2025-05-31

Blogs in May 2025

Archive of Blog posts in May 2025

Contents

2025-05-23

RPE 速度事件缓动的研究与适配

本文介绍了 RPE 速度事件缓动的实现原理与适配方法。

在 RPE 1.6.2 中，速度事件的定义得到了扩展，首次允许其拥有缓动类型（`easingType`）与缓动截取区间（`easingLeft`，`easingRight`）。

“**1.6.2 Changelog**

Added:

- **速度缓动 (见注记)**
- 自定义预制事件映射 (配置文件为 `./Resources/prefab_mapper.json`，默认 E 键生成一组重置事件)
- 可设置自动保存文件数量上限
- 上传至局域网服务器 (见注记)
- 可从 osu 和 malody 的谱面文件 (osz & mcz) 中导入 BPM 列表
- 选中单个音符时 (单选或多选)，`Ctrl+C` 会将音符的 x 坐标填入系统剪切板 (可选)
- 基本信息中显示创建和最后编辑时间
- 剪切板起点时间线染色 (可选)

Fixed:
- 现在点线切换选项也决定是否点音符换线
- 框选时点击事件不再会进行单选

Adjustments:
- 现在曲线填充快捷键 (默认 `Ctrl+F/G`) 会直接打开曲线填充面板

注记：
1. **约定：速度事件缓动不为 1 时，实际的速度变化与缓动的导函数形状相同，从而 floorposition 的变化遵循缓动曲线。为了兼容性，缓动为 1 时我们保持原含义不变，也即缓动为 1 和缓动为 5 都代表二次型的 floorposition 变化**
2. 端口始终为 8080，上传文件名始终为 test.pez。在软件关闭前可在同局域网下的其他设备上下载谱面
3. RPEVersion 已更新至 162

”

具体地，除了缓动类型为线性（`easingType = 1`）的情况照旧，在设置成**其余缓动类型（`easingType > 1`）**时，**note 速度将按照所设置的缓动函数的导函数进行变化**。

然而，导函数并不一定满足成为缓动函数的条件，需要对其进行额外的变换，从而导致这样的映射关系并不能保证 note 高度会随原函数进行变化。因此我们需要从定义出发，首先确定缓动函数的导函数及其变换，然后再通过积分运算得到符合速度规律的原函数作为 note 高度的变化依据。

1.1 缓动导函数的确定与归一化

注意到，缓动函数一般需要满足：定义域 $[0,1]$，起始值为 0，终止值为 1。

设缓动原函数为

$$f(t),\ t \in [0,1]$$

则我们需要找到一个 $g(t) = kf'(t) + c$ 使得

$$g(0) = 0,\ g(1) = 1$$

为了普遍性（同时为了确保精度），不妨设 $g(0) = a,\ g(1) = b$。由

$$g(0) = kf'(0) + c = a$$
$$g(1) = kf'(1) + c = b$$

我们不难得到

$$k = \frac{b-a}{f'(1) - f'(0)}$$
$$c = a - kf'(0)$$

由此，note 速度便可以表示为

$$\begin{aligned} g(t) &= kf'(t) + c \\ &= \frac{b-a}{f'(1) - f'(0)} f'(t) + a - \frac{b-a}{f'(1) - f'(0)} f'(0) \end{aligned}$$

代入 $a = 0, b = 1$，有

$$g(t) = \frac{1}{f'(1) - f'(0)} f'(t) - \frac{f'(0)}{f'(1) - f'(0)}$$

1.2 实际缓动函数的确定

在代码实现中，我们需要知道 note 高度的函数，也就是对 $g(t)$ 进行积分：

$$\begin{aligned} \int_0^t g(t)\,\mathrm{d}t &= \int_0^t [kf'(t) + c]\,\mathrm{d}t \\ &= \Big[kf(t) + ct\Big]_0^t \end{aligned}$$

由此，我们只需要知道 $f(t), k, c$ 即可计算出 note 高度。

1.3 代码实现

现成的速度事件处理逻辑：

```
export const getIntegral = (
  event: SpeedEvent | undefined,
  bpmList: Bpm[],
  beat: number | undefined = undefined,
): number => {
  if (!event) return 0;
  if (beat === undefined || beat >= event.endBeat) beat = event.endBeat;
  const lengthSec = getTimeSec(bpmList, event.endBeat) - getTimeSec(bpmList,
event.startBeat);
  return ((event.start + (getEventValue(beat, event) as number)) * lengthSec) / 2;
};
```

其中，`getTimeSec` 用于获取指定拍数所对应的时间，`getEventValue` 用于获取指定拍数时指定事件的值。

接下来，我们对速度事件的缓动进行处理。

1. 首先判断需要使用到缓动处理的情况：

```
if ('easingType' in event && event.easingType > 1) {
  // 需要处理缓动
}
```

2. 然后计算缓动原函数端点处的导数值 $f'(0), f'(1)$：

```
const df0 = derivative(event.easingType, 0);
const df1 = derivative(event.easingType, 1);
```

其中 `derivative` 函数负责计算缓动函数的导数值，大致定义如下：

```
export const derivative = (
  type: number,
  x: number,
  epsilon = 1e-12,
) => {
  const leftX = Math.max(1e-16, x - epsilon);
  const rightX = Math.min(1 - 1e-16, x + epsilon);
  const leftY = calculateEasingValue(type, leftX);
  const rightY = calculateEasingValue(type, rightX);
  return (rightY - leftY) / (rightX - leftX);
};
```

其中 `calculateEasingValue` 函数用于获取指定类型的缓动函数在指定进度的值，`epsilon` 用于控制计算精度。
注意：此处为了避免缓动函数有时会在端点处特判的情况，将自变量的取值范围限制在了 $[10^{-16}, 1-10^{-16}]$ 内。

3. 计算 k, c：

```
const k = (event.end - event.start) / (df1 - df0);
const c = event.start - k * df0;
```

此处直接将 a, b 取为事件的起始值与终止值，从而减少计算量，提高浮点计算精度。

4. 计算缓动值（note 高度）：

```
const x = (beat - event.startBeat) / (event.endBeat - event.startBeat);
const progress = integrate(event.easingType, x, k, c);
```

其中，`integrate` 函数定义如下：

```
export const integrate = (
  type: number,
  x: number,
  k: number,
  c: number,
) => {
  return k * calculateEasingValue(type, x) + c * x;
};
```

5. 最后，进行单位换算并返回结果：

```
const lengthBeat = event.endBeat - event.startBeat;
return (progress * lengthSec) / lengthBeat;
```

此处的 `lengthSec` 与 `lengthBeat` 分别表示事件的持续秒数与持续拍数。

1.4 缓动截取

缓动截取的本质是对缓动函数的定义域与值域进行缩放。在 RPE 中对速度事件的缓动函数进行截取，不难发现，是基于原函数进行的操作。

我们可以直接套用现成的缓动截取处理逻辑。对于定义域的缩放，已在 `calculateEasingValue` 函数中实现：

```
const progress = func(easingLeft + (easingRight - easingLeft) * x);
```

值域的缩放也就是函数的归一化，我们在前面也已经实现过了。

以下是考虑了缓动截取的逻辑实现：

```
const lengthBeat = event.endBeat - event.startBeat;
const easingLeft = 'easingLeft' in event ? event.easingLeft : 0;
const easingRight = 'easingRight' in event ? event.easingRight : 1;
const df0 = derivative(event.easingType, 0, easingLeft, easingRight);
const df1 = derivative(event.easingType, 1, easingLeft, easingRight);
const k = (event.end - event.start) / (df1 - df0);
const c = event.start - k * df0;
const x = (beat - event.startBeat) / (event.endBeat - event.startBeat);
const progress = integrate(event.easingType, x, k, c, easingLeft, easingRight);
return (progress * lengthSec) / lengthBeat;
```

对于导数值的计算，只需要将事件的截取区间直接向下传递到 `calculateEasingValue` 函数中即可，其余部分与之前的处理逻辑一致：

```
export const derivative = (
  type: number,
  x: number,
  easingLeft = 0,
  easingRight = 1,
  epsilon = 1e-12,
) => {
  const leftX = Math.max(1e-16, x - epsilon);
  const rightX = Math.min(1 - 1e-16, x + epsilon);
  const leftY = calculateEasingValue(type, leftX, easingLeft, easingRight);
  const rightY = calculateEasingValue(type, rightX, easingLeft, easingRight);
  return (rightY - leftY) / (rightX - leftX);
};
```

同理，对于实际缓动函数的计算，也只需要将截取区间传递到 `calculateEasingValue` 函数中即可，不需要其他改动：

```
export const integrate = (
  type: number,
  x: number,
  k: number,
  c: number,
  easingLeft = 0,
  easingRight = 1,
) => {
  return k * calculateEasingValue(type, x, easingLeft, easingRight) + c * x;
};
```