 playerTo 检测与 WindTrigger 的碰撞 (这是少有的不在玩家更新中检测与 Trigger 碰撞!), 如果存在有不在 toRemove 列表中的 WindTrigger, 则将 windController 的模式设置成这个 Trigger 的, 否则照常设置模式. 将玩家送回原位. (此时实际上检测了这个房间与下一个房间, 一共两个房间的 WindTrigger?)
- (7) 40f 切板的那个循环. 详见 [第 17.32 小节](#).

- (8) `UnloadEntities(toRemove)`.
- (9) 如果玩家碰到 `RespawnTargetTrigger`, 将 `Vector2 to` 设为这个 `RespawnTargetTrigger` 的 `Target`, 否则设为玩家位置. 然后用 `LevelData.Spawns` 中最靠近 `to` 的作为 `Session.RespawnPoint`.
- (10) `player.OnTransition()`.
- (11) `NextTransitionDuration = 0.65f`. 这个参数即对应于切板共 40f. 除此之外只有 `AscendManager` 可以将其设置为 `0.05f` (对应于切板共 4f)(用于 7A 每小节的切板).

(因此在做 mod 时, hook `LoadLevel` 加入自定义实体的时候, 如果这个实体是静态的那么就给它加上 `Tags.Persistent` 或 `Tags.Global`. 否则在切板的时候刚在 `LoadLevel` 里将其 (或许重复地)(但由于 `Scene.Add(..)` 自带判断, 不会重复) 加入, 然后就又因为已经在 `toRemove` 中而迅速被移除了.)

`PlayerDeadBody.DeathAction` 会交给 `level.DoScreenWipe(..)` 调用. 如果 `DeathAction` 为空, 则会设为 `level.Reload`. 不为空有两种情形: 一种是 (5A 的那个) `PlayerSeeker.OnPlayer(..)` 中会产生 `PlayerDeadBody` 并将 `DeathAction` 设置为 `PlayerSeeker.End()`, 其内容主要是 `level.UnloadLevel()`, `level.Session.Level = "c-00"`, `level.LoadLevel(Player.Intro)` 并没有移除玩家 (也不需要). 另一种情形是 `Player.Die(..)` 中, 如果携带有金草莓则会将 `Engine.Scene` 设为一个模式为金草莓重开的 `LevelExit`, 后者又会在一段 `Routine` 之后设置 `Engine.Scene = levelLoader`. 无论如何, 都会调用到 `Player.Die()`, 会同时调用 `Scene.Add(playerDeadBody)` 与 `Scene.Remove(player)`.

`LevelLoader.StartLevel()` 就只是 `Level.LoadLevel(playerIntro, isFromLoader: true)`, 然后如果 `Engine.Scene` 是自身则 `Engine.Scene = Level`.

一些小例外: `FrostHelper` 有名为 `InstantWarp` 的 `Trigger`, 它的顺序是 `Remove(player)`, `UnloadLevel()`, `LoadLevel(Player.Intro)`, `Add(player)`. 不同于我们之前的情形.

注意: 上面的大部分情形, 实际上并不是生成了一个新的 `Level` 对象, 而是对旧有的 `Level`, 清除几乎所有的实体, 根据下一个房间的 `LevelData`, 再加入实体.

17.32 切板卡死

切板卡死的原理:

切板前只检查对应位置是否有关卡, 然后 `BeforeSide/Up/DownTransition()` (上切板会有冲刺 CD), 然后是 `Level.TransitionRoutine`. 这个 `Coroutine` 的主体是上面图里那个循环. 循环结束会触发 `Player.OnTransition()`, 这会恢复冲刺.

切板的时候会有个切板目的地, 如果你没法成功移动过去, 那么就卡死了. 切板目的地就是常规切板移动多少 px 后, 能够在下一个关卡的框内, 无视这些固体啥的. 切板的时候会决定一个方向 (上下左右), 然后沿这个方向移动. 呃, 除此之外, 上下切板还会各自多移动一些 (大概是 6px), 使得玩家不完全贴在版边. 应该是不够的会强制移动, 够的就不动吧. 对, 够 = 减去了上下切版额外修正的几 px 后, 仍在房间内.

是这样, 这个循环的条件一个是玩家切板, 一个是控制镜头的 `cameraAt`, 这个 `cameraAt` 的条件正常跑完是 40 帧, 然后一般情况下玩家切板也会早已跑完.... 但像那种情况就卡死了.

此外, `Everest` 对于非官图设置了防卡死.


```

}

public static void PopRandom(){
    Random = randomStack.Pop();
}

```

使得在这一个代码块中用 newSeed 对应的随机数序列, 结束后返回原先的随机数序列 Random.

少部分情形下, Celeste 会直接使用 new Random 来生成随机数, 比如 Celeste.Level.orig_LoadLevel 中 HiccupRandom = new Random(Session.Area.ID * 77 + (int)Session.Area.Mode * 999);

在 Celeste.Level.orig_LoadLevel 中, 大部分都在 Calc.PushRandom(Session.LevelData.LoadSeed) 与 PopRandom() 中, 其中 LoadSeed 由 Name 的每一位字符的 ASCII 码直接相加得到. 这使得地图初始化阶段的随机数被完全由地图名决定 (呃, 是这样吗, 那么改变一个圆刺的位置究竟会不会影响随机数?).

值得庆幸的是, 除去种种视觉听觉效果外, 游戏的运动是没有随机数的. 许多需要随机数的对象, 都是自带的. i.e. 自己有一个随机数字段并在 ctor 中初始化了, 且每次调用的时候要么 push + Monocle.Calc.xxx + pop, 要么直接对这个随机数操作而不调用 Monocle.Calc 里涉及 Random 的.

部分视听觉效果有自己的随机数, 剩余的共享一个随机数序列. 它就是 Monocle.Calc 类最开始就创建的 public static Random Random = new Random() = new Random(Environment.TickCount).

你可以通过 Set Calc.Random XX 来手动设置这个随机数序列. 这会使得你就连所有特效都一样, 100% 无 RNG.

一些意外地方的 RNG: Debris (比如 MoveBlock 碎掉之后的粒子), 它看起来是个人畜无害的视觉效果, 但是它是 Actor 类! 而诸如 MoveBlock, CrumblePlatform 等碎掉之后恢复的, 都是判定!(CollideCheck<Actor>() || CollideCheck<Solid>()), 而 RNG 会影响 Debris 的 lifeTimer, 进而间接影响到它们. 好在云, 车, CrumblePlatform 等判定启动用的都是 HasPlayerRider() 之类的, 总算大部分情况下不至于遇到 RNG.

此外, 车碎掉后在恢复前会关掉自己产生的 debris 的碰撞, 因此不至于被自己的碎片卡住... 但这不妨碍别的碎片来卡你.

CelesteTAS 在 v3.24.1 将 debris 的 RNG 改为取决于位置和房间名. 似乎没有避开原版.

```

private IEnumerator MoveBlock.Controller(){
    ...
    Position = startPosition;
    Visible = (Collidable = false);
    yield return 2.2f;
    foreach (Debris item in debris)
    {
        item.StopMoving();
    }
    while (CollideCheck<Actor>() || CollideCheck<Solid>())

```

```

{
    yield return null;
}

Collidable = true;

...
}

```

出生点 RNG: <https://discord.com/channels/403698615446536203/519281383164739594/1149061720384483328>

AscendManager 在玩家高度超过自身后开始运作 (因此引起剧情的并不是 BadelineBoost). 在 7A 中会唤起 CS07_Ascend. CS07_Ascend 自带一个 Coroutine SpinCharacters() 作为自身的 component (而不是玩家的)(半空中与 Badeline 绕大圈对视的 coroutine), 不过这随着跳过剧情也很快消失. AscendManager 自身则有在玩家位置的设置上用到 Calc.Random.ShakeVector(), 因此有 RNG. 而这里进入下一面后的位置正好在两个重生点的中间, 因此如果 RNG 抖动一下就会导致重生点切换. (此 desync 已被修复 (非官图中) <https://github.com/EverestAPI/CelesteTAS/commit/104e13c2d6fbfc0a895df1c1db02c14c5ed36b49>)

可以用 Set Calc.Random 1234 来固定这个随机数.

17.34 CelesteTAS, Celeste Studio 相关的运算顺序

为什么要 Console load 之后有个 1

Invoke 的执行时机?

Mouse 的执行时机

Studio 指令, 指的是写在 Studio 中的指令, 分两类执行时机: 解析时和运行时. 部分 Studio 指令如 Set, Invoke, 也有其对应的控制台版本.

解析时的运算顺序自然是不必管的. 运行时, 在 MInput.Update() 的末尾处, 因此在几乎所有其他更新之前.

17.35 你可能不知道的一些 TAS mod Feature

(1) ConsoleEnhancements: 在按 ~ 打开控制台的情况下, 对实体按左键可以查看这个实体的一些信息 (不需要按逗号!). 注意, 这和 InfoWatchEntity 并不是同一件事情, 至少它们各自展现出的东西是不一样的. 在最近 (08.01) 的一次更新中, 还将实体来自哪个 mod 的信息加入了 ConsoleEnhancements.

17.36 杂项中的杂项

倒也不是真的杂项, 但他们的内容大多还没怎么写或太短, 姑且归到一块儿

(1) 数据类型: 基本没有用 double 的, 都是 float, int, bool. 呃, actualdepth 是 double.

(2) 草莓: BlockField 会阻止收集草莓, 呃那草莓倒计时会走吗? 草莓的收集时长的机制.

(3) 3A 屋顶: 卡进去并 cb 是因为切版加载问题.

- (4) 冰墙: 无法抓取的冰墙 WallBooster 是通过 ClimbBlocker 实现的. ClimbBlocker 覆盖在墙体表面. 试图抓墙/抓跳时, 会先根据抓墙/抓跳的判定范围 (2px/3px) 以及朝向, 用 ClimbBlocker.Check 判定范围内是否有 ClimbBlocker. 如果是则无法抓墙/抓跳. ClimbBlocker 在原版的实例只有 WallBooster 和 InvisibleBarrier, 而 CelesteTAS 只修改了 WallBooster 的 DebugRender. 此外, 这里写的判定范围不完整, WallJumpCheck(), ClimbJump() 在 NormalUpdate() 前的检查, 它们判定的距离条件是不一样的. 在比较规整的类原版地形应该没问题. 但除此之外说不定可以构造出奇怪的反例以推翻直觉 (直觉: 如果水平方向上有被冰覆盖的墙面, 那么无论如何也别想抓跳).

ClimbBlocker 的一些具体细节, 冰墙 Edge: false 而隐形墙 Edge: true.

此外, 冰墙只有在是冰墙 (IceMode) 的时候才会 ClimbBlocker, 在它是 Conveyer (!IceMode) 的时候, 并不会.

由于进入 StClimb 之后想要维持在 StClimb, 基本上只需按抓 + 前面有墙 + 体力充足, 并不检测是否有冰墙, 因此在一些特例下, 可以抓在冰墙上. 例: 利用 MoveBlock 的碰撞箱复活非常早, 在此期间抓住 MoveBlock, 然后 48f 之后才是 MoveBlock 的视觉及其侧面的冰墙复活. 如此就可以做到抓在冰墙上. 并且还能被 MoveBlock 带动. (但自身无法爬上爬下).

- (5) StaticMover: 弹簧随移动块运动的效果就是它

- (6) Dashless water:

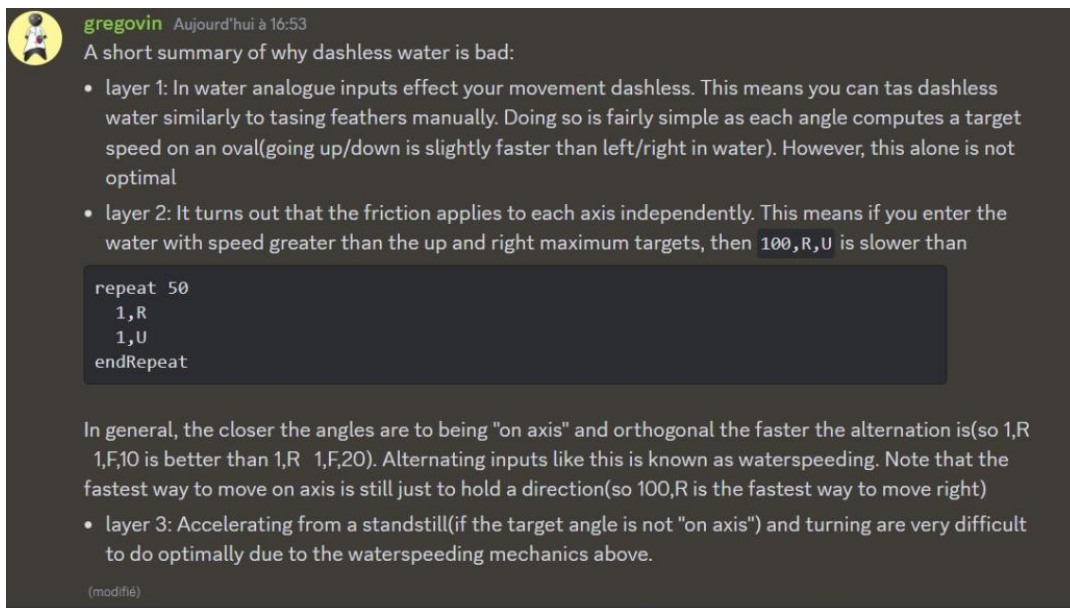


图 33: Dashless water movement

- (7) 撞击电箱/可撞碎块的弹起: 可以用狼跳打断, 速度没那么大. 前一帧 onGround, 下一帧撞击, 则可以恢复冲刺 (例子: 下冲电箱 / 2A 那个斜下冲 / 1A ARB 那个斜下冲). 但狼跳帧毕竟不是想获得就能获得的, 如果是侧面撞击的话需要比较好的地形.

- (8) 风: 蹲姿与风. 冲刺状态不吃风. 因此有时候大风的情况下, 尽可能少用单纯的冲刺赶路反而更快 (4c 第三面). ClimbHop 也会阻止风.

- (9) 吃心: StBoost 可以吗, 好像可以 (呃, 至少一些 mod 心可以). Theo 水晶也行, 那么到底哪些行为能吃心呢.
吃心 (带中间字幕的那种): 中间的字幕用 59, ; 1, O; 30, ; 来跳过
- (10) 输入与预输入 (buffer): 比如有时候我们需要按跳的半重力效应, 但不想踢墙跳. 这时候就需要尽早按下跳.
尽管更早的时候可能还没有半重力效应! Buffer 的具体机制: Input.cs
MInput 的更新层级相当高, 它在 Engine.Update(..) 最开始的地方.
此外, 冻结帧期间 buffer 也会流逝.
- (11) gliderboost: 这个机制让你上冲抓水母能飘很高
- (12) Player.idleTimer, sprite:
- (13) DashOnly: <https://discord.com/channels/403698615446536203/519281383164739594/1070995626617225336>,
there is a very niche case where this is bad and you would need movement only binds, if you want to dash
right while on top of a controllable moving block
- (14) MoveBlock: MoveBlock (车) 撞到墙壁会碎掉. MoveBlock Controller 的具体机制是... 移动速度是...
- (15) 由于先碰撞 (包括墙角修正), 再检测伤害, 因此对于向下有刺的墙角, 仍可以直接上冲接墙角修正.
- (16) 亚像素调整也不是总能成功的, 除了有时候就是差一点. 有可能由于速度一直是 20 的倍数, 时间恰好是 3
的倍数, 导致根本没有亚像素的事情, 或者其他类似情形. 类似的, 传送门子弹的位置调整也有这种离散性问
题, 速度实在太大了, 会在 N 或 N+1 帧到达且 N 很小, 导致 N+1/N 太大, 微调速度方向造成的变化无法
覆盖它 (呃, 我大概可以有一些更精确的计算说明此事).
- (17) 据说 Hitting a ceiling will end the jump early if it has been more than 3 frames since the jump started. 确
实有此事. 因此在较矮的通道里全靠跳跃加速时, 比起 1RJ, 另一种方案是按住跳直至头顶. 根据通道的具
体高度, 两种方案应该是各有优势区间?
- (18) 有的时候动画显示的朝向似乎和实际的 Facing 不一样, 比如 1A.tas lvl_6a 的第 19 帧.
- (19) Key cycle. 钥匙开门时间不是固定的. 应该是与 Key.UseRoutine 中 while (sprite.CurrentAnimationFrame
!= 4)yield return null; 有关.
- (20) 开带锁的门需要玩家中心与门中心的连线不与 Solid 碰撞.
- (21) key clone? 目前只存在于空想. 如果可行的话那么可以用来 skip 3A 开头的两个门, 噢可能更重要的是 huge
mess skip.
<https://discord.com/channels/403698615446536203/1066938051810558024/1068521777829007420>
<https://discord.com/channels/403698615446536203/1066938051810558024/1068522244298506322>
- (22) 撞击 Kevin, 第一次电箱 (或者 mod 实体 StationBlock 等) 时, 常见操作是下一帧用跳跃来取消 Rebound 的
(120f*dir, -120f) 速度并取消 AutoJump (如果在顶上的话还需要卡出狼跳时间再撞). 但是对于 DashBlock
或第二次电箱, 撞碎的下一帧它就消失了, 无法从它上面跳跃, 想要侧面撞击取消 Rebound 的话, 需要获得

狼跳时间, 然后撞完用狼跳, 这样的话水平速度会变成 $120 - 8.333 - 40 = 69.166$ (正向狼跳) 或 $120 - 4.333 + 40 = 155.666$ (反向狼跳).

- (23) 恢复冲刺检测的是是否与!CollideCheck<Spikes>(Position). 因此虽然向下的尖刺没有 LedgeBlocker, 它也会影响冲刺的恢复.
- (24) 在一些不太常见的情形, 在向右起飞的过程中, 右踢墙是可以不丢失速度的. 那就是先撞墙获得 retained, 且此时有超过 130 的 liftboost, 就可以右踢墙.
- (25) 不同于其他, 进入 StSwim 的判定是单独写在 Player.Update() 中的. 导致玩家无法从水面上, 在 StClimb 中向下爬进水面, 例如 6A 开头. 不过也是因为这里奇妙的判定, 在合适的条件下玩家可以从 StClimb 状态下直接瞬移到任意高的水面上. (比如两片水域, 两者高度差 8 px, 然后玩家抓在移动块上, 高度可以与下方水域碰撞但不与上方碰撞. 则移动块移至此处瞬间, 玩家移动至上方水域的水面)(或者单个水域, 果冻块放在水的一半高以上, 那么出果冻反抓也可瞬移)
- (26) 如果进红泡泡的时候 onGround (不仅仅只是有狼跳时间), 那么可以在红泡泡中一直保持蹲姿 (如果运动不含向下分量)
- (27) 无敌状态下碰到上升的岩浆海可以把岩浆海”击退”, 此外这也是能在 StClimb 下不依靠 wallbooster 或爬墙获得向上速度的方式..
- (28) 携带有草莓种子也会阻止 climbhop, 大概是为了避免草莓种子意外丢失, 很合理
- (29) 9A 心门 (以及其他初始隐藏的心门) 在最初实质上是关上的, Collidable = true 而 Visible = false, 你可以在远处扔一个水母发现此事. 然后在 Maddy 靠近时才将自身位置瞬移至打开, 再播放迅速关上的动画.
- (30) 水母会被挤压而死, 但它没有死亡动画.
- (31) Kevin 可以用来穿向上的单向刺板, 有不同方法 (假定 Maddy 紧贴 Kevin 底端, Kevin 下撞但未必满速):
 - A) Maddy 原先就处于蹲姿 (或在这一帧上 dd), 自身速度 (及其他发生在自身更新中的修正) 位移了 a px, Kevin 移动 b px, 原先在单向板上方 $4 \geq c > 0$ px (紧挨着算 0 px), 那么 $b > c > a$ 即可穿越单向板且这帧不死. 下一帧需要 $b - c + a' \geq 3$ 才可移动到刺下方, 这里 a' 是下一帧的自身速度位移 (由于前面未自己撞地, 不触发 OnCollideV, 因此竖直速度未重置为 0, 但亚像素确实会在 MoveVExact 里撞击发生时归零). 比如 $b = 4, c = 3, a = 2, a' = 2 \sim 3$ (自身速度 160 在前一帧取到移动 2px), 或者 $b = 4, c = 1, a = a' = 0$ (上 dd).
 - B) 原先处于站姿, $b = c > a$ 恰好将 Maddy 推至单向板 (仍是站姿), 然后下一帧 Maddy 自身主动蹲跳, 上升不超过 $2px \leq$ 蹲姿腾出的 $5px$, 不死, Kevin 需要移动 4 px, 再下一帧回到情形 A) 中 $c = 1, b = 4, a = 0$ 即可. 于是可行的组合是 $(b, c, b') = (4, 4, 4)$ 或 $(3, 3, 4)$, 后者需要 Kevin 在加速.
 - C) 原先处于站姿, $b > c > a$, 必有 $b \leq 4$ 所以 Maddy 被压至蹲在单向板上 (亚像素也重置) 后离 Kevin 块至少 2px, 下一帧 (蹲) 跳, 不死, Kevin 移动 b' 使得 Maddy 现在的 $c'' = c - (b + b' - 5)$, 需要 $c'' = 1$ 于是 $c = 3, b = 4, b' = 3$, 即可. 这需要 Kevin 本身在减速, 并不是常见的情形
- (32) 6A boss 战中的那些掉落块同样是 FallingBlock, 只不过它与一般的相比, 触发方式比较特殊, 比如无法从旁

边抓落。FallingBlock 有一些字段, 比如 Triggered, climbFall, finalBoss, 如果是 finalBoss 那么其流程就被完全固定。否则, 如果分为两种: 主动式的检验 PlayerFallCheck(), 被动式的 Triggered。前者就是 Player 在顶上或旁边 (如果 climbFall = true), 后者就是 FinalBoss (6A boss 战) 或者 Kevin 块等会触发。

- (33) FloatySpaceBlock 的 sineWave 并不是组件, 而仅仅是 float, 这导致通过浮点数加法精度限制, 挂机足够久之后 FloatySpaceBlock 没有自发的漂浮行为。
- (34) 单向 transition 实际上都是靠 InvisibleBarrier 实现的。
- (35) 原版 HeartGem 在收集期间会在关卡的左, 右, 上方临时增加 InvisibleBarrier。而像 SummitGem 就没有 InvisibleBarrier。CollabUtils2 里的 MiniHeart, 其流程与 HeartGem 基本相似, 但是没有 InvisibleBarrier。
- (36) Farewell 9-9 里有一些额外的重生点, 但是它们没有对应的 ChangeRespawnTrigger。疑似测图残留。
- (37) 意义不明的代码: SuperJump 里先把 gliderBoostTimer = 0f, 然后 = 0.55f... 我很确信两者之间的代码并没有什么诡异的副作用需要我们这样做...
- (38) Puffer 的 idleSine 的初始值是随机生成的, 导致它位置初始可能 offgrid。这是我们知道的。但此外还有一件事, idleSine 可以发生重置! 在 States.Hit 或 States.Gone 结束回到 States.Idle 时, 会重置 idleSine.Counter = 0f。因此 Puffer 的表现可能会不一样! 如图 34, 在初始状态下, Puffer 浮动到最高处时, PlayerCollider 中不可能触发 ExplodeLaunch。但是复活之后, 就可能触发 ExplodeLaunch。(startPosition 被 idleSine 影响过, 这是没法改动的。)



图 34: 重置 Puffer 的 idleSine

- (39) 据说顺着 SwapBlock 运动方向 dd 再按抓也可以蹲姿 StClimb
- (40) 弹簧有个字段 bool playerCanUse, 字面意思, 如果为 false 则它不对玩家生效。例如 Farewell 的 Spring[e-04:928]。不过它对 Holdable, Puffer 等不起作用。此外, Puffer 撞击弹簧同样需要像玩家一样判定速度, 只不过判定的值是 Speed.Y \geq 0 (若弹簧向上), Speed.X \leq 60 (若弹簧向右), Speed.X \geq -60 (若弹簧向左)。而原版 Holdable 类同样需要判定速度, 判定的值都以 0 为界。
- (41) Holdable 如果被拾取, 那么它的更新里就不再运动, 而是交给玩家 UpdateCarry 以及其他移动块推动等给出运动。然而, 玩家对 Holdable 大体上保证 PickUp-Drop 成对出现, 而 PickUp 里并不保证如果原先有 Holding, 将其 Drop。并且, UpdateCarry 是只更新玩家手拿的那个 Holding。因此, 如果先 PickUp

Holdable1, 再 PickUp Holdable2, 这时就有 Holdable1.IsHeld, 同时 Holding = Holdable2. 因此 UpdateCarry 只移动 Holdable2, 而 Holdable1 却因为 IsHeld 而不移动.

原版不太会出现这种问题, 不过一些不谨慎的 mod 可能会导致此问题.

<https://github.com/CommunalHelper/CommunalHelper/pull/173>

- (42) 基于 StPickUp 会速度归零并保留 varJumpTimer, 你可以在原版里做出一个两段跳. 见 2024.01.07 对 5S 的改进 (当然, 这个改进并不仅仅只是这一点...). 唔, 同样基于 StPickUp 也有一个速度还原机制, 可以在特定情形下发挥出 Retention 的效果, 不过由于我们没法移动, 只能指望移动块自己移走了... 好消息是 StPickUp 的持续时间可比墙面 Retention 长多了. 有 Theo 水晶在就无法切板的特性也可以用来多做一个 ultra (i.e. 从较高的地方飞至出口, 那么抛 Theo, 15 RDX, 跳起来接 Theo 切板). 呃, 这么说来, 我们应该也可以通过墙面 Retention 使得 StPickUp 下获得速度?
- (43) 我们知道 ExplodeLaunch 实际上不会杀掉 DashAttacking (而 Jump 之类的会)(也因此有了利用 Bumper 重定向进果冻的操作). 我们知道若冲刺方向水平 + DashAttacking + 脚下 3px 有地面/单向板, 就会修正到地面/单向板上. 这两者相结合, 横冲碰上 ExplodeLaunch 获得竖直速度, 就会见到每帧向下 7.5px/s 位移. 这是因为时序: 先修正到地面上, 再向上移动. 由于每次的向上速度都比之前少 7.5 (LaunchUpdate 导致的), 所以就呈现出向下 7.5px/s 位移速度的表象.

18 挑战/异变与 Mod

这部分旨在讨论较偏离主线游戏的机制.

18.1 超冲

1.44 原理.

Ultra hop: OnCollideV 1.2 倍修正的条件是 DashDir.Y > 0f, 但 Super/Hyper 的条件是 Abs(DashDir.Y) < 0.1f.

因此可以

5	R, Z
1	R, D, J
8	R
1	R, J

获得 378 的速度.

由于超冲拐弯优先于 Hyper, 因此 R,D 和 J 可以写在同一帧.

超冲在 5b 试图无限加速的时候, 可能会卡进地面

原因应该是, 的确无限加速了, 但只要下一面并不是一面墙壁, 而是有平移后再下落的空间, 就有可能.

因为无限加速是依赖于无限斜下冲. 但速度极高的情况下, 这一次冲刺会先水平移动到有下落空间的地方, 再竖直移动 (没撞地呢! 自然是有竖直速度的). 然后在无法穿越房间的情况下应该会试图把你拉回去, 但这大概只改变水平位置没改竖直位置. 所以就卡地里了. 我猜如此

这可能是 bound 相关的函数实现的效果。

超冲打断超冲虽然很遗憾的, 不能像 StNormal 和 StClimb 一样进入冲刺吃到 LiftBoost, 但是它也没有 Normal-End/ClimbEnd 的 wallSpeedRetentionTimer = 0f (DashEnd 里没有). 属实是塞翁失马了。

18.2 部分挑战/异变 (超冲, 无抓, 打嗝等) 相关

如何最快打开超冲 (不用 Console)

18.3 Ultra Jump 异变

ExtendedVariantMode 的 EveryJumpIsUltra 异变. 大体来说, 是所有不会导致横向速度重置的跳跃动作, 额外加上 Ultra. 具体来说, 在 Jump, ClimbJump 的调用末尾, 加上

```
DashDir.X = Math.Sign(DashDir.X);  
DashDir.Y = 0f;  
// Speed.Y = 0f; // this seems a bit counter-productive when applied after a jump...  
Speed.X *= GetVariantValue<float>(Variant.UltraSpeedMultiplier);  
Ducking = true;
```

但是由于 ClimbJump 的内部实际上是调用了 Jump 的, 导致实际上会有 1.44 倍速度加成. 在早先的版本中, 这被视为 bug, 不过现在已经视为特性了。

Ultra Jump 带来的一些新特性:

- (1) 依赖抓跳的 delayed ultra 不再可行. 不过你至少也无论如何能吃到落地后其他的 1.2 倍加速, 如果抓跳位置合适的话, 则是一共 1.728 倍加速.
- (2) 狼跳, 抓跳, 水面跳跃自带蹲姿.
- (3) 基于上一条, Ceiling pop 大多数情况下不再必要, 被抓跳自带蹲姿给代替. 同时抓跳逆天的 1.44 倍加速, 使得很多“掠角飞行”路线得以成立.

18.4 部分 mod 实体的特性

绿弹簧

铁弹簧

SidewayJumpThru

楼梯

18.5 Portaline

以下基于 1.0 版本. 新版本可能有变动.

传送门发生传送有两个时间点, 一个是 `OnCollideH`, `OnCollideV` 里面检测 (但注意传送门本身不是 `Solid!` 也就是说它起到了拦截其他 `OnCollideH/V` 的作用), 另一个是在 `PortalEntity.Update()` 当中. 因此涉及到切版时, 如果用 `OnCollideH/V` 触发传送, 可以快一帧. 同理, 由于发射传送门子弹也是在 `Player.Update()` 的末尾, 因此用 `OnCollideH/V` 触发传送即可在这一帧就在传送目的地发射子弹. 还是同理, 可以做到在带刺的墙壁上开传送门并进入不被扎死.

在传送门子弹碰撞的时候, 会创建一个 `PortalEntity`, 将其加入 `Scene`, 并将 `Instance.blue/orangePortal` 替换为它. 而在传送时, 检测的是对应的 `Instance.blue/orangePortal` 是否为空. 我们知道 `Scene.Add(Entity entity)` 实际上是将实体加入一个暂存池中, 等到下一帧调用 `Scene.BeforeUpdate` (或者其他 `Entities.UpdateLists()`, 这个时间点基本都是下一帧) 时才真正将其加入 `Entities`. 此外注意到子弹的 `Depth = 100`, 而传送门 `Depth = 0`. 因此在这一帧, 假如你预先已经建立好了蓝色传送门 (因此已经在 `Scene.Entities` 中, 会 `update`), 且这一帧橙色传送门子弹正好碰撞, 在建立橙色传送门, 且你在蓝色传送门处, 那么你就可以传送, 即使这一帧橙色传送门并不渲染.

18.6 Ceiling Ultra

见 [Ceiling Ultra 的 README](#)

19 还未处理的一些原始材料

19.1 Cassette Block

<https://discord.com/channels/403698615446536203/519281383164739594/1063844137440399492>

<https://discord.com/channels/403698615446536203/519281383164739594/1063871823948812419>

<https://discord.com/channels/403698615446536203/519281383164739594/1063880270262046761>

<https://discord.com/channels/403698615446536203/519281383164739594/1063886704706080840>

19.2 XNA/FNA desync

等到我回头写到这里, 估计 Everest 团队已经成功转移到.NET 了, .NET 成为事实标准, 这个 desync 也因此不复存在. 于是到时候这就可以一笔带过啦.

<https://discord.com/channels/403698615446536203/754495709872521287/1088432710806163567>

<https://discord.com/channels/403698615446536203/1068379326589976596/1088490611918766192>

<https://discord.com/channels/403698615446536203/754495709872521287/1088401589468938260>

<https://discord.com/channels/403698615446536203/407956898713960450/1074833187013931019>

<https://github.com/XMinty7/xminty7.github.io/blob/main/documents/bubblefaq.md>

MoveToX 用的是 `MoveH(toX - ExactPosition.X, onCollide)`, 由于浮点误差, 会导致最终位置离 `toX` 偏差一点. 而 XNA/FNA 对此的处理略有差异, 导致 desync.

19.3 一些申必应用

由于是先 `DashUpdate` 再 `DashCoroutine`, 特别的在第 5 帧, `DashUpdate` 时有 `DashDir = 0`, 因此无论之后是什么冲刺方向, 这一帧都可以吸附 (向上修正) 到单向板上. 例: 6B [a-06] 末尾.

<https://discord.com/channels/403698615446536203/519281383164739594/1081193572155535420>

retention, 然后用弹簧来跨过墙 (本来爬的话应该来不及), 从而达到了很高的水平 & 纵向速度.

<https://discord.com/channels/403698615446536203/519281383164739594/1084872686041301093>

<https://discord.com/channels/403698615446536203/854300208058073098/1104460710081216654>

<https://discord.com/channels/403698615446536203/617809769322774533/1007808556092895292>

<https://discord.com/channels/403698615446536203/519281383164739594/1084920691998085200>

<https://discord.com/channels/403698615446536203/519281383164739594/1085131576108658749>

<https://discord.com/channels/403698615446536203/754495709872521287/1085011460704436344>

<https://discord.com/channels/403698615446536203/519281383164739594/1085696979671203970>

<https://discord.com/channels/403698615446536203/519281383164739594/1021642871885013012>

<https://www.youtube.com/watch?v=3R-d4hDuRJQ>, file timer overflow.

<https://discord.com/channels/403698615446536203/519281383164739594/1088678079116873801>

<https://discord.com/channels/403698615446536203/519281383164739594/1088590093989716038>

<https://discord.com/channels/403698615446536203/519281383164739594/1088588732598321234>

<https://discord.com/channels/403698615446536203/519281383164739594/1089992665048821831>

<https://discord.com/channels/510279444368457730/510280159115608066/1060087080144875560>, 通过进入 Engine 减速区域再自杀, 来扰乱 TimeActive, 这可能会使你获得常规流程中不能得到的 TimeActive, 进而达成一些奇妙的 stun?

Nanashi: 这里有个东西不知道能不能用, 就是你调一个 6 帧兔子的亚像素, 然后 hyper 的第一帧抓水晶, 这样抓完水晶有个狼跳能再跳一次, 在 dash trails 里学到的, 就是你在抓取过程当中, 和平台有接触, 即便你原本是向上速度, 抓取结束后也能获得狼跳

<https://discord.com/channels/403698615446536203/1074148152317321316/1093693746949927057>

<https://discord.com/channels/403698615446536203/519281383164739594/1107419071613505536>

<https://discord.com/channels/403698615446536203/598977992957624343/1093996681692991528>, <https://discord.com/channels/403698615446536203/519281383164739594/1097290699621281885>, <https://discord.com/channels/403698615446536203/519281383164739594/1116088511305613504>.

<https://discord.com/channels/403698615446536203/519281383164739594/1116088511305613504>. 原版任意大

速度. 概要: 这样的装置能让你在与 Kevin 同向期间无限叠速度, 而反向期间是吃阻力, 一正一反合计是增加了速度的; Kevin 块仅在撞击方向与当前垂直时会增加 returnStack, 于是首先堆好前两个 Kevin 的栈, 然后再堆第三个 Kevin 的栈, 期间穿插一些竖直方向的撞击来调整时间, 使得他们同步; 最终堆好三个 Kevin 的 n 趟往返, 就可以叠 n 趟速度.

呃, 不完全, 首先叠 +40 会受到浮点数加法精度的限制. 其次, 速度乘 DeltaTime 一旦超过 int32 上限, 会导致速度朝右而运动朝左.

目前来看, 要叠到 float infinite 只能利用 * 1.2, 原版可以利用超冲卡墙里来保证运动朝左也没事... 包括卡 3a 墙里和 5a 墙里.

<https://discord.com/channels/403698615446536203/519281383164739594/1222710533489295441>

利用 theo 反复触发月亮块上互相面对的一对弹簧, 实现反复触发 retain, 以达成无限速度

Portaline 传送门无限速度, 呃好像很难克服这个速度朝右而运动朝左的问题. 一旦 OnCollideH 被调用, 那么 OnCollideV 的时候水平速度都是 0, 没法 * 1.2. 而平 u 又要面临如何取消这个冲刺的问题... 喔带个水母就行

<https://discord.com/channels/403698615446536203/519281383164739594/1016106250947285083>, <https://discord.com/channels/403698615446536203/519281383164739594/1016106250947285083>,
复制水母

由于 TrackSpinner/RotateSpinner 的位置未必是 onGrid, 导致它们的碰撞箱看起来不再是 C4 对称的.

<https://discord.com/channels/403698615446536203/754495709872521287/1095360275118563431>

<https://discord.com/channels/403698615446536203/519281383164739594/1106241731072114732>

<https://discord.com/channels/403698615446536203/519281383164739594/1107021482716037120>

<https://discord.com/channels/403698615446536203/1074148268407275520/1222248533021687819> 利用 moonblock recoil 来获得 86 的 liftboost, 而不是只有常规的冲刺 47, 不过好像说这个 liftboost 只能有 1f? 之后就会回归常规的 liftboost

没有来源, 自己看代码意识到的: FireBall 在冰/火状态切换下, 其运行速度也会不一样. 因此理论上可以设置好一个很极端的情形, 只有一堆火球的浮点数都凑对了, 才能过去? 由于 FireBall 内部参数的周期性, 把速度设置良好的情况下应该无法通过累计浮点误差来搞, 这样就只能靠切冰火状态来调整了.

此外, 冰球的伤害箱是弹跳碰撞箱之下的部分 (除非冰球坐标刚好是整数, 那时会再低 1px), 其上的部分碰到本体的红色碰撞箱只会无事发生, 除非碰到了粉色的弹跳碰撞箱.

利用 delayed ultra 可以实现碰狭窄平台的向上弹簧并处于蹲姿

撞 kevin 也可以实现 delayed ultra

<https://discord.com/channels/403698615446536203/754495709872521287/1135491514584272957>, 利用下风踩云但不触发云

<https://discord.com/channels/1089518756113428562/1089518756113428566/1094972674985447505>

<https://discord.com/channels/403698615446536203/519281383164739594/989441950006513664>, Jump timer abuse 之利用 wallbounce cancel 来实现比正常情况下更高的高度.

<https://discord.com/channels/403698615446536203/519281383164739594/1157881020486529074>, 总之就是撞天花板可以让你的 VarJumpTimer 不消失的情况下速度不朝上, 从而能进入 StClimb, 在 StClimb 下具有 VarJumpTimer. 而 StClimb 下 VarJump 机制不起作用, 于是你可以向下爬. 然后趁 VarJumpTimer 没结束再次回到 StNormal, 重新获得向上速度. 意义不明, 可能要说的话它用一次体力实现了两次向上抬升的效果? 假设我们跟着一个向右的竖直移动块, 在水平狭长通道里, 顶部是砖而底部是刺, 同时还有 8 组一上一下交替摆放的圆刺, 这样我们就不得不用这个技巧来实现了?

<https://m.youtube.com/watch?v=6X76NKwZhTM&start=1370>

holdable gultra / du, theo transition ultra, 见 2024.02.08 的 5S 改进.

在 CassetteBlock 的同水平位置有墙壁, 这时抓着墙壁且挨着 CassetteBlock (即能始终保持 StClimb 且紧邻 CassetteBlock), 那么每一轮 CassetteBlock 的上升下降会给出 -1px 再 +2px, 合计向下移动 1px 的效果 (至少 4A 磁带旁边那个是这样). 具体的机制我还没分析.

19.4 游戏崩溃与异常

https://cdn.discordapp.com/attachments/615402712011636777/834330998954393610/2021-04-21_03-12-05.mp4

<https://discord.com/channels/403698615446536203/754495709872521287/1095392387121025085>

<https://discord.com/channels/403698615446536203/754495709872521287/1095463256157601932>

<https://discord.com/channels/403698615446536203/754495709872521287/1071015717761208360>

在有剧情的情况下切板, 然后切板后跳过剧情. 很容易引起空引用异常. ³⁰

Seeker duplication ³¹

20 Code mod 相关

DetourContext 如何工作 (对于旧版本如此. 新版本的初始化可能不太一样): DetourSorter<T>.Sort(..), 大致有这些规则来排序: 如果一方的 Before 或 After 里有 *, 而另一方相应的位置没有, 则有明确的排序. 否则, 看 Before 或 After 中是否含有另一方的 ID (这一步不反向检验, 因此可能出现顺序问题). 否则, 按 Priority 的大小. 否则, 按 MonoMod.RuntimeDetour 内部给出的 GlobalIndex 来排序. ID 如果不指定的话, 那默认会是这个 detour 所在 Assembly 的 Name (这也为空的话还会有后备选项).

On/Il.Celeste.Blah.Blah hook 应该在 ILHook 更外面一层.

Everest.Boot() 的相关代码里有提到这些 hook 的层级问题.

21 习题集

允许查阅任何资料 (包括打开蔚蓝), 知道答案在哪里即可.

1. 移动块圆刺上能否恢复体力? 能否充能? 论述理由. 能, 能. 恢复体力的条件是 onGround. 充能的条件大体上是脚下是合适的地面 (固体 / 位置正确的 JumpThru (NegaBlock 不计入考核)), 且碰撞箱不与尖刺重合. 再加上运算顺序的原因, onGround -> 恢复体力 -> 充能 -> 移动 -> 判定死亡. 所以可以恢复体力/充能.
2. 移动块尖刺上能否恢复体力? 能否充能? 论述理由. 能, 不能. 同上. 注意游戏只对尖刺做了单独处理, 而非用了一类 component 来完成此事.
3. 如何理解实际碰撞箱?
4. 高速飞行的玩家能否穿墙?
5. 玩家在第几帧进行了什么 State 的更新?

³⁰crash_CSstrawberry.tas

³¹seeker_duplication.tas

6. 说明玩家的状态机调用 Begin, End, Update, Coroutine update 的时机.
 7. Ultra 机制.
 8. liftboost 从头到尾机制.
 9. 能否叠加 liftboost?
 10. 解释风与玩家更新顺序的案例.
 11. 为什么有时切板 43 帧.
 12. 解释利用 bino storage 切板 ultra.
 13. 解释 Spinner cycle.
 14. 解释 Spinner drift.
 15. 解释 CassetteBoost.
 16. 解释 Koral clip.
 17. 高速飞行的玩家为什么可能抓不到水母, 低速为什么一定可以.
 18. 解释 Ceiling pop.
 19. 解释 delayed ultra.
 20. minHoldTimer 期间可以丢掉水母吗?
 21. 解释岩浆块大跳.
 22. 解释春月加速.
 23. 解释偷体力.
 24. 解释 DashBounce.
 25. 解释为什么在移动块边上按对应方向键会加速下落. (表明有此机制即可)
 26. 解释 Hitbox = FeatherHurtbox
 27. 如何蹲姿 StClimb
 28. 试分析 <https://discord.com/channels/403698615446536203/519281383164739594/1175644079447609374> (的上下文中的 TAS) 中发生了什么. (话说如果在其中插入个 wallboost 会怎样)
- <https://www.bilibili.com/video/BV1uw411Y7JG/>

22 其他常用工具

ILSpy: 在一个东西上 Ctrl+R 或者右键分析, 可以看它在哪些地方被用到了. 上方的搜索按钮/Ctrl+Shift+F/Ctrl+E, 则可以让你快速找到某样东西.

CelesteTAS: custom info ³²

<https://discord.com/channels/403698615446536203/519281383164739594/1081832259260252301>

如何起到 custom info do math 的效果 (update: 现在已经被 lua in custom info 解决了)

³²[CelesteTAS-EverestInterop/Source/EverestInterop/InfoHUD/InfoCustom.cs](https://github.com/CelesteTAS-EverestInterop/Source/EverestInterop/InfoHUD/InfoCustom.cs)

Part II

实际应用

1 基础

1.1 干净利落地爬上平台

1.2 咖啡跳

过于基础, 本质上就是个踢墙跳, 最多偶尔利用机制不被刺杀死.

1.3 dd 穿刺

不能连续 dd.

1.4 Coyote Hyper

狼跳帧可以输入跳.

原理上包括 core hyper, dream hyper, 但一般用这个词指这些情形以外的 coyote hyper.

Coyote hyper = hyperdash, but start the dash in coyote time

E.g. dream hyper is a kind of coyote hyper

E.g. 走下方块边缘后做一个 (reverse) coyote hyper, 可以使得你恰好能在这个方块处额外做一个 cb

E.g. 在方块边缘处做正向 coyote hyper, 使得你可以更快落地, 以至于能接上一个 bunnyhop. 并且由于落地更快, 相比反 hyper 起手的兔子跳受到的空气阻力更少, 因此末速度更快. 相比单纯 hyper 不仅因为是兔子跳而跳得更高, 也因为跳跃 +40 而水平速度更大. (例: 春游图 donker19-intermediate ARB lvl_3 这里这样做才勉强能够到右边)

E.g. 4f bhop/5f bhop

1.5 gu / gultra / grounded ultra

地上的最快起手是兔子然后无限 gu

具体来说, 是

```

        6 | R,Z
        1 | R,J
        7 | R
        1 | R,J
    repeat n
        14 | R,D,X
        1 | R, J
    endrepeat

```

只要 $n \geq 1$ (跑完完整一个 14,R,D,X), 这样就是最快的. 否则可能需要换成 4,R,Z 之类的.

6f 兔子会比 325 快一点点, 从而产生比 390 更快的速度 (398.8)

1.6 DD 咖啡

DD 结束时解除蹲姿.

1.7 移动块多 cb

移动块包括月亮块, 车, Kevin, 岩浆块等.

人们很早就知道可以用前三者, 却很长时间都没想到用岩浆块.

移动块多 cb 的区间范围? (i.e. 在移动块左侧的话, 多少速度能多少 cb (还要考虑移动块本身移动速度和 Lift-Boost), 在右边又如何...)

如果是镜像 L 字型的车, 且并不是太矮, 那么你能无限叠加 cb.

1.8 rcb

1.9 水母 u, cutscene u

1.10 du

技巧: 为了减少风阻, 需要尽快落地. 因此需要控高使得能 retention 的同时, 剩余的纵向速度尽量小. 因此 cb 附近跳跃的帧数需要视情况调整.

1.11 落地同时保留向下的速度 / 下降蹲姿保留

恢复冲刺/体力且保持了向下速度. 可以用于长下落段, 也可以用于竖直高度限制比较苛刻的水平跳跃.

2 进阶

2.1 亚像素调整

v-t 图来理解亚像素调整.

亚像素调整不需要在” 最后一段”, 可以延伸到前面任何没有重置 (这个方向上的) 亚像素的时间段上进行调整

<https://www.youtube.com/watch?v=dD144drav5w>

240 速度移动 $240 * 0.0166667 = 4.0000080$, 比 4 多一点点!

官图绝妙案例: 6b d-03 到 d-04 的奇妙舞蹈

2.2 半体力 / 无体力

更快半体力.

假设面朝右对着墙, 并已抓在墙上.

半体力:

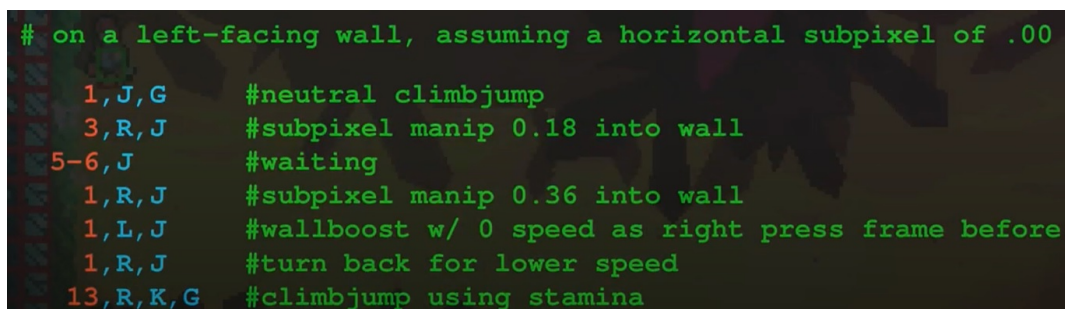
```
11 | K,G # neutral climbjump
   | 1 L,K # wallboost
   | 14 R,J,G # inward climbjump, costs stamina
```

超级无体力:

```
11 | K,G # neutral climbjump
   | 1 L,K # wallboost
   | 1 J # neutral walljump
   | 24 R,J # return to the wall
```

因为都需要 wallboost, 所以在冰墙上没法用

更快的半体力:



```
# on a left-facing wall, assuming a horizontal subpixel of .00
1,J,G #neutral climbjump
3,R,J #subpixel manip 0.18 into wall
5-6,J #waiting
1,R,J #subpixel manip 0.36 into wall
1,L,J #wallboost w/ 0 speed as right press frame before
1,R,J #turn back for lower speed
13,R,K,G #climbjump using stamina
```

图 35: 更快的半体力上墙

更快的半体力确实比半体力更快. 不过做完同样次数之后, 更快半体力上升的高度会稍微少一点 (因为它砍掉了上升较慢的部分). 假如有极端情况, 有一面高墙, 使得 X 个半体力 + Y 个超级无体力 + 收尾可以上去, 但 X 个更快半体力 + Y 个超级无体力 + 收尾上不去. 那或许可以考虑用一用常规的半体力? 应该不至于碰上这么倒霉的情况吧…

2.3 Cpop

复刻下泡芙佬的科学 cp.

2.4 踩刺

种种利用伤害箱和位置不统一的手法.

由于运算顺序原因, 在检测恢复冲刺的时刻看来, 刺与移动块是完全统一的, 因此你无法从覆盖满尖刺的移动块上恢复冲刺. 但恢复体力是可行的.

2.5 正向速度爬刺墙

2.6 大风爬刺墙

2.7 Corner glide

斜上冲 corner glide 的高度要求

2.8 Overclock

情形一: 中性抓跳启动 12f 的 wallboosttimer, 在这段期间恢复体力 (落地/切版/水晶等), 然后 wallboost + 27.5 (如果是通过落地等当然还需要离开地面, 避免获得的额外体力被刷新成 110.)

落地恢复体力的 overclock, 由于 OnGround, 110, +27.5 三者的顺序, 需要前一帧 OnGround 并在这一帧离开地面, 后一帧触发 wallboost.

情形二: 从地面上中性抓跳, 然后 wallboost. 在地面上抓跳不耗体力. (因此如果有落地无法恢复体力的拓展异变 (也就不会被向下重置到 110), 就可以无限刷体力!)

2.9 Wallboost 急刹车

2.10 Wallboost 白嫖距离

水平位移占主导因素的情况下, 基本都要用 wallboost 代替 walljump.

2.11 grounded wallboost / delayed wallboost

就是 $130 + 40$.

案例: 1A 无冲 [lvl_11]

2.12 idu

instant 指的是向下运动的第一帧就碰地.

实战案例: 9.tas 的 lvl_j-14b (2022/11/18, 9:09.763(32339) 版)

注意这里通过暂停预输入 J 来减半重力却不触发蹬墙的神奇技巧

关键点: 在速度 -15 的时候输入跳获得速度 -7.5, 下一帧速度 7.5 撞地.

模板: 起手纵向亚像素 0.5, 车速度 60.

14	L,D,X
1	R,J
10	R,D,X
2	R,J,G
1	R
1	R,K,G
1	R,J,G
1	R,K,G
6	R
1	S,N
10	J
1	R,J
1	R

2.13 moon spring boost

2.14 gultra retention

众所周知, 冲刺墙角 cb 的实质是 Retention, 再加一个有用但不必须的隔空 cb+40. 通过撞墙, Maddy 提前结束了 StDash 进入了 StNormal, 同时水平速度存储在 Retention 里, 然后通过抓跳跃过墙壁, 从而恢复速度. 就避免了 StDash 完整结束时的速度丢失.

特别的, 你可以做一个 gultra retention, 就是平 u 接 Retention, 从而保留住 390 的速度...听起来是这样, 但是问题是, 一般情况下用 cb 抓跳做 Retention, 但平 u 接抓跳需要跳 9f 才能够越过 1 格高, 因此一般情况下是没法做 gultra retention 的.

可以做 gultra retention 的情况包括, 3A 的钥匙门, 5A 的机关块在内的各类移动块, 掉落块等.

其中掉落块 gultra retention 非常有趣, 它能让你在很短的距离上就再次落地, 恢复冲刺. 2022/11/28 的 9A.tas-9:08.845 就利用了这一点, 在 j-12 做出了一个天才般的优化!

通过这样的路线改动, 得以吃到平 u 速度再叠加 du*1.2, 并顺利地保留到了后面, 从而实现了老路线的反超.

2.15 core boost

发现于 2022/12/22

<https://discord.com/channels/403698615446536203/598945702554501130/1055324032364449882>

简单的说, 利用岩浆块的第一段回弹做出移动块多 cb.

core hyper 速度最多 $325 + 150 * 1.25 = 512.5$. 而且还需要一开始在右边, 以及等待岩浆块回弹. 远不如冲刺 + 岩浆块 4cb 直接有大概 $240 + (40 + 50)*4 = 600$ 的速度.

反 hyper + cb 打断倒还有些应用场景, 速度 500+, 而且能飞得很高.

此外, 岩浆块由于可以碎裂, 无需完全翻越, 因此给了后面很大的竖直位置调整空间.

案例: overclock + 岩浆块 5cb: 8b [c_05] 2022/12/22

2.16 Cross-screen cornerboosts

适用于这个边缘在跨版上方很近的情形.

以向右的情形为例:

一般来说, 低速情形下 (90 130) 双 cb 就会撞上墙壁, 产生 retention.

如果想要获得更多 cb, 那么需要先离开墙壁获得 wallSpeedRetained, 然后再 cb.

这需要在第 i 帧想办法向左移动, 并在第 i 帧的所有运算结束之后, 速度不朝左. 那么你就会在第 i+1 帧一切操作之前获得 wallSpeedRetained, 然后在撞到墙之前 cb 就行了 (在第 i+1 或 i+2 帧)(需要上升的高度较多的情形就不得不在第 i+2 帧 cb, 否则来不及跨过边缘获得 retention).

向上切版恰恰会使得速度设为 0.

于是

1		R, K, G
1		R, J, G
1		K
40		
1		R, J, G

即可.

在第 3 个 cb 前, 取决于边缘的高度, 可能还需要插入 1 帧 R 或 1 帧无操作.

使用案例: 1A 无冲的多处向上切版.

同样原理的, 如果是有不超过 (Madeline 身宽 + 一格) 距离的左右墙, 也可以用踢墙撞墙来将水平速度归零. 或者其他比如心之类的弹跳实体也行.

2.17 float rounding error

Part III

附录

1 学名-俗名/译名对照表

表 1: 学名-俗名/译名对照表

学名	中文俗名	译名	英文俗名	注释
MoveBlock	车/机关	车		
Glider	水母		jelly/jellyfish	
WallBoost			wallboost	
WallBooster	传送带		conveyer	冰模式下不这么叫
CrushBlock	Kevin 块		Kevin block	
Holdable	抓取物		throwable	
Player.Holding		手持物品		
FloatySpaceBlock	月亮块/月球块		moon block	
FlingBird	鸟		bird	
Puffer	河豚/鱼		puffer/fish	
Seeker	新浪/鱼		fish	
Spikes	尖刺			
CrystalStaticSpinner	圆刺		spinner	
TheoCrystal	Theo 水晶/水晶		theo	
Solid		固体/固块		
SolidTiles	墙体/砖块	墙体		
DashBlock				可撞碎的砖块
DashSwitch	按钮/开关	按钮		只能开 TempleGate
TouchSwitch	硬币/开关	硬币		Mural Skies 中的那些
SwitchGate		硬币门		2A 逃亡段的硬币门
LockBlock		钥匙门		3A 第一节的钥匙门
TempleGate		神庙门		开启/关闭的方式多样
TempleCrackedBlock				5A 新浪能撞碎的砖块
ZipMover	红绿灯/拉链块/滑块/机关	红绿灯块	traffic block/zipper	9A 也有 ZipMover, 主题为月亮
SwapBlock	滑块/机关			能远程响应冲刺的机关
StarJumpBlock				6A 开头与 8A 结尾的浮动块
BounceBlock	岩浆块			冰模式下不这么叫
DashAttack				

2 状态表

```
StNormal = 0;
StClimb = 1;
StDash = 2;
StSwim = 3;
StBoost = 4;
StRedDash = 5;
StHitSquash = 6;
StLaunch = 7;
StPickup = 8;
StDreamDash = 9;
StSummitLaunch = 10;
StDummy = 11;
StIntroWalk = 12;
StIntroJump = 13;
StIntroRespawn = 14;
StIntroWakeUp = 15;
StBirdDashTutorial = 16;
StFrozen = 17;
StReflectionFall = 18;
StStarFly = 19;
StTempleFall = 20;
StCassetteFly = 21;
StAttract = 22;
StIntroMoonJump = 23;
StFlingBird = 24;
StIntroThinkForABit = 25;
```

3 深度表

深度越高的, 越先更新. 以最近的子类为准.

渲染也是如此, 因此越深的物体确实在越“后面”的位置, 会被前面的物体挡住.

此表只表明大致情形, 具体深度取决于实体的构造函数.

TheoCrystal = 100.

Madeline = 0.

MoveBlock = -1.

Spikes = -1.

FlingBird = -1.

Glider = -5.

CrystalStaticSpinner = -8500.

Booster = -8500.

IceBlock = -8500.

Spring = 其所在的 Platform.depth + 1.

Platform = -9000.

ZipMover = -9999.

SwapBlock = -9999.

Snowball = -12500.

Puffer = 未指定.

WindController = 未指定.

```
public static class Depths{
    public const int BGTerrain = 10000;

    public const int BGMirrors = 9500;

    public const int BGDecals = 9000;

    public const int BGParticles = 8000;

    public const int SolidsBelow = 5000;
```

```
public const int Below = 2000;

public const int NPCs = 1000;

public const int TheoCrystal = 100;

public const int Player = 0;

public const int Dust = -50;

public const int Pickups = -100;

public const int Seeker = -200;

public const int Particles = -8000;

public const int Above = -8500;

public const int Solids = -9000;

public const int FGTerrain = -10000;

public const int FGDecals = -10500;

public const int DreamBlocks = -11000;

public const int CrystalSpinners = -11500;

public const int PlayerDreamDashing = -12000;

public const int Enemy = -12500;

public const int FakeWalls = -13000;

public const int FGParticles = -50000;

public const int Top = -1000000;
```

```
public const int FormationSequences = -2000000;  
}
```

4 英文社区常见术语

这里收录 Celeste Discord 上 TASING 版块常见的术语. 注意, 部分缩写并不受到广泛认可, 例如 wb.

表 2: 英文社区常见术语

完整名字	缩写	注释
Bunny hop	bhop	兔子跳
Spring cancel	sprancel	弹簧取消
Jellyvator		构词由 Jelly + Elevator 得到
Archie		蹲姿进泡泡抬升高度
Wallbounce	wb	蹭墙跳
Wallboost	wb	缩写与蹭墙跳一样
Corner boost	cb	
Double corner boost	dcb/2cb	
Triple corner boost	tcb/3cb	
Quatre corner boost	qcb/4cb	
Ceiling pop	cp	
Grounded ultra	gultra	

5 FlingBird

abs(Vel.X) 的范围	StFlingBird 时长
0 ~ 71	30 f
71 ~ 163	31 f
163 ~ 225	32 f
225 ~ 277	33 f
277 ~ 324	34 f
324 ~ 368	35 f
368 ~ 410	36 f
410 ~ 451	37 f
451 ~ 490	38 f
490 ~ 529	39 f
...	...

表 3: StFlingBird 时长

临界值:

71, 163, 225, 277, 324, 368, 410, 451, 490, 529, 567, 604,
641, 678, 714, 750, 786, 821, 857, 892, 927, 962, 997, 1032 ...

6 常用数据表

cb 翻越如 图 36 摆放的圆刺, 速度上限是 $4.5 * 60 + 4.33 + 4.33 - 40 = 238.66$. 同时, 需要将垂直亚像素调至最顶端的 0.25 px.

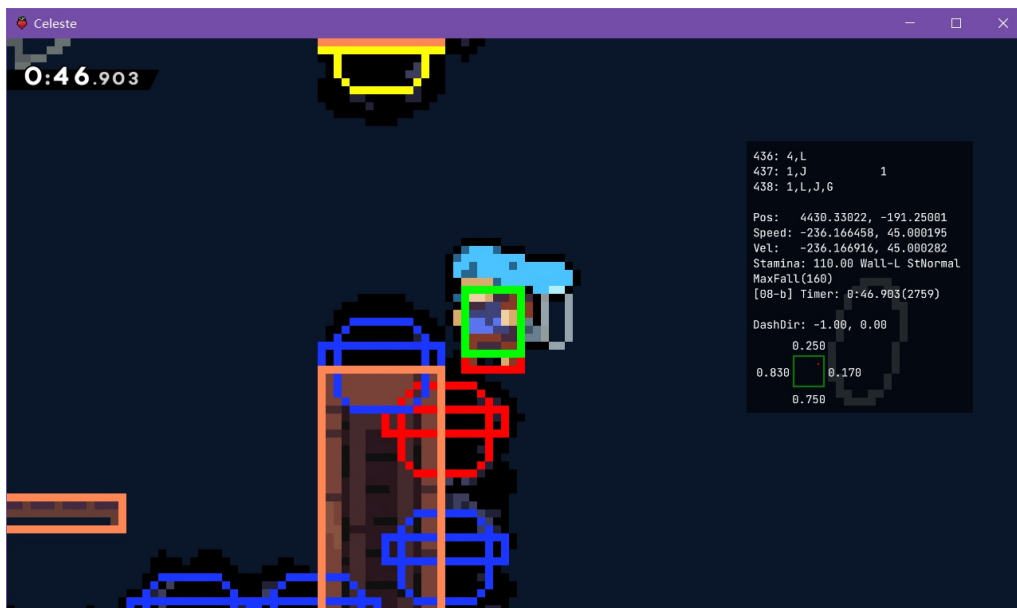


图 36: Spinner cb

6.1 快速重生

最快重试/自杀: 1,S ; 1,D,J ; 69,O

呃, 主动撞刺之类的会不会更快啊, 对于某些需要停留的情形 (e.g. Satellite FC 吃草莓).

确实如此, 少了开菜单的一帧.

不过撞刺会多 3 帧“冻结帧”(IGT 不走, 但也没显示 Frozen), 而自杀不会.

7 源码中常用的部分函数

- Math.Sign: 返回 0, -1, 1.

Part IV

学习资料

你可以点击超链接.

- 1) [Celeste TAS Reference](#)
- 2) [zball 的 Celeste 机制研究](#)
- 3) [一只胖熊 qwq 的 蔚蓝小课堂系列](#)
- 4) [TRB_ 叛逆王八的 tas 教程系列](#)
- 5) [一只蓝冰的 【更新至第 1 期】Tas 不再神秘! 这可能是最好的 tas 教学视频!!](#)
- 6) [桐 815 的 \[TAS\] celeste-单向高速案例分享](#)
- 7) [总有刁民想害我 a 的 蔚蓝 celeste 的 TAS 中常用的一些技巧合集 \(以及某些常见的容易引起误解的技巧\)](#)
- 8) [明月棕的 【蔚蓝】TAS 是怎样制作的? 一个简短的案例展示](#)
- 9) [早期版本的 Player.cs \(与当前版本差异大, 不建议使用\)](#)
- 10) [最新版本的源码. 请自行寻找反编译工具取得.](#)
- 11) [Getting Started with CelesteTAS: Mechanics and Tech](#)
- 12) [Getting Started with CelesteTAS: Routing and Optimization](#)
- 13) [Celeste TAS Tech](#)
- 14) [Celeste: The Art of Floating Point Manipulation: a Documentation](#)
- 15) [Celeste Discord server 中的众多交流](#)
- 16) [《Celeste 一些进阶 TAS 技巧》. 请翻阅 Celeste TAS 群的群文件.](#)
- 17) [《科学 CeilingPop》. 同上.](#)
- 18) [《岩浆块机制》. 同上.](#)
- 19) [Monocle-Reference](#)
- 20) [Subpixel manipulation tutorial](#)
- 21) [CelesteTAS Custom Info Bundles](#)
- 22) [useful celeste TAS input sequences](#)