PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

Avaliando o uso de RxLua em Jogos

Felipe Pessoa de Freitas

PROJETO FINAL DE GRADUAÇÃO

CENTRO TÉCNICO CIENTÍFICO - CTC
DEPARTAMENTO DE INFORMÁTICA
Curso de Graduação em Ciência da Computação

Rio de Janeiro, Junho de 2020

Felipe Pessoa de Freitas

Avaliando o uso de RxLua em Jogos

Relatório de Projeto Final, apresentado ao programa de Ciência da Computação da PUC-Rio como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientadora: Noemi de La Rocque Rodriguez

Rio de Janeiro

Junho de 2020

Resumo

Pessoa de Freitas, Felipe. de La Rocque Rodriguez, Noemi. Programação reativa em Lua aplicada em jogos. Rio de janeiro, 2019. 92p. Relatório de Projeto Final – Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

Esse trabalho estuda o uso de ReactiveX no ambiente de jogos. São discutidas tanto vantagens quanto desvantagens do uso desse paradigma. O objeto de estudo desse projeto foi um jogo desenvolvido de duas maneiras diferentes, uma utilizando Rx e outra de modo convencional, a fim de realizar análises sobre as diferenças entre eles. A linguagem de programação Lua e a *engine* LÖVE foram as ferramentas utilizadas para implementá-los.

Palavras-chave

ReactiveX; Rx; Programação Reativa; Desenvolvimento de jogos; Lua.

Resumo

Pessoa de Freitas, Felipe. de La Rocque Rodriguez, Noemi. Programação reativa em Lua aplicada em jogos. Rio de janeiro, 2019. 92p. Relatório de Projeto Final I – Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

This work studies the use of ReactiveX in the videogame environment. Both advantages and disadvantages of using this paradigm are discussed. The object of study for this project was a game developed in two different ways, one using Rx and the other in a conventional way, in order to carry out analyzes on the differences between them. The programming language Lua and the *engine* LÖVE were the tools used to implement them.

Keywords

ReactiveX; Rx; Reactive Programming; Game development; Lua.

Sumário

Lista de Códigos

1 Introdução

Ao longo dos anos, o mercado de videogames teve um crescimento bastante elevado. Entre 2011 e 2020, ele praticamente dobrou [1] e, com isso, também houve um crescimento de desenvolvedores, gêneros de jogos [2] e jogadores. Com uma demanda crescente como essa, várias tecnologias começaram a ser adaptadas a esse processo de desenvolvimento.

As linguagens mais comumente usadas no desenvolvimento de jogos atualmente são linguagens multiparadigma com orientação a objetos, como C++, Java e C# - sendo a primeira a mais utilizada - e em geral são utilizadas juntamente com tratamento de eventos, uma vez que jogos são programas altamente baseados em eventos. Por isso, as *engines*[1] costumam disparar eventos para que o programador possa tratá-los como for melhor, como o *input* do jogador ou colisões. No entanto, não existem um padrão de tecnologia e linguagem de programação para o desenvolvimento de jogos que seja usado por todo mundo.

E por esse motivo, é sempre possível aprimorar o desenvolvimento combinando técnicas diferentes, como a programação reativa. Até o momento, ela ainda não é muito utilizada, principalmente por já existirem outras tecnologias e linguagens de programação no mercado que são utilizadas há anos.

Programação reativa [3] trabalha com um modelo não padrão de programação, em contraposição ao paradigma de programação imperativa e de estados, exigindo do programador uma forma diferente de trabalhar. Assim como outros paradigmas, este funciona a base de eventos, de maneira que os objetos reagem à mudança de dados. No entanto, esses eventos são implementados na forma de *streams* de dados assíncronas, e o programa passa a ser pensado em termos das interações entre elas.

Nesse projeto estudei programação reativa e explorei o uso da mesma no desenvolvimento de jogos eletrônicos. Pelo fato de serem programas muito baseados em eventos, é uma área interessante a ser investigada em como pode ser beneficiada desse paradigma. O foco foi o ambiente de jogos 2D[2] para computador. Para a implementação dos elementos que compõem o projeto, foi utilizada a linguagem Lua[3] e a *engine* LÖVE[4].

[1] https://pt.wikipedia.org/wiki/Motor_de_jogo

[2] https://pt.wikipedia.org/wiki/Gráficos_de_jogos_eletrônicos#2D

[3] https://www.lua.org/

[4] https://love2d.org/

2 Programação reativa e *ReactiveX*

Programação reativa é um paradigma que tem o foco em resolver problemas que *softwares* altamente dependentes em eventos para seu funcionamento, principalmente da interação com o usuário, acabam encontrando. Ela encoraja a modelar o programa para tratar propagações de mudanças e as reações à elas. E a maneira como ela faz isso é a partir de *streams* de dados, em que cada uma é basicamente uma sequência de eventos ordenados no tempo, que são capturadas de forma assíncrona. É, por um lado, muito semelhante ao padrão *Observer*, no entanto em vez de trabalhar com um evento por vez, considera uma sequência de eventos que são recebidos ao longo do tempo. Com isso, é possível aumentar o nível de abstração do código, podendo assim focar na interdependência de eventos que definem a lógica do programa.

2.1 *ReactiveX*

A biblioteca *Reactive Extensions*, ou *ReactiveX* [12], originalmente desenvolvida para .NET e posteriormente portada para diversas linguagens [5], oferece uma série de ferramentas para que linguagens imperativas possam trabalhar com dados de forma assíncrona e criar programas baseados em eventos. Nessa biblioteca são utilizados alguns conceitos já conhecidos do padrão *Observer*, como *Observable* (que produz valores) e *Observer* (que consome esses valores) [11].

A ideia é que as *streams*, também chamadas de *Observable*, podem ser criadas de diversas maneiras diferentes, não somente a partir de eventos de mouse como cliques e *hover*, mas também a partir de variáveis, *inputs* do usuário, estruturas de dados, entre outros. E a maneira como capturamos esses eventos que são transmitidos pela *stream* é através do mecanismo de *subscribe*. Com isso, estamos fazendo a ligação de um *Observable* com um *Observer*.

O *Observable*, então, ao produzir um valor novo (um evento), chama os métodos apropriados de todos os *Observers* que estão ligados a ele (pelo método `subscribe`). Estes são:

- `onNext`: o *Observable* chama esse método sempre que produzir um novo valor.

  Parâmetro: o valor produzido.

[5] http://reactivex.io/languages.html

- `onError`: o *Observable* chama esse método ao falhar em gerar um novo valor ou ao encontrar um outro erro. Após isso não irá fazer outras chamadas `onNext` e `onCompleted`.

  Parâmetro: indicador da causa do erro.

- `onCompleted`: o *Observable* chama esse método após a última chamada de `onNext`, se não tiver encontrado nenhum erro.

Uma das características mais importantes desse paradigma é a existência de uma série de métodos utilizados para manipular *streams*, provendo facilidades como filtrar, transformar e mapear valores, combinar uma ou mais *streams*, entre outros. Com isso, podem ser geradas novas *streams* a partir das manipulações executadas na original.

É importante ressaltar que existe uma diferença entre o que são chamados de *hot* e *cold Observables*. O *hot* é um *Observable* que começa sua produção de valores a partir do momento em que é criado, de maneira que *Observers* que posteriormente venham a dar `subscribe` irão começar a observar a *stream* já emitindo valores. Por outro lado, o *cold* é um *Observable* que espera um *Observer* dar `subscribe` para começar a emitir valores, de maneira que o *Observer* irá poder observar toda a *stream* desde o início.

E além disso, existe um outro conceito, chamado *Subject*[6], que pode atuar tanto como um *Observer* quanto como um *Observable*. Ele pode ser usado como um *hot Observable* (como vai ser mostrado em alguns casos nesse relatório). Existem alguns tipos de *Subject*, porém os únicos usados nesse projeto foram *Subject* e *BehaviorSubject*.

Um outro elemento importante de *ReactiveX* é o *Scheduler* [13]. Ele permite executar *Observables* em *threads* diferentes, assim como agendar tarefas.

3 Situação Atual

Atualmente, existem diversas maneiras de desenvolver um jogo eletrônico. O caminho mais usual é através de *engines* de desenvolvimento de jogos, cada uma com suas vantagens e desvantagens. A quantidade de *engines* que estão disponíveis para serem utilizadas é enorme e abrange um público bem extenso,

[6] http://reactivex.io/documentation/subject.html

até para indivíduos que não possuem conhecimento de programação, mas esse não será o foco do projeto.

Cada *engine* costuma ter uma ou mais linguagens de programação que o desenvolvedor pode escolher para escrever seus *scripts* que irão controlar o jogo. Podem ser scripts que irão controlar a câmera, o personagem, o inimigo, ou qualquer outro elemento do jogo. Mas o que muitos deles têm em comum é a implementação de mecânicas [14].

Mecânicas são ações básicas que, combinadas, podem determinar um gênero de jogo ou um comportamento do jogador ou de NPC's [7]. Por exemplo, uma mecânica de pulo irá fazer com que, ao jogador apertar determinado botão, o personagem pule no mundo virtual. E cada uma dessas mecânicas pode ser implementada de diversas maneiras diferentes.

No entanto, não existe um padrão de programação para essa implementação, o que leva a muitos desenvolvedores acabarem escolhendo o caminho mais fácil ou acessível. Isso pode se dar porque não sabem da existência de outras maneiras de programar ou porque não encontram um motivo grande que faça valer a pena mudar de paradigma.

Por esse motivo, é interessante buscar a existência de bibliotecas auxiliares, pois ao mesmo tempo em que fornecem funcionalidades normalmente difíceis de implementar, também elevam o nível de abstração do código, de maneira que o programador pode focar na implementação do programa em si. Já é possível encontrar algumas bibliotecas específicas de programação reativa voltadas para o uso dentro de *engines*. Alguns exemplos são:

- UniRx [4], que foi desenvolvida para ser utilizada com a linguagem C# na *engine* Unity[8].
- RxLove [5], para ser usado com a *engine* LÖVE, e que requer a RxLua [6], desenvolvida em Lua.

A RxLove será parte do estudo desse projeto.

Essas bibliotecas permitem utilizar o paradigma de programação reativa no meio de desenvolvimento de jogos sem maiores complicações. Além disso, programação reativa já é usada em muitas situações que não envolvem jogos. Até o momento, já foram implementadas diversas bibliotecas em diferentes linguagens,

[7] https://pt.wikipedia.org/wiki/Personagem_não_jogável

[8] https://unity.com/pt

alguns exemplos são RxJava, RxCpp, RxPy, entre outros[9]. Muitas empresas, como Microsoft, Netflix e GitHub já utilizam esse novo paradigma em suas aplicações[10]. Contudo, não existe muito material acerca de como utilizar esse paradigma no desenvolvimento de jogos. E por isso, esse projeto tem como um de seus objetivos contribuir para essa área.

4 Objetivos do trabalho

O principal objetivo do projeto é mostrar como a programação reativa pode ser utilizada no processo de desenvolvimento de jogos e, a partir disso, explorar as consequências que a mesma possa acarretar, sejam elas positivas ou negativas. Essas podem ser desde possíveis prevenções de *bugs* até mudanças na estrutura e legibilidade do código.

Pretendo, assim, poder ajudar outros desenvolvedores de jogos a conhecer esse paradigma, mostrando como ele pode ser aplicado nesse mundo. Além disso, possivelmente despertar uma vontade de buscar novas maneiras de desenvolver jogos.

Como base do projeto, irei apresentar algumas mecânicas de jogos que selecionei e suas possíveis implementações com programação reativa que irão compor um jogo. Serão abordados conceitos gerais desse paradigma, focados nos exemplos de códigos. Os produtos desenvolvidos durante o projeto utilizaram a linguagem de programação Lua e a *engine* LÖVE, também conhecida como Love2D.

5 Atividades Realizadas

5.1 Estudos preliminares

Antes desse projeto, eu já possuia uma experiência razoável com desenvolvimento de jogos. Comecei a estudar essa área de maneira autodidata logo após ingressar na faculdade, pois já tinha estudado programação antes, e tenho estado continuamente envolvido com o assunto. Hoje em dia, trabalho ativamente com o desenvolvimento de jogos para computador e celular.

Eu possuía uma breve experiência com programação reativa, pois utilizei uma biblioteca chamada UniRx, implementada para ser utilizada com a *engine*

[9] http://reactivex.io/languages.html

[10] http://reactivex.io/

Unity. Ela foi utilizada por mim e minha equipe onde trabalho como um experimento, para testar se esse paradigma era interessante de ser seguido ou não. Começamos a utilizá-la em um pequeno projeto, porém não utilizamos o seu potencial, limitando-nos a elementos simples. No entanto, esse teste nos deixou bastante intrigados, e esse é um dos motivos pelo qual escolhi continuar estudando esse paradigma.

5.2 Estudos de tecnologia

O primeiro foco de estudo foi estudar programação reativa em si a partir de algumas pesquisas e artigos, sendo um deles o *"'A Survey on Reactive Programming"* [8]. Após obter uma pequena base de conhecimento, passei a estudar a linguagem de programação Lua e a *engine* LÖVE.

Após isso, passei por uma parte importante do projeto que foi analisar bibliotecas de programação reativa em Lua para tomar conhecimento das tecnologias já existentes e avaliar se seria necessário implementar uma biblioteca própria a ser utilizada durante o restante desse processo.

Encontrei duas bibliotecas, RxLua (citada previamente) e FRLua [7]. As bibliotecas acabaram se provando extremamente robustas, o que me levou a optar por não implementar a minha própria. As duas possuem métodos familiares de programação reativa (ReactiveX), possibilitando o uso de qualquer uma delas a principio, o que inicialmente me fez ficar na dúvida de qual eu iria utilizar. No entanto, acabei escolhendo a primeira, pois além de fazer parte das bibliotecas citadas no ReactiveX [11], ela já possuia uma integração com a *engine* LÖVE, a partir de uma outra biblioteca chamada RxLove.

5.2.1 RxLua, RxLove e LÖVE

Para familiarizar o leitor com as tecnologias utilizadas, vou apresentar o básico de cada uma, o suficiente para poder entender os exemplos de código que discuto em seções posteriores.

A biblioteca RxLua é a implementação oficial de ReactiveX da linguagem Lua. Ela oferece a maioria dos métodos comuns citados na lista de operadores de Reactive Extensions. Alguns dos operadores que utilizei nesse projeto incluem filtragem, mapeamento e combinações de *streams*, como o `filter`, `map` e `merge`.

[11] http://reactivex.io/languages.html

Uma grande vantagem dessa biblioteca é que ela é facilmente extensível. Durante o projeto implementei três métodos novos: `Execute`, `TimeInterval` e `TimeStamp`, que serão discutidos posteriormente e cujos códigos se encontram no apêndice.

Além disso, a RxLua implementa 3 tipos de *schedulers*:

- *ImmediateScheduler*: usado para executar operações imediatamente
- *CooperativeScheduler*: usado para gerenciar *Observables* usando corrotinas e um relógio virtual que deve ser atualizado manualmente
- *TimeoutScheduler*: usado para executar operações após um intervalo específico de tempo

No entanto, apesar de poderem ser usados independentemente, algumas implementações de *ReactiveX*, como no caso da RxLua, possuem operadores que recebem como parâmetro um *scheduler*. Isso faz com que o *Observable* execute em um *scheduler* específico.

A *engine* LÖVE possui uma estrutura simples, em que executa um arquivo de entrada onde são definidos no geral 3 métodos principais:

- `love.load()`: é executado pela *engine* no início da execução, nesse método geralmente é feita a inicialização necessária para o jogo
- `love.update(dt)`: é o *loop* principal do jogo, é executado continuamente e é onde se altera o estado do jogo
- `love.draw()`: é o método responsável por desenhar os elementos do jogo, também é chamado continuamente pela *engine*

Alguns eventos disparados pela *engine* podem ser capturados em métodos específicos, que também são chamados, se existirem, nesse arquivo de entrada:

- `love.keypressed(key)`: chamado quando uma tecla é pressionada
- `love.keyreleased(key)`: chamado quando uma tecla é solta

A RxLove, por sua vez, cria *Subjects* a partir dos métodos e eventos disparados da *engine* LÖVE - como `love.update`, `love.keypressed` - de modo que possam ser observados. Isso será demonstrado na seção 7.1.

Uma característica que eu havia utilizado na biblioteca UniRx e que não está presente na RxLua é o conceito de *reactive properties*. Com isso, é possível fazer algo como:

```
1  // Reactive Notification Model
2  public class Hero {
3      public ReactiveProperty<long> Health { get; private
           set; }
4
5      public Hero(int initialHp){
6          // Declarative Property
7          Health = new ReactiveProperty<long>(initialHp);
8          CurrentHp.Subscribe(value => print("Hero health: "
               + value);
9      }
10
11     public void Damage(int damage){
12         Health.Value -= damage
13     }
14 }
```

Código 1: *reactive properties* em UniRx.

Essa funcionalidade permite transformar variáveis que uma classe precisaria ter em propriedades reativas, ou mais precisamente, em *hot Observables*. Dessa maneira, é possível, por exemplo, observar a vida atual do jogador dando `subscribe` na propriedade e reagindo de acordo. E para acessar ou escrever o valor basta usar o campo *value* da propriedade (como no código 1).

Para realizar algo similar com a biblioteca RxLua, é preciso criar um objeto chamado de *BehaviorSubject*. Funciona de maneira parecida, em termos de ser um *hot Observable*. No entanto para acessar o valor é preciso chamar seu método `getValue()` e para escrever é necessário chamar o método `onNext()` passando o valor como parâmetro, como mostrado no código 2. Na linha 2 é criado um *BehaviorSubject* e associado ao campo `hero.health`. Já na linha 7, o método `getValue()` está sendo chamado para recuperar o último valor emitido, e assim na linha 8 é chamado o método `onNext()` para atualizar o campo com seu valor atual subtraído de uma quantia recebida.

```
1  --Player.lua
2  hero.health = rx.BehaviorSubject.create(initialHp)
3  --criando um Observer implicitamente a partir do
       metodo subscribe
4  hero.health:subscribe(function(value)
5      print("Hero health: " .. value)
6  end)
7  hero.Damage = function(value)
8      currentValue = hero.health:getValue()
9      hero.health:onNext(currentValue - value)
10 end
```

Código 2: *BehaviorSubject* em RxLua.

5.3 Estudos conceituais - mecânicas

Depois do estudo das tecnologias, passei para a parte das mecânicas a serem trabalhadas. Preparei, então, uma lista a partir do meu conhecimento prévio dessa área e estudando alguns materiais como o livro de Rogers [15], para saber quais mecânicas são mais comuns ou famosas em jogos. Categorizei-as em grupos para poder escolher as que de fato seriam implementadas. A criação desses grupos levou em conta dois critérios: a semelhança entre o sistema básico de cada mecânica e a maneira como as mecânicas poderiam ser implementadas com programação reativa.

A partir disso, escolhi mecânicas de diferentes grupos com o objetivo de ter opções variadas. Foram separadas as mecânicas básicas a serem implementadas para o funcionamento do jogo e as destinadas como alvo de um estudo mais aprofundado. As mecânicas básicas selecionadas foram: a movimentação de um personagem (horizontal e vertical) e a movimentação da câmera (acompanhando o jogador). As que foram escolhidas para serem melhor estudadas foram:

- **Plataformas móveis**: plataformas que se movem constantemente de um ponto a outro
- ***Falling platforms***: plataformas que quebram após determinado tempo em que o jogador se encontra em cima delas

- ***HUD (heads-up display)***[12]: utilizada para indicar ao jogador diversas informações através de elementos visuais
  - Regeneração da barra de vida
  - Aumento do contador de pontos
- ***Quick Time events***[13]: eventos que requerem *inputs* rápidos do jogador
- **Coletáveis**: objetos que podem ser coletados
- **Perigos**:
  - Objeto que "esmaga" o jogador
  - Piso que afeta negativamente o jogador ao passar em cima
  - Alerta de perigo iminente
- ***Slow Motion***: jogo em "câmera lenta" em determinadas situações
- **Campo de visão/detecção**: área predefinida onde um inimigo pode "enxergar" o jogador, que deve ser evitada pelo mesmo
- **Projétil**:
  - Jogador/inimigo podem atirar projéteis
  - Jogador pode bloquear ou refletir projéteis atirados por inimigos
- ***Crafting***: sistema onde é possível juntar diversos recursos do jogo para formar novos objetos

No entanto, a lista se estendeu mais do que o esperado e, pelo tempo proposto do projeto ser limitado, realizei uma nova etapa de seleção, de maneira a tratar desafios diferentes. As mecânicas selecionadas e implementadas serão apresentadas na seção 7.

5.4 Testes e Protótipos para aprendizado e demonstração

No primeiro semestre de desenvolvimento desse projeto, eu estudei e implementei algumas mecânicas da lista previamente citada. Essas foram:

- Barra de vida

[12] https://pt.wikipedia.org/wiki/HUD_(jogo_eletrônico)
[13] https://pt.wikipedia.org/wiki/Quick_Time_Event

- Projétil
- Alerta de perigo
- Campo de visão/detecção

Isso me permitiu aprender e adquirir familiaridade tanto com a programação reativa quanto com a biblioteca utilizada para desenvolver o produto.

5.5 Método

Para esse projeto segui um modelo de *scrum* [16]. Dessa maneira, pude me organizar durante o semestre e planejar as tarefas que iria realizar em cada semana do projeto. Além disso, utilizei uma ferramenta para medir o tempo gasto em cada tarefa para que eu conseguisse estimar melhor as tarefas futuras e manter um controle maior sobre as mesmas.

Utilizando esses métodos, consegui dividir o desenvolvimento do projeto em algumas etapas. Primeiramente, realizei alguns testes iniciais com a biblioteca RxLove, para validar seu funcionamento. Em seguida, para praticar o uso da biblioteca, resolvi recodificar um jogo de exemplo da *engine* LÖVE para utilizar programação reativa.

A segunda etapa foi realizar alguns testes para esclarecer algumas dúvidas que tinha da biblioteca UniRx, no entanto não consegui produzir resultados satisfatórios inicialmente. Ao longo do desenvolvimento do projeto pude compreender melhor as funcionalidades presentes nas diferentes bibliotecas de *Reactive Extensions* e consegui esclarecer minhas dúvidas, ao estudar a questão da similaridade de *ReactiveProperty* da UniRx e *BehaviorSubject* da RxLove, discutido na seção 5.2.1.

A terceira etapa foi o desenvolvimento de um *playground* com algumas mecânicas básicas selecionadas. Essa foi a última e mais importante parte do projeto final I.

E, finalmente, no projeto final II, continuei desenvolvendo as mecânicas da lista. Também foi a etapa de realizar uma análise entre programação reativa e programação convencional.

Todas as mecânicas e análises estão detalhadas na seção 7.

6 Projeto e especificação do sistema

O objeto de estudo do projeto foi um jogo implementado de duas maneiras diferentes. Enquanto uma foi desenvolvida utilizando Rx, a outra foi implementada de maneira convencional.

O jogo é do gênero de plataforma, com mecânicas básicas comuns ao gênero como movimentação lateral, pulo, coletáveis, plataformas móveis e outras um pouco mais avançadas. Algumas dessas mecânicas incluem inimigos com campo de detecção e *quick time events*. O *input* para controlar o jogador é todo feito pelo teclado.

7 Implementação e análise

Para focar o desenvolvimento na implementação das mecânicas, utilizei um programa chamado *Tiled*[14] para a criação do mapa do jogo e uma biblioteca chamada *Simple Tiled Implementation*[15] para carregar esse mapa e cuidar da renderização do mesmo.

É importante ressaltar que a maneira como cada mecânica foi implementada não é única e não foi pensada para ser necessariamente a melhor, e sim para apresentar uma variedade maior de elementos de programação reativa.

Para cada mecânica, implementei uma versão reativa e uma convencional, para poder fazer a análise e comparação dessas duas maneiras de programar. Foquei a implementação em Rx para evitar ao máximo soluções convencionais, enquanto que na implementação convencional foquei em utilizar somente o básico, para que a análise ficasse compatível em cada mecânica.

Na maioria das explicações sobre a implementação das mecânicas o código a que me refiro estará depois do texto em questão.

Inclui no apêndice diagramas para melhor compreender as implementações utilizando Rx e imagens retiradas do jogo para ajudar a visualizar o mesmo caso as explicações não sejam suficientes.

[14] https://www.mapeditor.org/

[15] https://github.com/karai17/Simple-Tiled-Implementation

7.1 Mecânicas de input

As mecânicas de *input* do jogador foram muito simples de serem programadas. Por isso, reuni nessa subseção o que todas elas têm em comum, a utilização de eventos como `keypressed` e outros. Para exemplificar, utilizei parte do código relativo ao jogador.

7.1.1 Rx

No caso da programação reativa, utilizei os eventos de `keypressed` e `keyreleased` da *engine* LÖVE que foram transformados em *Subjects* pela biblioteca RxLove. Dessa maneira, é possível aplicar os operadores de Rx nesses eventos. Eu então filtro eles com o método `filter`, que retorna um *Observable* cujos valores emitidos respeitam a função recebida.

```
1  --Player.lua
2  love.keypressed
3      :filter(function(key) return key == 'space' end)
4      :subscribe(function()
5          hero.shoot()
6      end)
7
8  love.keyreleased
9      :filter(function(key) return key == 'a' or  key
           == 'd' end)
10     :subscribe(function()
11         hero.stopMoving()
12     end)
```

Código 3: *Input* em Rx utilizando o método `filter`.

7.1.2 Convencional

Sem a programação reativa, utilizei os mesmos eventos, porém diretamente como são mandados na *engine*. Nesse exemplo, mostro apenas o evento `keypressed`, mas o mesmo ocorre para `keyreleased`.

```
1  --main.lua
2  function love.keypressed(key)
3      -- Sends to Player
4      hero.keypressed(key)
5  end
6
7  --Player.lua
8  hero.keypressed = function (key)
9      if key == 'space' then
10         hero.shoot()
11     end
12 end
```

Código 4: *Input* convencional.

7.1.3 Análise

Nesse caso, as duas maneiras ficaram bem similares. A única vantagem da programação reativa foi o fato de poder acessar a *stream* diretamente e não precisar de outro módulo para passar o evento adiante.

7.2 Barra de vida

A primeira mecânica escolhida foi a barra de vida. Ela consiste em uma representação da vida do jogador, que diminui sempre que o mesmo sofre dano. Além disso, para um *game feel* [16] melhor, escolhi implementar uma segunda barra de vida, por trás da original, que espera uma quantidade de tempo específica antes de diminuir.

7.2.1 Rx

Comecei então com o código 5, que utiliza o método `delay`. Nesse bloco, é definida a vida do jogador como um *BehaviorSubject* (um *hot Observable*) que recebe um valor inicial. Na linha 4 o *Observable* inicial que representa a vida do jogador é transformado em outro *Observable*, retornado pelo método `delay`. No

[16] https://pt.wikipedia.org/wiki/Game_feel

entanto esse novo *Observable* espera 1 segundo antes de produzir cada valor recebido.

Portanto, digamos que o *Observable* `hero.health` recebe um novo valor em **t0 = 0**. O *Observable* retornado pelo método `delay` irá receber esse valor, esperar 1 segundo, e então produzir o mesmo valor em **t1 = 1**.

Por fim é feito um `subscribe`, que cria implicitamente um *Observer* cujo método `onNext` só tem uma função: associar o valor recebido à variável `hero.backHealth`, correspondente à segunda barra de vida, que fica por trás da original.

```
1 --Player.lua
2 hero.health = rx.BehaviorSubject.create(initialHealth)
3 --Espera 1 segundo antes de atribuir o valor
4 hero.health:delay(1, scheduler)
5             :subscribe(function (val)
6                  hero.backHealth = val
7              end)
```

Código 5: Barra de vida em Rx utilizando o método `delay`.

No entanto, quando o jogador recebia danos continuamente, a barra de vida diminuía aos poucos, o que não era desejado. Por exemplo, se o jogador sofrer dano nos segundos **t0 = 0**, **t1 = 1** e **t2 = 2**, a barra detrás irá diminuir em **t1 = 1**, **t2 = 2** e **t3 = 3**. Contudo, o efeito desejado era a barra de vida secundária esperar todos os valores que ocorreram em um intervalo de tempo curto antes de aplicar a mudança de valor.

Assim, no código 6 foi utilzado o método `debounce`. Ele retorna um *Observable* que roda um cronômetro interno, utilizando o *scheduler*, nesse caso de até 1 segundo. Se ele não receber nenhum valor ao final da contagem, é emitido o último valor recebido. A cada valor recebido, o cronômetro recomeça.

```
1 --Player.lua
2 hero.health = rx.BehaviorSubject.create(initialHealth)
3 --Atualiza variavel depois de 1 segundo sem receber
    novos valores
4 hero.health:debounce(1, scheduler)
5             :subscribe(function (val)
6                  hero.backHealth = val
7              end)
```

Código 6: Barra de vida em Rx utilizando o método `debounce`.

7.2.2 Convencional

Sem as funcionalidades da biblioteca de programação reativa, para chegar no mesmo resultado, tive que controlar o tempo manualmente. Primeiramente, tive que decidir onde tratar esse controle, e o lugar que julguei mais apropriado foi dentro do método `update`.

No código 7, utilizei uma variável nova, de nome `lastDamageTime`, que é atualizada toda vez que o jogador sofre dano. Ela é utilizada para calcular se já passou tempo suficiente (nesse caso 1 segundo) para atualizar a barra de vida secundária, e assim atribuir o valor inicial de -1 à variavel.

```
1  --Player.lua
2  hero.damage = function (value)
3      hero.health = hero.health - value
4      hero.lastDamageTime = love.timer.getTime()
5  end
6
7  hero.update = function (dt)
8      ...
9      if hero.lastDamageTime > 0 then
10         if love.timer.getTime() > hero.lastDamageTime
               + 1 then
11             hero.lastDamageTime = -1
12             hero.backHealth = hero.health
13         end
14     end
15 end
```

Código 7: Barra de vida convencional.

7.2.3 Análise

Essa foi a primeira mecânica relacionada ao controle de tempo que implementei. Ficou claro para mim que nesse ponto, a programação reativa tem muitas facilidades para oferecer. Alguns pontos vantajosos que pode-se notar já nessa simples mecânica são a redução da quantidade de variáveis necessárias e código reduzido, mais compreensível e com maior facilidade de ser estendido. Além disso, pude trocar com facilidade a maneira como queria que a barra de vida reagisse ao tempo, apenas trocando o método utilizado. Se fosse fazer essa mudança no convencional, teria sido um retrabalho pois provavelmente teria criado uma estrutura para implementar algo similar ao método `delay`.

7.3 Projétil

A segunda mecânica escolhida foi a de projétil. Nela, implementei um sistema de *object pooling* [17] básico, que toda vez que o jogador aperta a barra de

[17] https://gameprogrammingpatterns.com/object-pool.html

espaço, o personagem atira um projétil caso haja algum para atirar.

7.3.1 Rx

Para a inicialização dos projéteis, utilizei o *Observable "fromRange"* (código 8), que funciona de maneira semelhante ao *for* de Lua, no entanto, por ser um *Observable*, conta com as características que os métodos de *ReactiveX* tem para oferecer.

```
1 --Player.lua
2 hero.shots = Shot.Init()
3 rx.Observable.fromRange(1, 5)
4     :subscribe(function ()
5         Shot.Create()
6     end)
```

Código 8: Inicialização de projéteis em Rx.

O método `shoot` é chamado a partir da captura de input mencionado na seção 7.1. Para atirar o projétil, é necessário buscar o primeiro disponível no *object pool*. Como mostrado no código 9, isso foi feito com o *Observable "fromTable"*, que produz valores a partir de uma tabela de Lua. A partir disso, são filtrados os projéteis que não estão sendo utilizados no momento. Na linha 9, o método `first` retorna um *Observable* cujo único valor produzido é o primeiro dos valores filtrados previamente.

```lua
--Player.lua
function shoot()
    rx.Observable.fromTable(hero.shots, pairs, false)
        --filtra projeteis nao utilizados
        :filter(function(shot)
            return not shot.fired
        end)
        --seleciona o primeiro
        :first()
        :subscribe(function(shot)
            ...
            --marca como usado
            shot.fired = true
            ...
        end)
end
```

Código 9: Método `shoot` em Rx.

O projétil do inimigo foi implementado de maneira semelhante em relação a inicialização e o *object pooling*. No entanto, o que o faz atirar é um *scheduler* (código 10). Foi utilizado o *CooperativeScheduler*, que permite gerenciar *Observables* usando corrotinas, que respeitam um relógio virtual que deve ser atualizado manualmente. A cada chamada para `coroutine.yield` a execução do método "dorme" durante o tempo passado como argumento, posteriormente retornando na instrução seguinte.

```
1 --Enemies.lua
2 scheduler:schedule(function()
3         while true and enemy.alive do
4             enemyShoot()
5             coroutine.yield(math.random(.5,2))
6         end
7     end)
```

Código 10: Ativação do projétil inimigo em Rx.

7.3.2 Convencional

Como mencionado na subseção anterior, o *Observable "fromRange"* é parecido com o *for* de Lua, por isso, utilizei o segundo para a inicialização dos projéteis sem programação reativa.

```
1 --Player.lua
2 hero.shots = Shot.Init()
3 for i=1,5 do
4     Shot.Create()
5 end
```

Código 11: Inicialização de projéteis convencional.

No método de atirar o projétil, para continuar com a lógica de *object pooling*, utilizei um *for* para percorrer a tabela de projéteis. No entanto, tive que filtrá-los com um *if* convencional e, para garantir que só o primeiro fosse escolhido, utilizei o `break` para encerrar o *loop*.

```
1  --Player.lua
2  hero.shoot = function ()
3      for k, shot in pairs(hero.shots) do
4          if not shot.fired then
5              ...
6              shot.fired = true
7              ...
8              break
9          end
10     end
11 end
```

Código 12: Método `shoot` convencional.

Para o projétil inimigo, não tinha mais como usar o *scheduler*, então novamente tive que controlar o tempo manualmente dentro do método `update`.

Criei duas novas variáveis, `lastShotTime` para guardar o tempo em que o último projétil foi atirado e `nextShotTimeInterval` para guardar o intervalo para o próximo projétil ser atirado.

```
1  --Enemies.lua
2  enemy.update = function (dt)
3      if enemy.alive then
4          local curTime = love.timer.getTime()
5          --calcula se ja passou tempo suficiente
6          if curTime > enemy.lastShotTime +
               enemy.nextShotTimeInterval then
7              enemy.lastShotTime = curTime
8              enemy.enemyShoot()
9              enemy.nextShotTimeInterval =
                   math.random(.5,2)
10         end
11     end
12 end
```

Código 13: Ativação do projétil inimigo convencional.

7.3.3 Análise

As diferenças entre inicialização de projétil não apresentaram pontos significativos para serem discutidos. Já o controle do projétil inimigo, por trabalhar com tempo, foi novamente acompanhado de uma mudança significativa. O código ficou mais difícil de ser compreendido inicialmente e tomou um tamanho maior do que comparado à outra versão.

7.4 Alerta de perigo

Tornar a experiência justa é importante em muitos jogos, para isso alguns implementam um sistema de alerta de perigo. No projeto em questão, os inimigos podem atirar em qualquer momento, de onde quer que estejam. Portanto, é possível ocorrer uma situação em que o jogador, andando pelo mapa, seja atingido repentinamente por um projétil inimigo cuja posição só foi revelada ao entrar na tela. Então, a proposta é ter um tipo de aviso ou alerta para indicar que um projétil está chegando perto, mesmo se não visível ainda, para que o jogador tenha um tempo de resposta adequado.

7.4.1 Rx

Na linha 2 do código 14 criei um *Subject* que irá emitir a posição de cada projétil em uma certa frequência, determinada por um *scheduler*. Nas linhas 25-27 mostro, através do método `onNext`, como estão sendo inseridos os valores das posições. Inicializei, então, o *Observable* `enemyShotsAlertRange` na linha 3, que filtra as posições dos projéteis próximos à tela. A partir disso, nas linhas 8 e 9 são atualizadas a cor e posição do alerta para ser renderizado. Na linha 12, utilizei novamente o método `debounce` para trocar a cor de maneira que o alerta não apareça na tela enquanto não houver novos perigos iminentes.

```
1  --Enemies.lua
2  enemiesShotsPos = rx.Subject.create()
3  enemyShotsAlertRange =
       enemiesShotsPos:filter(function(pos)
4          return isCloseToScreen(pos)
5      end)
6
7  enemyShotsAlertRange:subscribe(function (pos)
8      alertaPerigo.cor = {1,0,0,1}
9      alertaPerigo.y = pos[2]
10 end)
11
12 enemyShotsAlertRange:debounce(.2, scheduler)
13     :subscribe(function (pos)
14         alertaPerigo.cor = {0,0,0,0}
15     end)
16
17 -- Atualiza posicao dos tiros
18 scheduler:schedule(function()
19     while true and enemy.alive do
20         rx.Observable.fromTable(enemy.shots, pairs,
               false)
21             :filter(function(shot)
22                 return shot.fired
23             end)
24             :subscribe(function(shot)
25                 enemiesShotsPos:onNext({
26                     shot.body:getPosition()
27                 })
28             end)
29         coroutine.yield(.3)
30     end
31 end)
```

Código 14: Alerta de perigo em Rx

7.4.2 Convencional

Nessa mecânica, precisei criar duas novas variáveis para controlar os intervalos de tempo. Caso a condição seja satisfeita, os projéteis são percorridos para conferir se a posição de cada um está próxima à tela.

```
1  --Enemies.lua
2  enemy.update = function (dt)
3      if enemy.alive then
4          -- Calculate dangerAlert
5          if curTime > enemy.lastDangerAlertTime +
               enemy.dangerAlertTimeInterval then
6              enemy.lastDangerAlertTime = curTime
7              for _, shot in pairs(enemy.shots) do
8                  local posX, posY =
                       shot.body:getPosition()
9                  if isCloseToScreen(posX, posY) then
10                     enemy.alertaPerigo.cor = {1,0,0,1}
11                     enemy.alertaPerigo.y = posY
12                 end
13             end
14         end
15
16         -- Reset alert
17         if curTime > enemy.lastDangerResetTime +
               enemy.dangerTimeMax then
18             enemy.lastDangerResetTime = curTime
19             enemy.alertaPerigo.cor = {0,0,0,0}
20         end
21     end
22 end
```

Código 15: Alerta de perigo convencional

7.4.3 Análise

É possível ver no código convencional que o mesmo ganhou um nível de complexidade em relação ao reativo. O principal fator foi o controle de tempo.

No entanto, no Rx, queria precisar somente reagir aos valores de posição de cada projétil, mas não consegui evitar de percorrer todos eles para poder saber suas posições, então nesse sentido não ficou muito distante do que foi feito no modo convencional.

7.5 Campo de visão/detecção

Muitos jogos do estilo *stealth*[18] implementam mecânicas de campo de visão ou similares, que definem áreas onde inimigos detectam ou não os jogadores. Esse campo pode ser estático ou dinâmico. Normalmente o jogador precisa passar por essa área sem ser percebido. Ele pode fazer isso desviando da mesma, apertando alguma tecla para se abaixar ou então se escondendo dentro de algum objeto, como por exemplo um armário, para ficar seguro enquanto o inimigo passa por ele.

Para a implementação, o campo de visão do inimigo foi dividido em três estados, cada um representado por uma cor para que possa ser identificado visualmente durante o jogo:

- **Alerta (laranja)**: o jogador está fora do campo de visão.
- **Perigo (vermelho)**: o jogador está dentro do campo de visão e não está escondido.
- **Seguro (verde)**: o jogador está dentro do campo de visão, porém escondido.

7.5.1 Rx

O código 16 trata a colisão do jogador com o campo de visão. Os *Observables* `playerEnter` e `playerExit` foram omitidos por serem códigos simples de filtragem a partir do evento de colisão interno da *engine*. Nas linhas 3 e 7 utilizei o método `map`, que retorna um novo *Observable* cujos valores passam por uma função de mapeamento antes de serem emitidos. Nesse caso, são transformados em uma tabela que contém uma *string* para informar o estado e uma tabela que

[18] https://pt.wikipedia.org/wiki/Jogo_eletrônico_de_stealth

representa o campos de visão do inimigo. Além disso, é feita a troca de cor do campo de visão dependendo do estado da colisão nas linhas 10-15.

```
1  --CollisionManager.lua
2  enterRange = playerEnter
3      :map(function (player, enemyRange)
4          return {state = "enter", enemyRange =
               enemyRange}
5      end)
6  exitRange = playerExit
7      :map(function (player, enemyRange)
8          return {state = "exit", enemyRange =
               enemyRange}
9      end)
10
11 enterRange:subscribe(function (info)
12     info.enemyRange.color =
           info.enemyRange.dangerColor
13 end)
14 exitRange:subscribe(function (info)
15     info.enemyRange.color =
           info.enemyRange.outRangeColor
16 end)
```

Código 16: Campo de visão/detecção Rx 1

O mapeamento também é feito no código 17, porém para o *Observable* da tecla “C”. Os *Observables* `cKeyPressed` e `cKeyReleased` foram omitidos por serem códigos simples de filtragem a partir do evento de input interno da *engine*.

```
1 --CollisionManager.lua
2 cPressed = cKeyPressed:map(function () return
    "pressed" end)
3 cReleased = cKeyReleased:map(function () return "not
    pressed" end)
```

Código 17: Campo de visão/detecção Rx 2

No código 18 utilizo novos métodos da biblioteca RxLua. O primeiro que usei deles foi o `merge`, nas linhas 2 e 3. Ele retorna um novo *Observable* que combina as emissões de todos os outros passados como argumento com o próprio *Observable* em que foi chamado. Isso é feito com os *Observables* de colisão e da tecla “C”.

Em seguida, chamo o método `combineLatest` em cima desses dois novos *Observables*. Com isso é retornado um novo *Observable* que combina a última emissão de cada um dos que foram passados como argumento e do próprio *Observable* em que foi chamado. Ou seja, sempre que um deles emitir um valor, `combineLatest` vai emitir um valor novo a partir do último valor emitido de cada *Observable*, de acordo com a função recebida.

```
1 --CollisionManager.lua
2 rangeState = enterRange:merge(exitRange)
3 cPressState = cPressed:merge(cReleased)
4
5 heroRangeState = rangeState
6     :combineLatest(cPressState, function (a, b)
7         return a,b
8     end)
```

Código 18: Campo de visão/detecção Rx 3

Por último, no código 19 é tratado o resultado do *Observable* retornado por `combineLatest`. Caso o último estado da colisão for *“enter”* e da tecla “C” for *“pressed”*, então o jogador está seguro. Por outro lado, se na mesma situação de colisão o estado da tecla “C” for *“not pressed”*, então o jogador está em perigo. Nas linhas 5-10 a cor do campo de visão é atualizada de acordo.

```
1  --CollisionManager.lua
2  heroSafe = heroRangeState:filter(function(a,b) return
       a.state == "enter" and b == "pressed" end)
3  heroNotSafe = heroRangeState:filter(function(a,b)
       return a.state == "enter" and b == "not pressed"
       end)
4
5  heroSafe:subscribe(function(a,b)
6      a.enemyRange.color = a.enemyRange.safeColor
7  end)
8  heroNotSafe:subscribe(function(a,b)
9      a.enemyRange.color = a.enemyRange.dangerColor
10 end)
```

Código 19: Campo de visão/detecção Rx 4

7.5.2 Convencional

Nessa versão, mudei um pouco a maneira como verificava se o jogador estava dentro de um *enemy range*. Criei uma nova variável `inEnemyRange` no jogador que é atualizada quando ocorre colisão entre o jogador e o campo de visão, como mostra-do no código 20.

```
1  --CollisionManager.lua
2  -- entrada da colisao entre jogador e campo de visao
3  enemyRange.color = enemyRange.dangerColor
4  hero.inEnemyRange = enemyRange
5
6  -- saida da colisao entre jogador e campo de visao
7  enemyRange.color = enemyRange.outRangeColor
8  hero.inEnemyRange = nil
```

Código 20: Campo de visão/detecção convencional 1

Ela é então utilizada pelo módulo do jogador durante os eventos de `keypressed` e `keyreleased` (código 21).

```
1  --Player.lua
2  hero.keypressed = function (key)
3      if key == "c" and hero.inEnemyRange ~= nil then
4          hero.inEnemyRange.color =
               hero.inEnemyRange.safeColor
5      end
6  end
7
8  hero.keyreleased = function (key)
9      if key == "c" and hero.inEnemyRange ~= nil then
10         hero.inEnemyRange.color =
               hero.inEnemyRange.dangerColor
11     end
12 end
```

Código 21: Campo de visão/detecção convencional 2

7.5.3 Análise

A primeira coisa que eu percebi foi um *bug* na versão convencional. A mecânica funciona bem para o caso em que o jogador pressiona a tecla “C” quando já está dentro do campo de visão, mas não funciona se ele entra no campo já apertando a tecla. Para corrigir isso seria necessário manter o estado da tecla, toda vez que ela apertada e solta. Assim, seria possível, ao entrar no campo de visão, determinar se o jogador estaria seguro ou não.

Esse foi um bom exemplo de como a programação reativa pode evitar a geração de *bugs*, pois o código fica mais direto e mais fácil de compreender.

7.6 Plataforma móvel

No gênero de jogo de plataforma, é muito comum encontrar plataformas móveis, que são elementos que compõem o mapa do jogo e permitem aprimorar o *level design*[19]. São objetos que ficam em movimento constante (normalmente entre dois pontos) em que o jogador pode subir em cima de forma a ser transportado de um ponto a outro no mapa.

[19] https://pt.wikipedia.org/wiki/Level_design

7.6.1 Rx

Para essa mecânica, a solução foi simples. O movimento da plataforma depende de uma velocidade que é atualizada sempre que a plataforma se encontra fora dos dois pontos definidos como o trajeto a ser percorrido. Isso é feito a partir do método `filter` para cada valor do *Observable* `love.update`. Precisei de uma *flag* `canInvert` para garantir que só mudaria a direção uma vez ao sair do trajeto. Caso contrário, a plataforma ficaria num *loop* mudando de direção continuamente quando saísse do trajeto pela primeira vez.

```
1  --MovingPlats.lua
2  love.update
3      :filter(function()
4          return plat.insidePath()
5      end)
6      :subscribe(function()
7          plat.canInvert = true
8      end)
9
10 love.update
11     :filter(function()
12         return not plat.insidePath() and
               plat.canInvert
13     end)
14     :subscribe(function()
15         plat.velocity = plat.velocity * -1
16         plat.body:setLinearVelocity(plat.vertical and
               0 or plat.velocity, plat.vertical and
               plat.velocity or 0)
17         plat.canInvert = false
18     end)
```

Código 22: Plataforma móvel em Rx

7.6.2 Convencional

De maneira similar à implementação em Rx, o movimento da plataforma é controlado dentro do método `update`, verificando se a mesma se encontra dentro ou fora do trajeto.

```
1  --MovingPlats.lua
2  plat.update = function()
3      if plat.insidePath() then
4          plat.canInvert = true
5      end
6
7      if not plat.insidePath() and plat.canInvert then
8          plat.velocity = plat.velocity * -1
9          plat.body:setLinearVelocity(plat.vertical and
               0 or plat.velocity, plat.vertical and
               plat.velocity or 0)
10         plat.canInvert = false
11     end
12 end
```

Código 23: Plataforma móvel convencional

7.6.3 Análise

A diferença entre as implementações ficou bem pequena, porque se encaixa no caso em que é necessária a mudança de estados dentro do *loop* principal do jogo, somente verificando valor de condicionais. Os filtros foram substituídos por condicionais e o `subscribe` pelas ações dentro das condicionais.

7.7 *Falling platform*

Assim como a plataforma móvel, *falling platform* é um elemento muito presente em jogos de plataforma. São plataformas que o jogador usa como apoio para chegar em determinado local, mas que no entanto quebram ou caem após um curto tempo a partir do momento em que o jogador sobe nela. Em muitos

jogos, e é também o caso nesse projeto, a plataforma pode voltar ao seu estado inicial após alguns segundos depois de cair. Dessa maneira, o jogador pode voltar atrás para poder subir de novo.

7.7.1 Rx

Para essa mecânica, precisava de alguma maneira inserir uma chamada de função ou ação no meio da sequência de transformações dos *Observables*, para executar código antes de algumas transformações. Já tinha observado na lista de operadores de Rx e em alguns exemplos de outras linguagens que existia um método chamado `Do`. No entanto, esse método não se encontra presente na biblioteca *RxLua*. Então resolvi implementá-lo, com o nome de `execute`. O resultado mostrou que a biblioteca é facilmente extensível.

Ao ocorrer uma colisão entre a *falling platform* e o jogador, um novo valor é gerado para o *Subject* `touchedPlayer` pelo método `onNext`.

```
1 --CollisionManager.lua
2 --colisao entre plataforma movel e jogador
3 plat.touchedPlayer:onNext(true)
```

Foi necessário uma variável indicando se o jogador está tocando na plataforma para controlar o fluxo de dados do *Subject* `touchedPlayer`, que emite um novo valor toda vez que o jogador colide com a plataforma.

```
1 --FallingPlats.lua
2 plat.playerTouching = false
3 plat.timeToFall = 1
4 plat.touchedPlayer = rx.BehaviorSubject.create()
```

O funcionamento dessa mecânica encaixou muito bem com o paradigma reativo pois é uma sequência de transformação bem direta. A partir do *Subject* `touchedPlayer`, filtro os valores para só serem emitidos quando o jogador subir na plataforma, e não emitir enquanto está pisando na mesma. Após isso, marco a *flag* `playerTouching`, utilizando o método `execute` que implementei, para impedir o processamento de novos valores até o jogador sair e entrar de novo. Utilizo

então um `delay` para dar um tempo ao jogador antes que a plataforma de fato caia. Passado o tempo, novamente utilizando o método `execute` eu mudo o valor da velocidade da plataforma para que ela comece a cair, e mais uma vez utilizo um `delay` para que ela possa voltar a posição original após alguns segundos.

```
1  --FallingPlats.lua
2  plat.touchedPlayer
3      :filter(function()
4          return plat.playerTouching == false
5      end)
6      :execute(function()
7          plat.playerTouching = true
8      end)
9      :delay(plat.timeToFall, scheduler)
10     :execute(function()
11         plat.body:setLinearVelocity(0, plat.velocity)
12     end)
13     :delay(plat.timeToFall*3, scheduler)
14     :subscribe(function()
15         plat.playerTouching = false
16         plat.reset()
17     end)
```

Código 24: *Falling platform* em Rx

7.7.2 Convencional

Ao ocorrer uma colisão entre a *falling platform* e o jogador, eu chamo o método `touchedPlayer` da plataforma.

```
1  --CollisionManager.lua
2  --colisao entre plataforma movel e jogador
3  plat.touchedPlayer()
```

Para o modo convencional, tive que criar uma variável para controlar o estado da plataforma. O estado 0 é o inicial, a plataforma se encontra parada até o jogador

subir nela. Ao acontecer isso, é mudado para o estado 1, em que começa a contar o tempo antes da plataforma cair. Quando o tempo acaba, a plataforma começa a cair, passando para o estado 2, que contabiliza o tempo para que a plataforma retorne ao estado inicial.

Foi necessário também criar a variável `previousTime` para poder controlar a mudança de estado.

```
1 --FallingPlats.lua
2 plat.previousTime = -1
3 plat.timeToFall = 1
4 plat.state = 0
```

O método `touchedPlayer` controla a mudança de estado 0 para 1, ou seja, quando o jogador encosta na plataforma, passa do estado em que está parada para o estado em que está prestes a cair.

```
1 --FallingPlats.lua
2 plat.touchedPlayer = function()
3     if plat.state == 0 then
4         plat.previousTime = love.timer.getTime()
5         plat.state = 1
6     end
7 end
```

Para controlar a mudança de estados que depende da contagem de tempo, utilizei o método `update`. A cada chamada pego o valor do tempo atual e comparo com o último valor guardado, e dependendo do resultado eu mudo de estado.

```
1  --FallingPlats.lua
2  plat.update = function(dt)
3      if plat.state == 1 then
4          local curTime = love.timer.getTime()
5          if curTime > plat.previousTime +
               plat.timeToFall then
6              plat.body:setLinearVelocity(0,
                   plat.velocity)
7              plat.previousTime = curTime
8              plat.state = 2
9          end
10     elseif plat.state == 2 then
11         local curTime = love.timer.getTime()
12         if curTime > plat.previousTime +
               plat.timeToFall*3 then
13             plat.body:setLinearVelocity(0, 0)
14             plat.body:setPosition(plat.initX,
                   plat.initY)
15             plat.state = 0
16         end
17     end
18 end
```

Código 25: *Falling platform* convencional

7.7.3 Análise

Da mesma maneira como outras mecânicas relacionadas ao controle de tempo, tive que criar variáveis extras e gerenciar esse controle no caso convencional. Além disso, tive que criar uma estrutura de estados, que apesar de simples, aumenta a complexidade do código. Enquanto que no modo Rx, o fluxo de dados funcionou de maneira muito direta, podendo ver no próprio código o que vai acontecer durante a execução da mecânica.

7.8 Escudo (coletável)

Itens coletáveis estão presentes em diversos gêneros de jogos. Podem trazer efeitos tanto positivos quanto negativos, e podem ser temporários ou permanentes. O escudo, nesse projeto, é um item coletável que permite o jogador bloquear projéteis inimigos para evitar sofrer dano. No entanto, para ativar o efeito do escudo, o jogador precisa pressionar a tecla de ativação em um intervalo de tempo curto com a colisão entre o projétil inimigo e o escudo.

7.8.1 Rx

Precisava de alguma maneira então calcular o intervalo de tempo entre duas emissões de *Observables* diferentes, e novamente lembrei de um operador listado de Rx que não estava presente na biblioteca que utilizei. Então resolvi implementar o método `TimeStamp` que, para cada valor emitido pelo *Observable* original, emite o tempo em que o mesmo foi emitido, juntamente com os valores originais.

Ao jogador colidir com o escudo, mando como valor novo para o *Subject* `item` do `hero` o próprio escudo. Além disso, mando para o escudo o valor `true` para o *Subject* `touchedPlayer`. Na colisão entre escudo e projétil inimigo, mando para o escudo o próprio projétil como valor para o *Subject* `touchedShot`.

```
1 --CollisionManager.lua
2 --colisao entre jogador e escudo
3 hero.item:onNext(shield)
4 shield.touchedPlayer:onNext(true)
5
6 --colisao entre escudo e projetil inimigo
7 shield.touchedShot:onNext(enemyShot)
```

No módulo do jogador, para enviar as teclas pressionadas pelo jogador para o módulo do escudo, combino as emissões do *Observable* `love.keypressed` com o *Subject* `hero.item` com o operador `combineLatest` já utilizado previamente no código 18. No entanto, isso fará com que assim que o jogador coletar um item (no caso o escudo), será emitido um novo valor para o *Subject* `playerPressed`. Esse problema será corrigido nos próximos códigos.

```
1  --Player.lua
2  love.keypressed
3      :combineLatest(hero.item, function (key, item)
4          return key,item
5      end)
6      :subscribe(function (key, item)
7          item.playerPressed:onNext(key)
8      end)
```

No código 26 primeiro defino, com o *Subject* `touchedPlayer`, que quando o jogador encostar no escudo, ele irá aumentar seu raio para que fique em volta do jogador (maior do que ele). Para corrigir o problema mencionado anteriormente, utilizei um método novo na linha 12 chamado `skip`, que "pula" emissões do *Observable* original. Nesse caso, ignoro a primeira tecla emitida pois não nos interessa, já que foi pressionada antes do jogador adquirir o escudo. Além disso, filtro para apenas receber a tecla "f". Para finalizar, utilizo o método `TimeStamp` que implementei para saber o momento em que a tecla foi pressionada.

Na linha 18 novamente utilizo o método `TimeStamp` para saber o tempo em que o escudo encostou no projétil inimigo. Após isso, combino esses dois *Observables* para poder comparar o tempo entre as duas últimas emissões de cada um. E então eu filtro para que somente emissões com menos de 0,5 segundos entre si possam ativar o efeito do escudo.

```
1  --Shield.lua
2  shield.touchedPlayer = rx.BehaviorSubject.create()
3  shield.playerPressed = rx.BehaviorSubject.create()
4  shield.touchedShot = rx.BehaviorSubject.create()
5
6  shield.touchedPlayer
7      :subscribe(function()
8          shield.shape:setRadius(shield.initRadius)
9      end)
10
11 local activatedTimeStamp = shield.playerPressed
12     :skip(1)
13     :filter(function(key)
14         return key == "f"
15     end)
16     :Timestamp(scheduler)
17
18 local touchedShotTimeStamp =
       shield.touchedShot:Timestamp(scheduler)
19
20 touchedShotTimeStamp
21     :combineLatest(activatedTimeStamp, function
           (shotInfo, activatedInfo)
22         return shotInfo, activatedInfo
23     end)
24     :filter(function(shotInfo, activatedInfo)
25         return math.abs(shotInfo.time -
               activatedInfo.time) < 0.5
26     end)
27     :subscribe(function(shotInfo, activatedInfo)
28         shotInfo.other.reset()
29     end)
```

Código 26: Escudo em Rx

7.8.2 Convencional

Na colisão entre jogador e escudo, eu atribuo o escudo à variável `item` do jogador e chamo o método `touchedPlayer` do escudo. Além disso, na colisão entre escudo e projétil inimigo, eu chamo o método `touchedShot` do módulo do escudo, passando o projétil como parâmetro.

```
1 --CollisionManager.lua
2 --colisao entre jogador e escudo
3 hero.item = shield
4 shield.touchedPlayer()
5
6 --colisao entre escudo e projetil inimigo
7 shield.touchedShot(enemyShot)
```

No módulo do jogador, apenas passo adiante o evento de `keypressed` chamando o método `playerPressed` do item, no caso o escudo, caso o jogador já tenha coletado.

```
1 --Player.lua
2 hero.item = rx.BehaviorSubject.create()
3 hero.keypressed = function (key)
4     if hero.item ~= nil then
5         hero.item.playerPressed(key)
6     end
7 end
```

No código 27 foi necessária a criação de três variáveis, duas para controlar o tempo em que ocorreram os eventos de tecla pressionada e de colisão entre escudo e projétil e outra para guardar o objeto do projétil. Cada uma dessas variáveis é atualizada nos seus respectivos métodos, chamados pelos módulos do jogador e de colisão mostrados anteriormente.

E a partir desses métodos chamo uma função chamada `checkActivation` que irá verificar se a ativação do efeito do escudo é válida. Para isso, faço comparações para primeiro determinar se as variáveis possuem valores válidos. Em seguida,

verifico se a diferença de tempo entre as duas variáveis é menor do que 0,5 segundos.

```
1  --Shield.lua
2  shield.activatedTimeStamp = -1
3  shield.touchedShotTimeStamp = -1
4  shield.touchedShotObj = nil
5
6  shield.touchedPlayer = function()
7      shield.shape:setRadius(shield.initRadius)
8  end
9
10 shield.playerPressed = function(key)
11     if key == "f" then
12         shield.activatedTimeStamp =
               love.timer.getTime()
13     end
14
15     checkActivation()
16 end
17
18 shield.touchedShot = function(shot)
19     shield.touchedShotTimeStamp = love.timer.getTime()
20     shield.touchedShotObj = shot
21
22     checkActivation()
23 end
24
25 local function checkActivation()
26     if shield.activatedTimeStamp > 0 and
27     shield.touchedShotTimeStamp > 0 and
28     shield.touchedShotObj ~= nil and
29     math.abs(shield.activatedTimeStamp -
          shield.touchedShotTimeStamp) < 0.5 then
30         shield.touchedShotObj.reset()
31         shield.touchedShotObj = nil
32     end
33 end
```

Código 27: Escudo convencional

7.8.3 Análise

Novamente tratando-se de uma mecânica que envolve o controle de tempo, a implementação em Rx se sobressai pela facilidade de manipulação. Um outro ponto importante é o fato de na implementação convencional ter precisado de um código extra, separado, para tratar a verificação de ativação do escudo, enquanto que utilizando *Observables* em Rx foi possível combinar o fluxo de dados em um lugar só.

7.9 *Quick Time events*

Quick Time event é uma mecânica que requer do jogador apertar teclas específicas em um intervalo de tempo curto, em que o erro leva a uma consequência ruim para o jogador. Para esse jogo, implementei um inimigo que para ser derrotado requer que o jogador entre na área de alcance dele e pressione corretamente uma sequência de teclas, com um intervalo de tempo de no máximo 0,5 segundos entre cada uma. Caso erre a tecla ou demore muito para acertar a seguinte, o jogador sofre dano e a sequência começa de novo.

7.9.1 Rx

Para implementar essa mecânica em Rx, busquei novamente na lista de operadores existentes e encontrei um chamado `TimeInterval`, que também não estava presente na RxLua, então esse foi o terceiro método que implementei na biblioteca. Esse operador retorna um *Observable* que emite os valores do *Observable* original, no entanto, junto com esses valores, também emite o intervalo de tempo que se passou desde a última emissão. Dessa maneira, é possível ver se o jogador passou do tempo permitido para completar a sequência.

Ao jogador colidir com a área de alcance do inimigo, envio um novo valor para o *Subject* `quickTimeRange` do objeto do jogador, para que ele tenha acesso ao objeto da área de alcance. Além disso, fiz o oposto com o objeto da área de alcance, que recebe o jogador como novo valor em seu *Subject* `playerInRange`, para que possa reagir quando o jogador de fato entrar no alcance do inimigo. Na saída da colisão, ou seja, quando os objetos deixam de se tocar, envio nil para os respectivos *Subjects*.

```
1  --CollisionManager.lua
2  --entrada da colisao entre jogador e area do inimigo
3  hero.quickTimeRange:onNext(quickTimeRange)
4  quickTimeRange.playerInRange:onNext(hero)
5
6  --saida da colisao entre jogador e area do inimigo
7  hero.quickTimeRange:onNext(nil)
8  quickTimeRange.playerInRange:onNext(nil)
```

No módulo do jogador, simplesmente passo adiante o evento de `keypressed` para o inimigo, caso o jogador esteja dentro do alcance dele.

```
1  --Player.lua
2  hero.quickTimeRange = rx.BehaviorSubject.create()
3
4  love.keypressed
5      :filter(function (key)
6          return hero.quickTimeRange:getValue() ~= nil
7      end)
8      :subscribe(function (key, item)
9          hero.quickTimeRange
10             :getValue().playerPressed:onNext(key)
11     end)
```

No módulo do inimigo, primeiramente defino a sequência aceita pelo mesmo. Também defino três *Subjects*: `playerPressed`, `playerInRange` e `sequence`. Estes recebem, respectivamente, os valores de *input* do jogador, objeto do jogador dentro da área de alcance e sequencia atual. Implementei a primeira resposta à esses eventos na linha 11 do código a seguir, que se refere ao jogador entrar dentro da área de alcance, e nesse caso reinicio a sequência do inimigo.

```
1  --Enemies.lua
2  enemy.sequence = {
3      "down",
4      "up",
5      "left",
6      "right"
7  }
8
9  quickTimeRange.playerPressed =
       rx.BehaviorSubject.create()
10 quickTimeRange.playerInRange =
       rx.BehaviorSubject.create()
11 quickTimeRange.playerInRange
12     :filter(function(value)
13         return value ~= nil
14     end)
15     :subscribe(function()
16         enemy.resetSequence()
17     end)
18 quickTimeRange.sequence = rx.BehaviorSubject.create()
```

A partir desse ponto, criei vários *Observables* a partir dos *Subjects* definidos anteriormente e a partir dos novos gerados também. O primeiro deles foi o `trySequence`, que filtra se a tecla apertada pelo jogador e o contador do passo da sequência são válidos. E para cada valor emitido por esse *Observable* será emitido o próximo elemento da sequência para o *Subject* `sequence`, como mostrado nas linhas 7-11 do código a seguir .

```
1  --Enemies.lua
2  local trySequence = quickTimeRange.playerPressed
3      :filter(function(key)
4          return acceptedKeys[key] and
               enemy.sequenceTries > 0
5      end)
6
7  trySequence
8      :subscribe(function()
9          quickTimeRange.sequence
10             :onNext(enemy.sequence[enemy.sequenceTries])
11     end)
```

Para verificar se o jogador acertou ou não a tecla da sequência e reagir de acordo, utilizei o método `zip`, que combina as emissões dos *Observables* recebidos por parâmetro - e o próprio *Observable* em que foi chamado - pelos índices das emissões. Dessa maneira, o par de valores de `trySequence` e de `quickTimeRange.sequence` estarão corretamente alinhados para ser feita a verificação. E para poder tratar os dois eventos (erro e acerto) separadamente, utilizei o método `partition`. O operador recebe uma função para avaliar uma condição, e então retorna dois *Observables*, um que emite os valores que respeitam a condição e outro para os outros valores. Dessa maneira, criei os *Observables* `match` e `wrong`, para quando o jogador acertar e errar, respectivamente.

```
1  --Enemies.lua
2  local match, wrong = trySequence
3      :zip(quickTimeRange.sequence)
4      :partition(function(try, answer)
5          return try == answer
6      end)
```

Código 28: Quick Time Event em Rx 1

A partir do *Observable* `match`, eu precisava checar se o tempo passado entre as emissões foi válido, e assim utilizei o método `TimeInterval` que implementei

para assim poder filtrar os valores que eram maiores e menores do que 0,5, atribuindo os novos *Observables* às variáveis `miss` e `onTime`, respectivamente. A condição do filtro também incluiu avaliar em que número da sequência o jogador estava, pois se for o primeiro, o intervalo de tempo pode ser maior do que 0,5 segundos.

```
1  --Enemies.lua
2  local miss = match
3      :TimeInterval(scheduler)
4      :filter(function(dt, try, answer)
5          return dt > 0.5 and enemy.sequenceTries > 1
6      end)
7
8  local onTime = match
9      :TimeInterval(scheduler)
10     :filter(function(dt, try, answer)
11         return dt < 0.5 or enemy.sequenceTries == 1
12     end)
```

Código 29: Quick Time Event em Rx 2

E finalmente, no código a seguir mostro a reação à emissão de valores para esses *Observables*. Primeiro utilizei o método `merge` para unificar as emissões dos *Observables* `miss` e `wrong` pois os dois resultam na mesma reação. Nessa reação, aplico um dano ao jogador, mudo a cor da área de alcance para dar um *feedback* visual para o jogador e marco o número da sequência como -1 para ficar inválido por um momento. Então utilizo o método `delay` para esperar 1 segundo e então reiniciar a sequência.

Para o *Observable* `onTime`, também realizo uma mudança de cor e incremento o contador da sequência. Em seguida, filtro para verificar se o jogador atingiu o final da sequência. Caso tenha completado, o inimigo é eliminado.

```
1  --Enemies.lua
2  miss
3      :merge(wrong)
4      :execute(function()
5          hero.health:onNext(hero.health:getValue() -
               10)
6          quickTimeRange.color =
               quickTimeRange.wrongColor
7          enemy.sequenceTries = -1
8      end)
9      :delay(1, scheduler)
10     :subscribe(function(try, step)
11         enemy.resetSequence()
12     end)
13
14 onTime
15     :execute(function()
16         quickTimeRange.color =
               quickTimeRange.matchColor
17         enemy.sequenceTries = enemy.sequenceTries + 1
18     end)
19     :filter(function()
20         return enemy.sequenceTries ==
               #enemy.sequence+1
21     end)
22     :subscribe(function()
23         killEnemy(enemy)
24     end)
```

Código 30: Quick Time Event em Rx 3

7.9.2 Convencional

Implementei parecido com a maneira que fiz em Rx. Durante a colisão entre o jogador e o objeto do alcance do inimigo, atribuo à variável `quickTimeRange` do jogador o objeto do alcance e chamo um método do objeto do alcance para

informar que o jogador se encontra dentro dele.

```
1 --CollisionManager.lua
2 --entrada da colisao entre jogador e area do inimigo
3 hero.quickTimeRange = quickTimeRange
4 quickTimeRange.playerInRange()
5
6 --saida da colisao entre jogador e area do inimigo
7 hero.quickTimeRange = nil
```

No módulo do jogador, novamente passo adiante o evento de `keypressed` para o inimigo, caso o jogador esteja dentro do alcance dele, porém dessa vez chamando o método `playerPressed`.

```
1 --Player.lua
2 hero.keypressed = function (key)
3     if hero.quickTimeRange ~= nil then
4         hero.quickTimeRange.playerPressed(key)
5     end
6 end
```

Como no caso Rx, defini uma tabela com a sequência aceita pelo inimigo. Além disso, foram necessárias duas variáveis, `lastMatchTime`, para guardar o tempo da última vez em que o jogador acertou uma parte da sequência, e `timeToReset` para controlar o tempo de reiniciar a sequência após errar.

```
1  --Enemies.lua
2  enemy.sequence = {
3      "down",
4      "up",
5      "left",
6      "right"
7  }
8
9  quickTimeRange.lastMatchTime = love.timer.getTime()
10 quickTimeRange.timeToReset = 0
```

Para o controle de estado de erro e acerto da sequência, implementei o método `playerPressed` que é chamado toda vez em que o jogador pressiona um tecla. A partir disso, utilizei uma cadeia de condicionais para realizar cada mudança.

```
1  --Enemies.lua
2  quickTimeRange.playerPressed = function(key)
3      if acceptedKeys[key] and enemy.sequenceTries > 0
           then
4          if key == enemy.sequence[enemy.sequenceTries]
               then
5              --match
6              local dt = love.timer.getTime() -
                   quickTimeRange.lastMatchTime
7              if dt < 0.5 or enemy.sequenceTries == 1
                   then
8                  --onTime
9                  quickTimeRange.lastMatchTime =
                       love.timer.getTime()
10                 quickTimeRange.color =
                       quickTimeRange.matchColor
11                 enemy.sequenceTries =
                       enemy.sequenceTries + 1
12
13                 --finished successfully
14                 if enemy.sequenceTries ==
                       #enemy.sequence+1 then
15                     killEnemy(enemy)
16                 end
17             else
18                 --miss
19                 quickTimeRange.missWrong()
20             end
21         else
22             --wrong
23             quickTimeRange.missWrong()
24         end
25     end
26 end
```

Código 31: Quick Time Event convencional

Para tratar a entrada do jogador na área de alcance, implementei um método para reiniciar a sequência do inimigo que é chamado quando ocorre a colisão. Também implementei um método para tratar o erro do jogador, chamado no código anterior em duas situações. Por fim, o método `update` para controlar o tempo antes de reiniciar a sequência após um erro do jogador.

```
1  -- Functions
2  quickTimeRange.playerInRange = function()
3      enemy.resetSequence()
4  end
5
6  quickTimeRange.missWrong = function()
7      hero.damage(10)
8      quickTimeRange.color = quickTimeRange.wrongColor
9      enemy.sequenceTries = -1
10
11     quickTimeRange.timeToReset = 1
12     quickTimeRange.shouldReset = true
13 end
14
15 enemy.update = function (dt)
16     quickTimeRange.timeToReset =
           quickTimeRange.timeToReset - dt
17     if quickTimeRange.shouldReset and
           quickTimeRange.timeToReset < 0 then
18         enemy.resetSequence()
19         quickTimeRange.shouldReset = false
20     end
21 end
```

7.9.3 Análise

Além das vantagens de manipulação de tempo em Rx já mencionadas algumas vezes em outras mecânicas, nesse exemplo também pude notar que uma grande vantagem de Rx é a possibilidade de melhor separar o código. Enquanto

que no modo convencional tive que implementar uma cadeia de condicionais para tratar todos os casos em um método só, ou seja, em uma passagem de dados, no Rx pude separar esse tratamento em diversos *Observables*, mantendo o mesmo fluxo de dados.

7.10 *Boss*

Para finalizar o projeto, queria combinar algumas mecânicas já implementadas em um lugar só, com a ideia de tentar reaproveitar os *Observables* no caso de Rx e assim poder traçar uma análise da diferença para o convencional. Resolvi implementar então um *boss* (inimigo final), que é bastante comum em muitos gêneros de jogos para que o jogador tenha um desafio antes de terminar o jogo.

A ideia é que ele seja mais forte e difícil de derrotar do que os inimigos comuns. Por esse motivo, implementei ele de um jeito que sua vida é muito maior e que sua mecânica é na verdade uma mistura de todos os outros inimigos. O *boss* fica em um ciclo mudando de estado, em que cada um desses é uma mecânica de um inimigo já apresentado. No primeiro estágio ele atira projéteis, e pode ser atingido por projéteis também. No segundo, ele cria um campo de detecção, e só pode ser atingido por projéteis se o jogador estiver “seguro” dentro dele, que no caso desse jogo é apertando a tecla “C”. No terceiro e último estado, ele utiliza a mecânica de *quick time event*, e nesse caso para causar dano é preciso completar a sequência.

7.10.1 Rx

A minha ideia inicial para a implementação em Rx era de reutilizar os *Observables* criados para as mecânicas dos outros inimigos, mas isso não foi possível pois a criação deles envolvia objetos específicos de cada inimigo. No entanto, simplifiquei a implementação em alguns pontos.

Primeiro, criei um *Observable* para controlar a mecânica de atirar projéteis. Esse vai receber novos valores de acordo com a corrotina mostrada no código a seguir, seguindo o *scheduler*. Dessa maneira, qualquer inimigo interessado em um *scheduler* para atirar projéteis pode dar `subscribe` no *Observable* `shootScheduler` e reagir de acordo.

```
1  --Enemies.lua
2  -- Commom Observables/schedulers
3  Enemies.shootScheduler = rx.Subject.create()
4
5  scheduler:schedule(function()
6      coroutine.yield(1)
7      while true do
8          Enemies.shootScheduler:onNext()
9          coroutine.yield(math.random())
10     end
11 end)
```

No caso do inimigo normal, como agora todos vão respeitar o mesmo *scheduler*, resolvi filtrar com uma probabilidade de atirar para cada valor recebido.

```
1  --Enemies.lua
2  Enemies.shootScheduler
3      :filter(function()
4          return enemy.alive and math.random() > 0.3
5      end)
6      :subscribe(function()
7          enemyShoot(enemy.shots, {enemy.body:getX(),
               enemy.body:getY()})
8      end)
```

Para o *boss*, resolvi deixar sem probabilidade, no entanto ele atira quatro projéteis por vez, e somente se estiver no primeiro estágio. A vetor *pos* foi omitido para deixar o código mais legível.

```
1  --Enemies.lua
2  Enemies.shootScheduler
3      :filter(function()
4          return enemy.alive and enemy.state == 1
5      end)
6      :subscribe(function()
7          enemyShoot(enemy.shots, pos[1])
8          enemyShoot(enemy.shots, pos[2])
9          enemyShoot(enemy.shots, pos[3])
10         enemyShoot(enemy.shots, pos[4])
11     end)
```

Para as outras mecânicas, simplesmente criei métodos para retornar as tabelas correspondentes a cada objeto. No entanto, queria um resultado diferente para o *quick time*, pois enquanto o inimigo normal é derrotado ao jogador acertar a sequência, o *boss* somente sofre um pouco de dano. Então passo como parâmetro uma função que será chamada no caso de acerto.

```
1  --Enemies.lua
2  -- Area alcance visao
3  local enemyRange = createRange(enemy, 350, 500)
4
5  -- Area quicktime
6  local quickTimeRange = createQuickRange(enemy,
       scheduler, function()
7      enemy.damage(10)
8  end)
```

Para controlar a mudança de estado, utilizei uma corrotina a partir do *scheduler* que troca o estado a cada 5 cinco segundos.

```
1  --Enemies.lua
2  -- Change state
3  scheduler:schedule(function()
4      while true do
5          coroutine.yield(5)
6          local nextState = enemy.state + 1
7          enemy.state = nextState <= enemy.maxState and
               nextState or 1
8      end
9  end)
```

Em relação ao dano sofrido, criei um *Observable* que recebe valores novos sempre que um projétil colide com o *boss* e filtro ele para validar ou não o dano. Além disso, transformei a vida do *boss* em um *Subject*, para observar quando a mesma chegar em 0.

```
1  --Enemies.lua
2  enemy.shotHit
3      :filter(function()
4          return (enemy.state == 1) or (enemy.state ==
               2 and enemyRange.color ==
               enemyRange.safeColor)
5      end)
6      :subscribe(function()
7          enemy.damage(10)
8      end)
9
10 enemy.health
11     :filter(function(val)
12         return val <= 0
13     end)
14     :subscribe(function()
15         killEnemy(enemy)
16     end)
```

7.10.2 Convencional

O primeiro passo foi tentar reproduzir o *scheduler* de projétil. No entanto, não podendo utilizar *Observables* e *schedulers* tive que recorrer a chamar um método para cada inimigo dentro do `update` definido no módulo dos inimigos, que é por sua vez chamado no método `love.update` do módulo principal.

```
1  --Enemies.Lua
2  Enemies.lastShotTime = love.timer.getTime()
3  Enemies.nextShotTimeInterval = 1
4
5  Enemies.update = function(dt)
6      -- Shoots
7      local curTime = love.timer.getTime()
8      if curTime > Enemies.lastShotTime +
          Enemies.nextShotTimeInterval then
9          for k, enemy in pairs(Enemies.enemies) do
10             if enemy.alive and enemy.shootSchedule
                  then
11                 enemy.shootSchedule()
12             end
13         end
14         Enemies.lastShotTime = curTime
15         Enemies.nextShotTimeInterval = math.random()
16     end
17 end
```

Dentro do método, a implementação foi bem parecida, apenas trocando filtro por condicionais.

```
1  --Enemies.Lua
2  enemy.shootSchedule = function()
3      if math.random() > 0.3 then
4          enemyShoot(enemy.shots, {enemy.body:getX(),
               enemy.body:getY()})
5      end
6  end
7
8  enemy.shootSchedule = function()
9      if enemy.state == 1 then
10         enemyShoot(enemy.shots, pos[1])
11         enemyShoot(enemy.shots, pos[2])
12         enemyShoot(enemy.shots, pos[3])
13         enemyShoot(enemy.shots, pos[4])
14     end
15 end
```

Assim como em Rx, transformei as outras mecânicas em métodos para simplificar a criação dos objetos.

```
1  --Enemies.Lua
2  -- Area alcance visao
3  local enemyRange = createRange(enemy, 350, 500)
4
5  -- Area quicktime
6  local quickTimeRange = createQuickRange(enemy)
7  quickTimeRange.onMatch = function()
8      enemy.damage(10)
9  end
```

Para controlar a mudança de estados, mais uma vez recorri ao método `update`, para poder controlar a passagem de tempo.

```
1  --Enemies.Lua
2  enemy.update = function()
3      -- Change state
4      local curTime = love.timer.getTime()
5      if curTime > enemy.lastChange +
           enemy.TimeToChange then
6          local next = enemy.state + 1
7          enemy.state = next <= enemy.maxState and next
               or 1
8          enemy.lastChange = curTime
9      end
10 end
```

E finalmente para controlar o dano sofrido pelo *boss*, criei um método que é chamado quando um projétil colide com ele, similar ao que foi implementado em Rx, e a verificação da vida foi feita dentro do método *damage*.

```
1  --Enemies.Lua
2  enemy.shotHit = function()
3      if (enemy.state == 1) or (enemy.state == 2 and
           enemyRange.color == enemyRange.safeColor) then
4          enemy.damage(10)
5      end
6  end
7
8  enemy.damage = function(val)
9      enemy.health = enemy.health - val
10
11     -- Checa vida
12     if enemy.health <= 0 then
13         killEnemy(enemy)
14     end
15 end
```

7.10.3 Análise

A minha ideia inicial era trazer uma diferença relativamente grande entre as duas implementações, tendo como principal fator a reutilização de *Observables* em Rx.

Nâo consegui transformar os *Observables* já existentes das outras mecânicas para que isso fosse possível, no entanto consegui demonstrar um outro ponto importante, que é em relação ao fluxo de dados. Isso foi feito com o *scheduler* em comum para atirar o projétil.

Em Rx, todos os inimigos interessados em utilizá-lo podem dar *subscribe*, de maneira que não é necessário saber quem vai precisar do *scheduler*, não sendo necessário tratar isso na implementação do mesmo. Por outro lado, na implementação convencional, o fluxo de dados não funciona do mesmo modo. É preciso passar o dado de objeto pai para objeto filho e assim por diante, para que chegue ao destino. Por isso, precisei chamar um método para atirar projéteis em cada inimigo que o implementasse.

8 Análise geral

Após estudar e utilizar Rx durante esse projeto, pude notar muitas diferenças em implementar um jogo usando essa tecnologia.

Uma das facilidades que a programação reativa tem para oferecer que mais pude notar foi em relação ao controle de tempo. A quantidade de operadores existentes para manipular dados com essa finalidade é bem grande, além de contar com os *schedulers* para trabalhar com corrotinas de maneira simples e sem maiores complicações.

Outro ponto muito importante é em relação à organização e estruturação do código. Utilizando *Observables* de maneira adequada, é possível melhorar vários aspectos do código, como:

- Reduzir a quantidade de variáveis necessárias e assim o código como um todo
- Tornar o código mais compreensível
- Tornar o código mais facilmente extensível
- Facilidade de trocar operadores

- Evitar a geração de *bugs*

A facilidade de trocar operadores traz ainda outra comparação importante, que aconteceu quando eu estava implementando a barra de vida. Em Rx é muito mais simples trocar uma funcionalidade ou comportamento de mecânica apenas trocando um operador por outro, enquanto que sem a biblioteca o programador vai ser obrigado a mudar a implementação ou parte dela.

Um outro aspecto muito importante que deve-se considerar ao programar com *streams* e programação reativa é o fluxo de dados. Os objetos deixam de receber dados e passam a pedir por eles. Um exemplo muito bom é o caso dos métodos da *engine* LÖVE. Na programação convencional, para usar o método `love.update` por exemplo em cada objeto do jogo, é preciso definir um método em cada um desses objetos e chamá-los no `love.update` do módulo principal. Enquanto que com Rx, é possível transformar o método `love.update` em um *Subject* (como é feito pela biblioteca RxLove) e assim todos os objetos do jogo interessados em implementar uma reação ao `update` podem simplesmente dar `subscribe` e reagir de acordo.

Além disso, dependendo da maneira como o programador controlar o fluxo, é possível combinar várias *streams* que possuem objetivos semelhantes no mesmo lugar, simplificando o código. E da mesma maneira, é possível deixar o código mais desacoplado, facilitando a manutenção.

E apesar de não ter conseguido implementar no *boss* como gostaria, as *streams* podem ser "reutilizáveis", de maneira que o fluxo de dados só precisa ser feito uma única vez para diversas situações em que for necessário. E mais uma vez enfatizando que só os objetos que precisarem desses dados irão pedir por eles.

Um ponto importante também é que a biblioteca é facilmente extensível. Como mencionei previamente, implementei três métodos que não existiam antes e consegui utilizá-los em conjunto com o que a biblioteca já oferecia. Dessa maneira, é possível customizá-la para usos específicos como por exemplo um jogo.

No entanto, com certeza existem desvantagens ou pelo menos situações em que o uso de Rx não é ideal ou é desnecessário. Por exemplo, em casos em que é necessária a mudança de estados dentro do *loop* principal do jogo, somente verificando valor de condicionais, a diferença entre as implementações fica bem pequena. Os filtros são substituídos por condicionais e o `subscribe` pelas ações

dentro das condicionais.

Então cheguei a conclusão de que o melhor resultado se encontra na escolha correta de onde utilizar programação reativa e os operadores de Rx. O intuito do projeto era forçar a sua utilização em todos os pontos do jogo, mesmo que fossem mais complicados do que implementando de maneira convencional. É uma ferramenta muito poderosa, mas que com o uso indevido pode trazer complexidades desnecessárias ao código.

9 Considerações Finais

Esse projeto se mostrou ser um grande desafio para mim, pois essa nova maneira de programar, apesar de já conhecer e ter trabalhado brevemente com ela, é muito diferente do que estou habituado a usar no dia a dia.

O projeto é muito atrativo para mim porque trabalha com o mundo de desenvolvimento de jogos, uma paixão minha de anos e que hoje tenho prazer de trabalhar com. Além disso, como já dito, já tinha experimentado um pouco com programação reativa, porém não tinha tido a chance de explorá-la. Esse projeto me permitiu realizar isso e compreender melhor esse paradigma.

Eu aprendi maneiras de usar Rx muito além do que já tinha visto. Além disso, aprendi diversos conceitos sobre programação reativa e Rx que me ajudaram a usá-la propriamente e também poder implementar novos métodos para a bilbioteca.

O mais difícil de trabalhar com Rx é que eu tive que mudar bastante minha maneira de pensar para conseguir trabalhar com essa tecnologia e ao mesmo tempo me forçando a não utilizar soluções convencionais para implementar as mecânicas.

O código fonte do projeto se encontra hospedado no GitHub, através do link: https://github.com/Felipessoaf/ProjFinal.

10 Apêndice

10.1 Diagramas

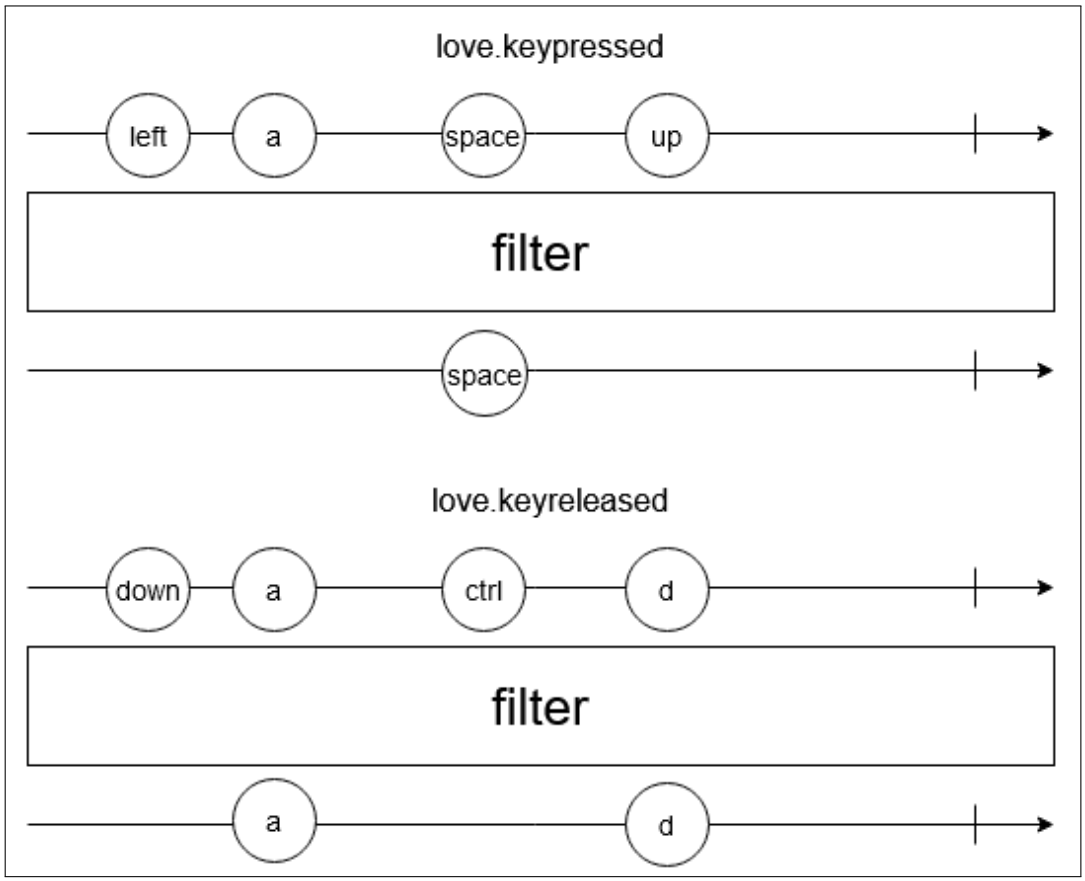


Figura 1: *Input*

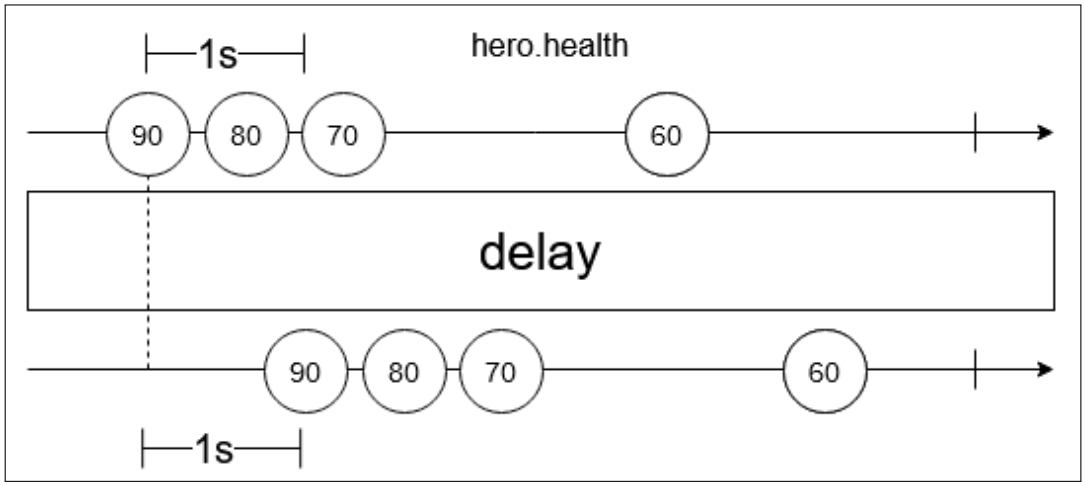


Figura 2: Barra de vida `delay`

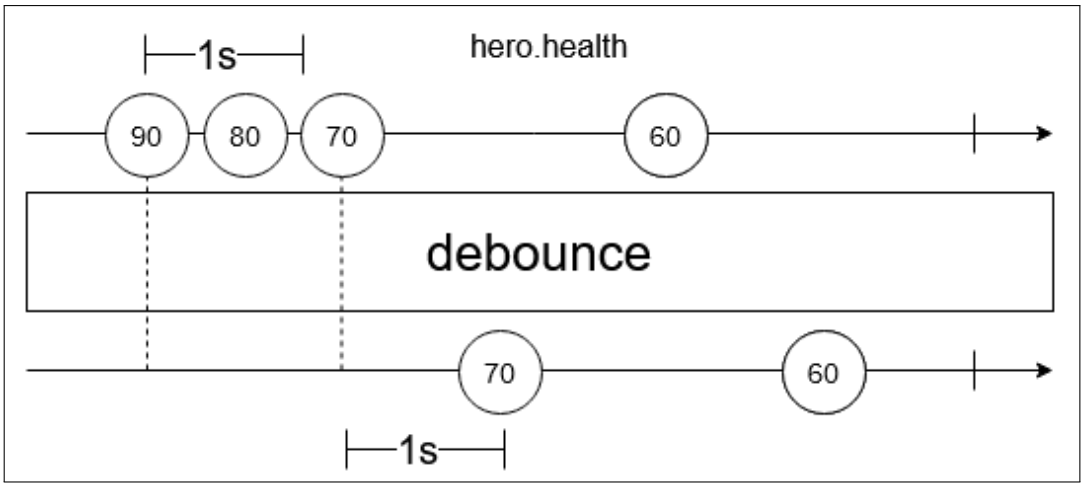


Figura 3: Barra de vida `debounce`

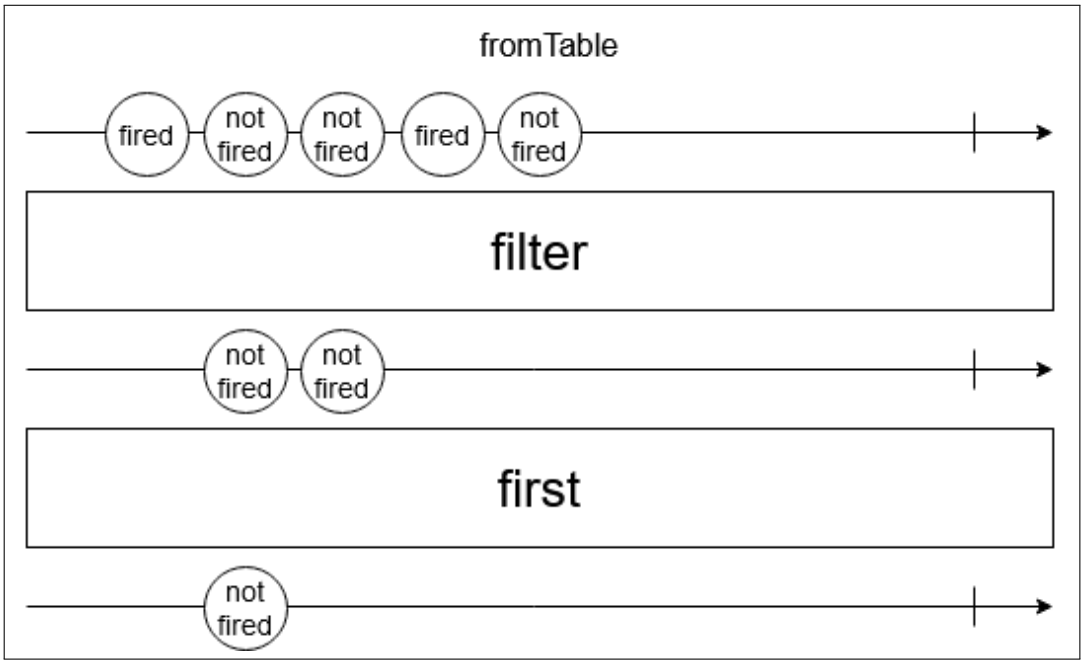


Figura 4: Projétil

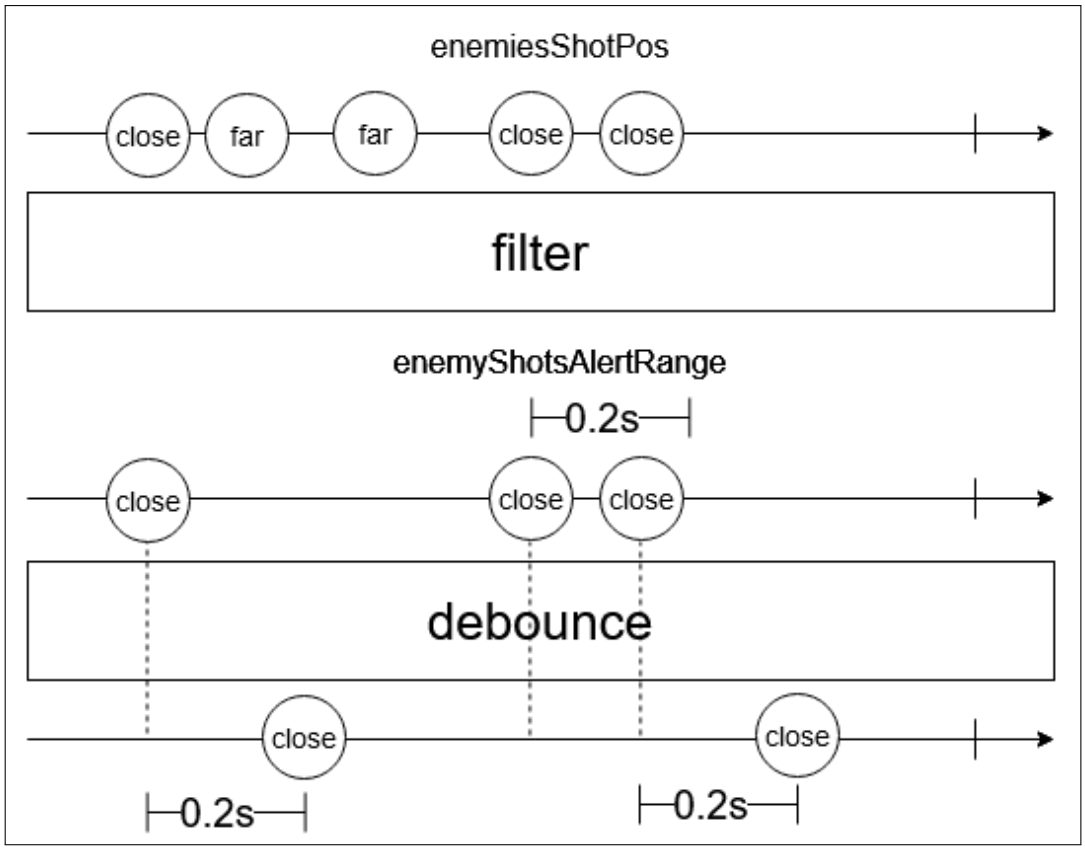


Figura 5: Alerta de perigo

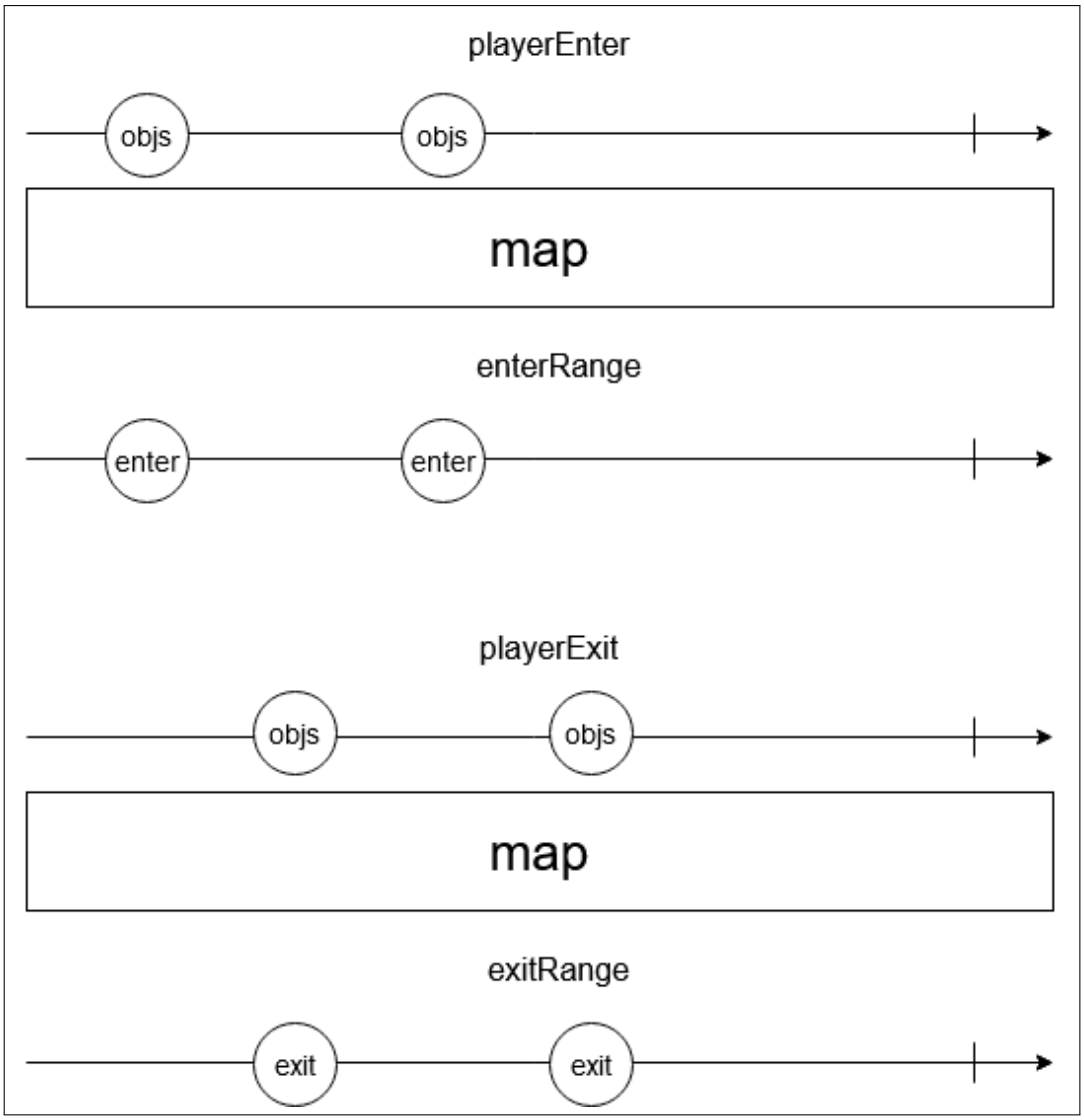


Figura 6: Campo de visão/detecção 1

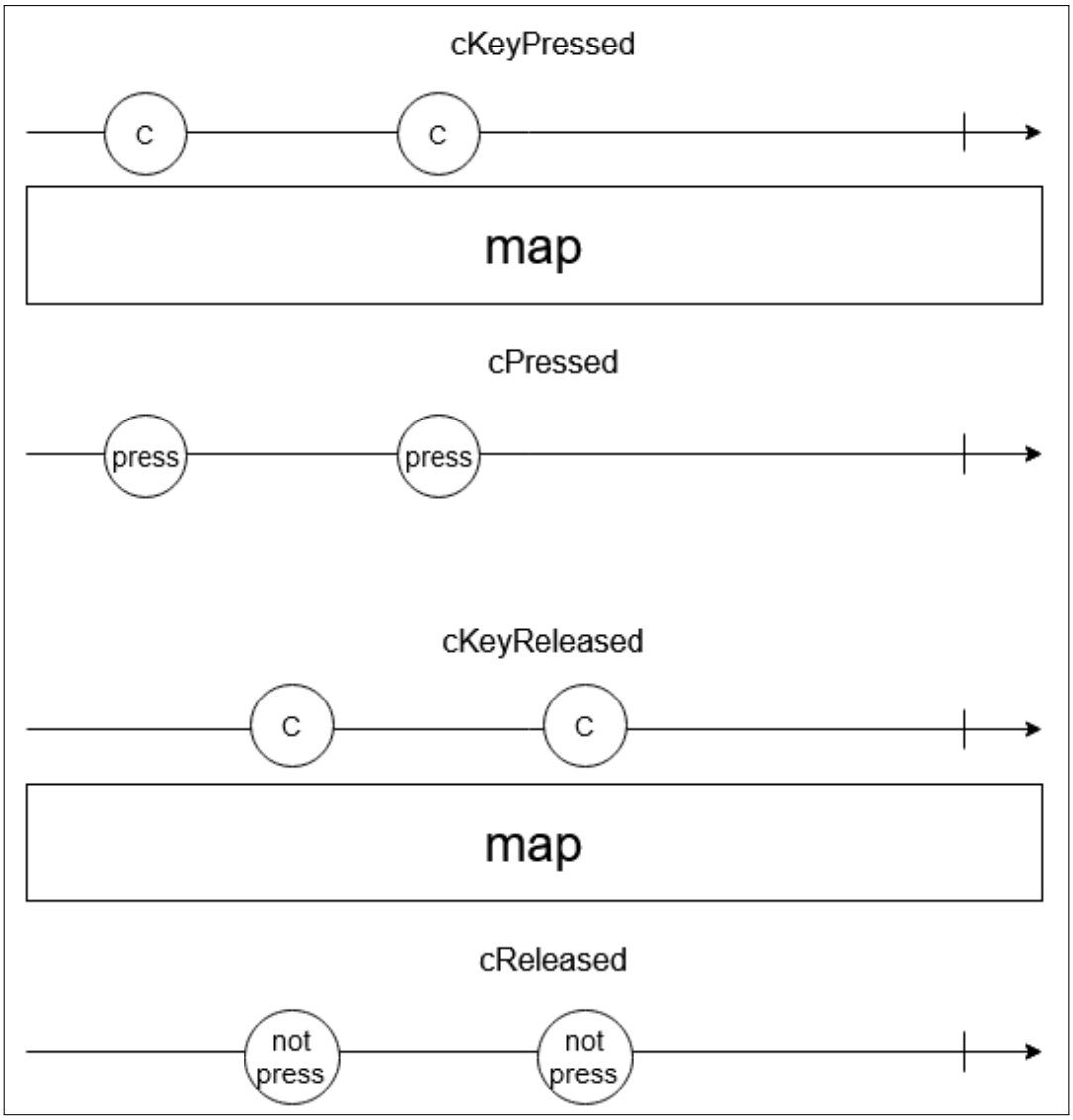


Figura 7: Campo de visão/detecção 2

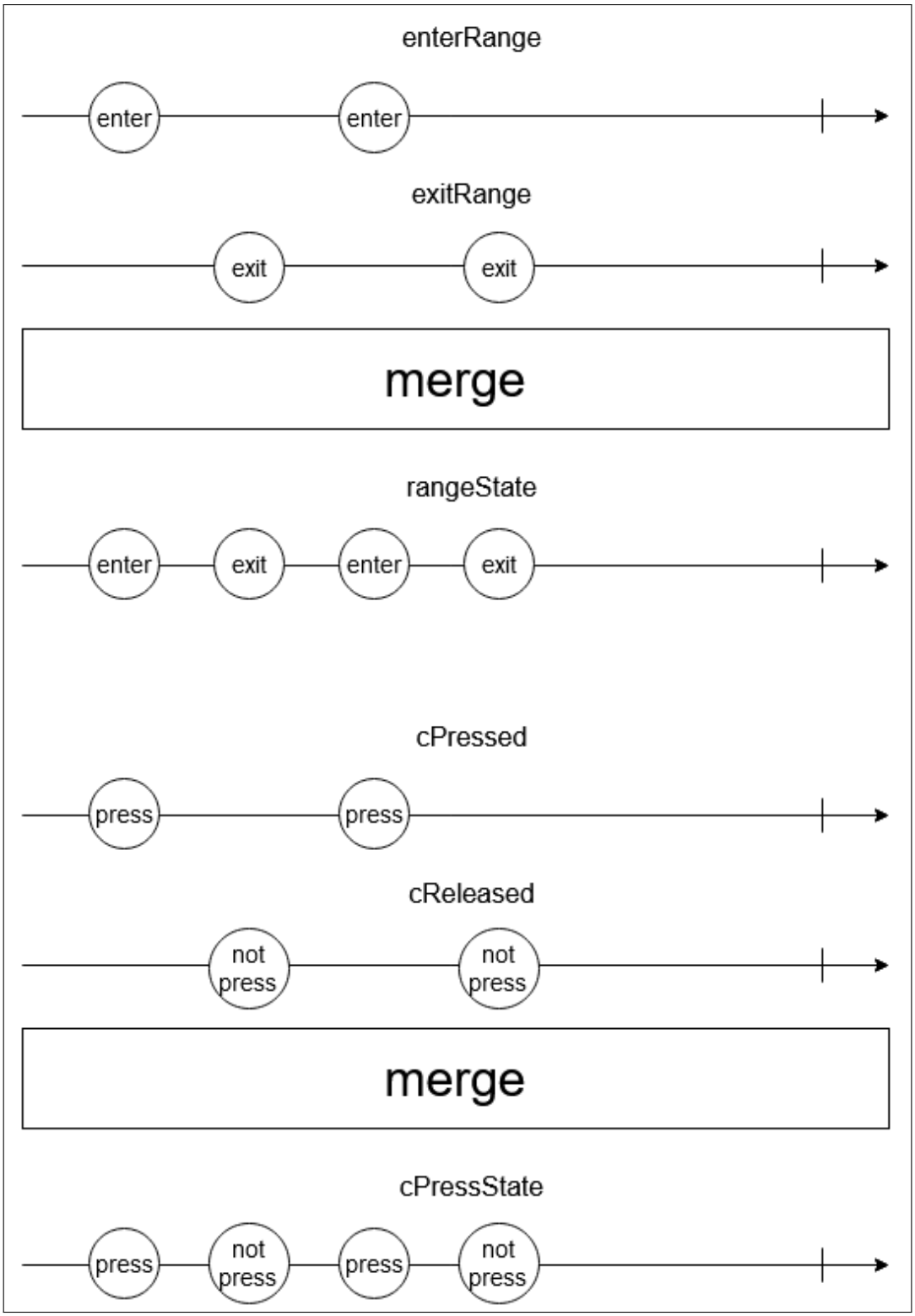


Figura 8: Campo de visão/detecção 3.1

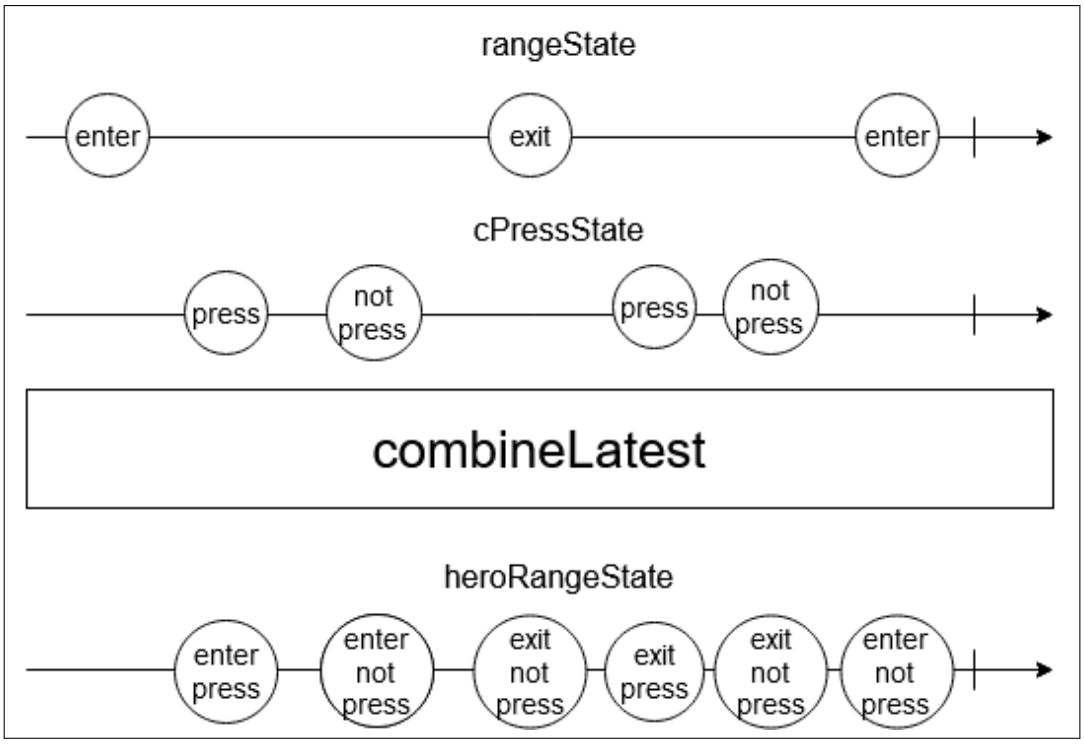


Figura 9: Campo de visão/detecção 3.2

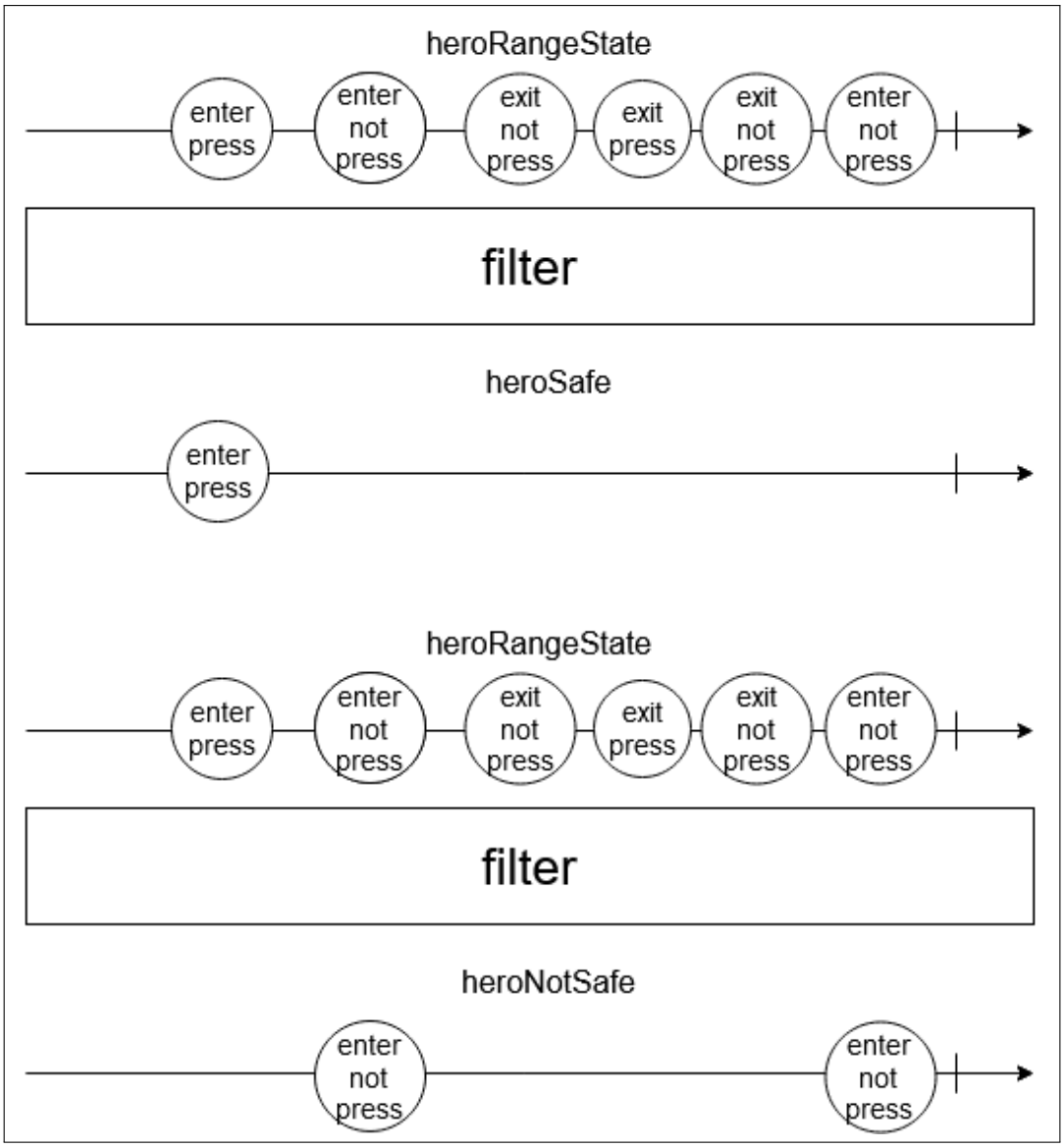


Figura 10: Campo de visão/detecção 4

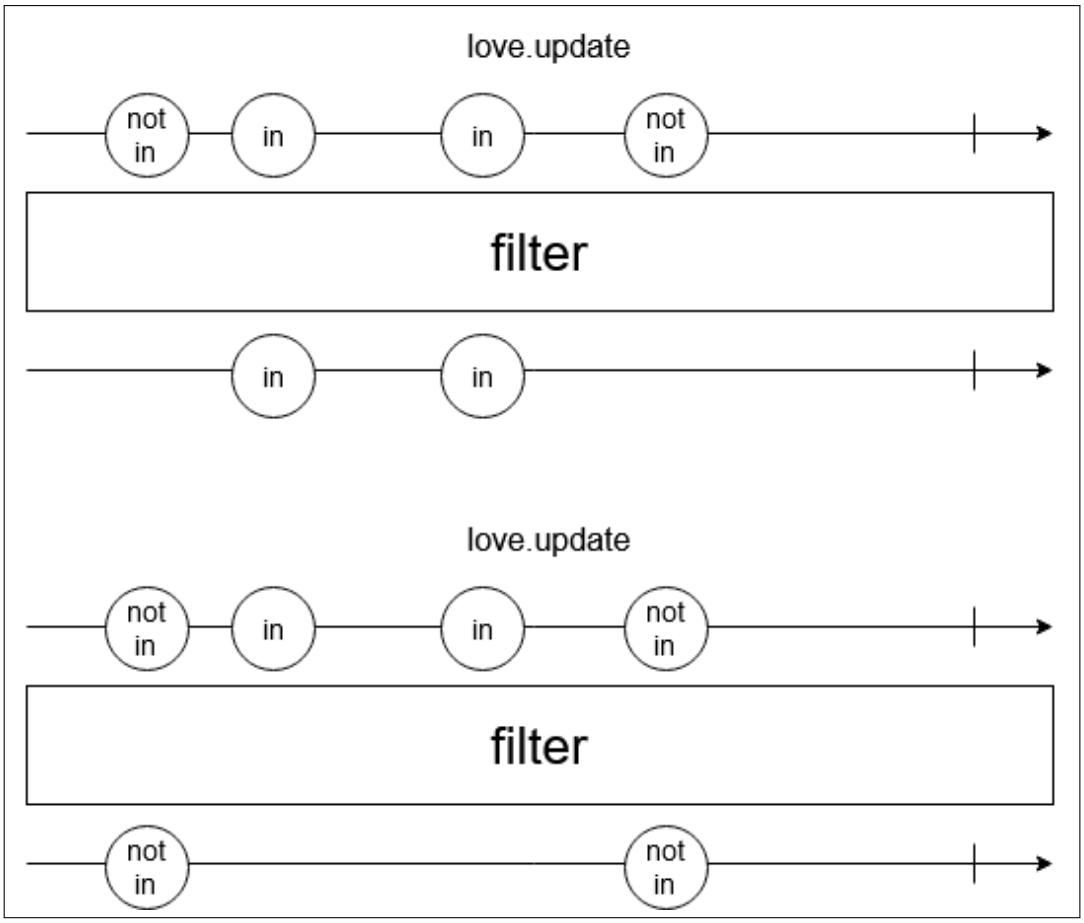


Figura 11: Plataforma móvel

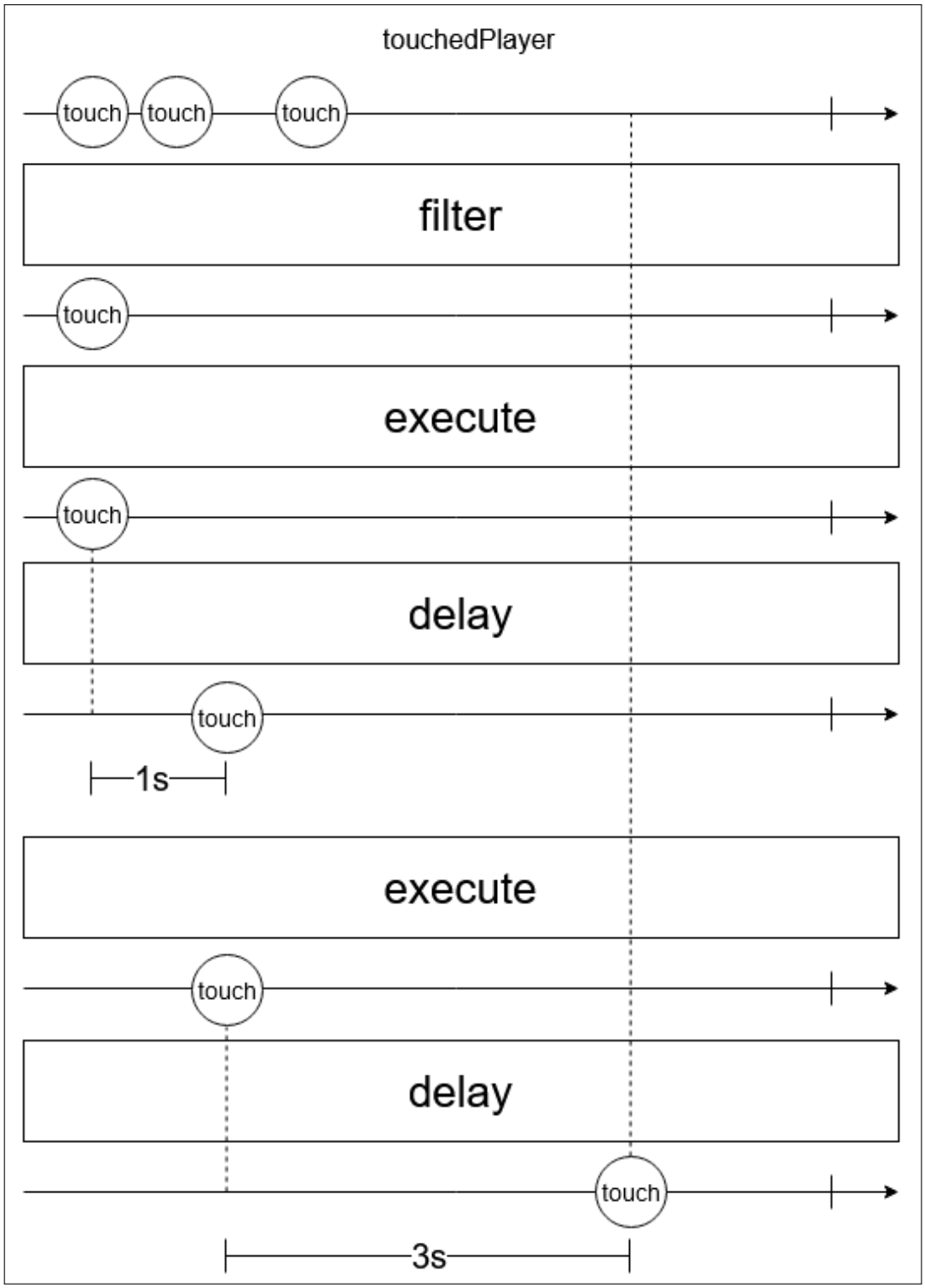


Figura 12: *Falling platform*

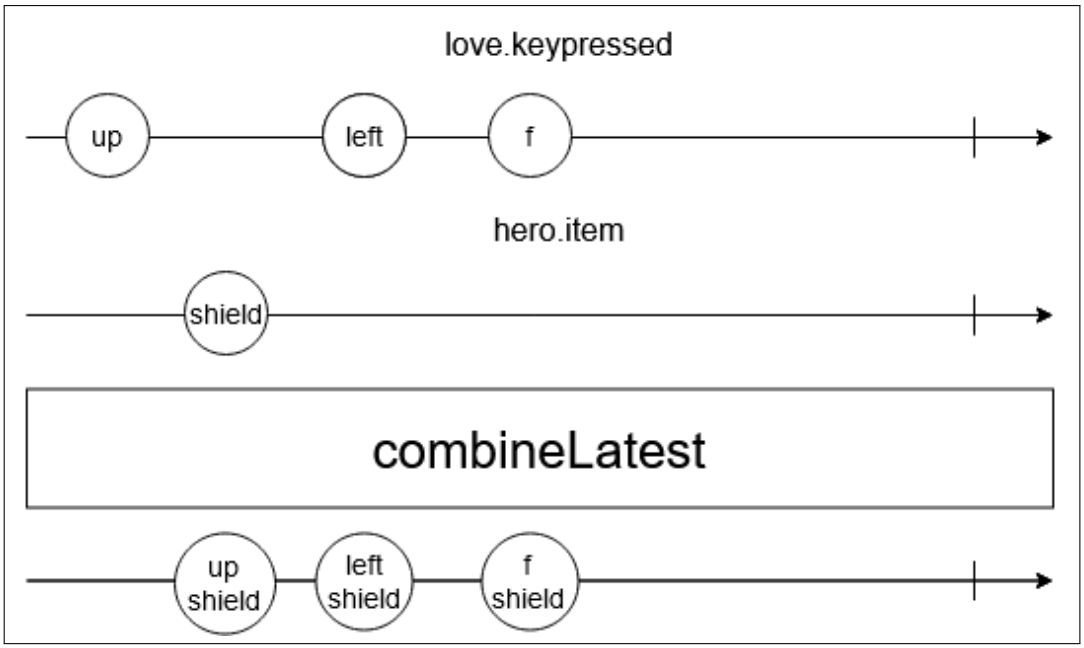


Figura 13: Escudo 1

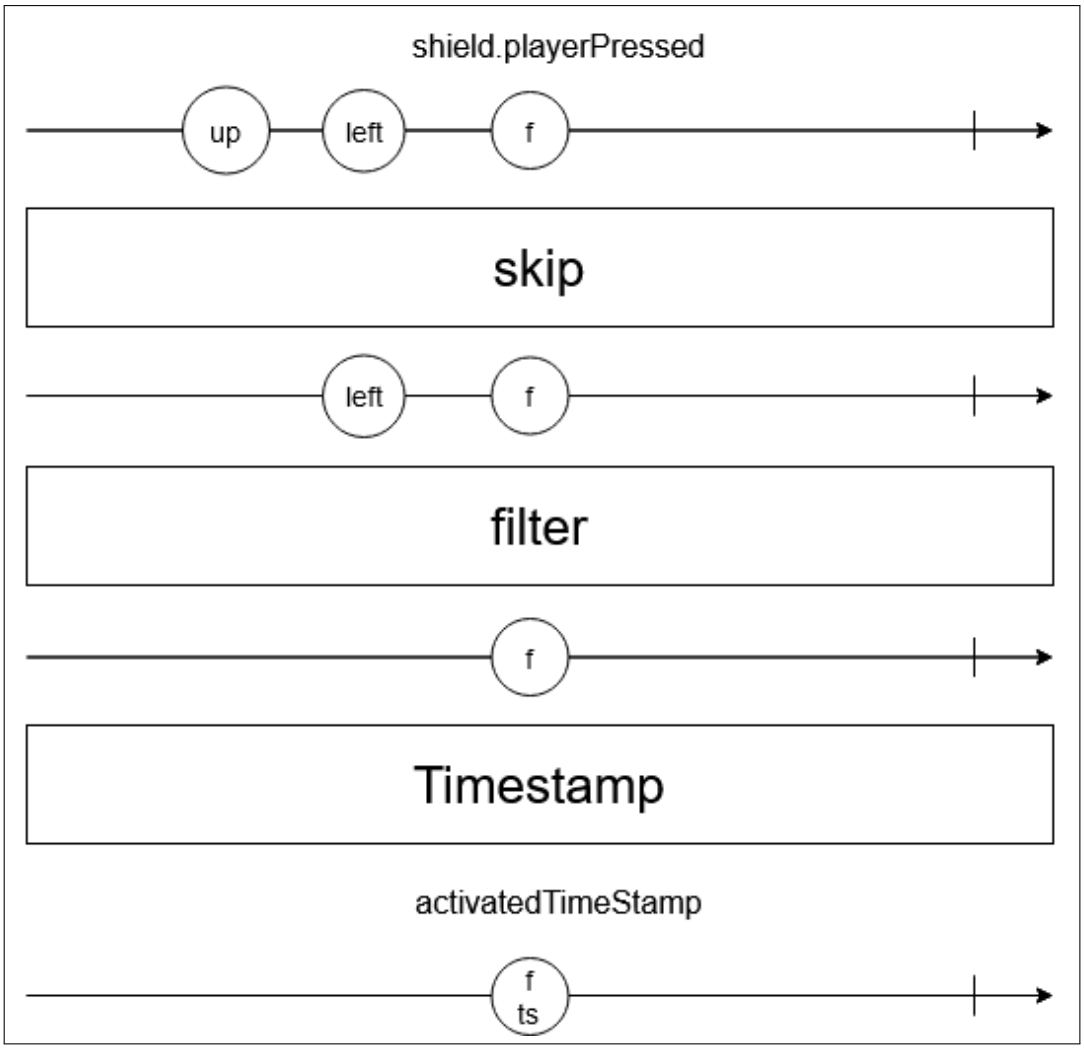


Figura 14: Escudo 2.1

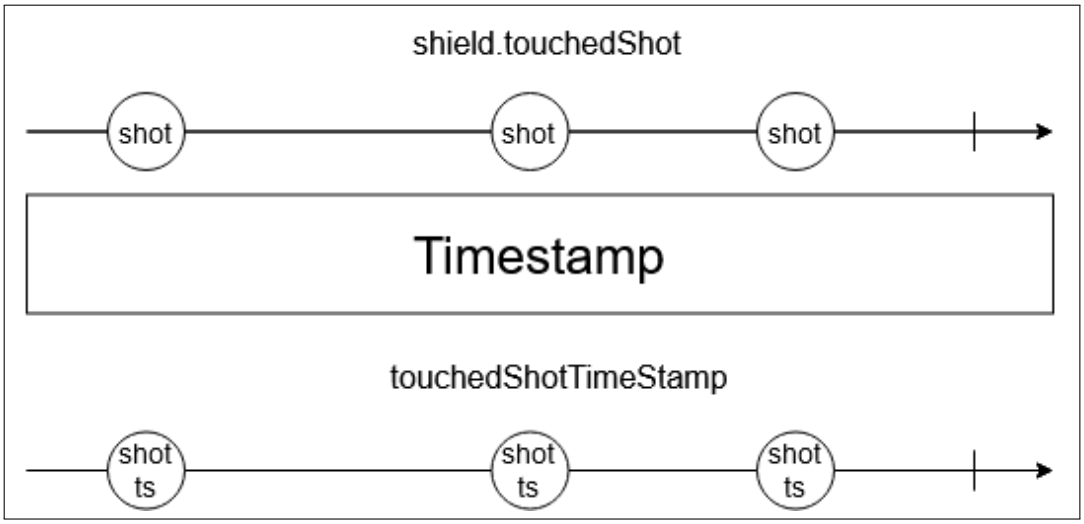


Figura 15: Escudo 2.2

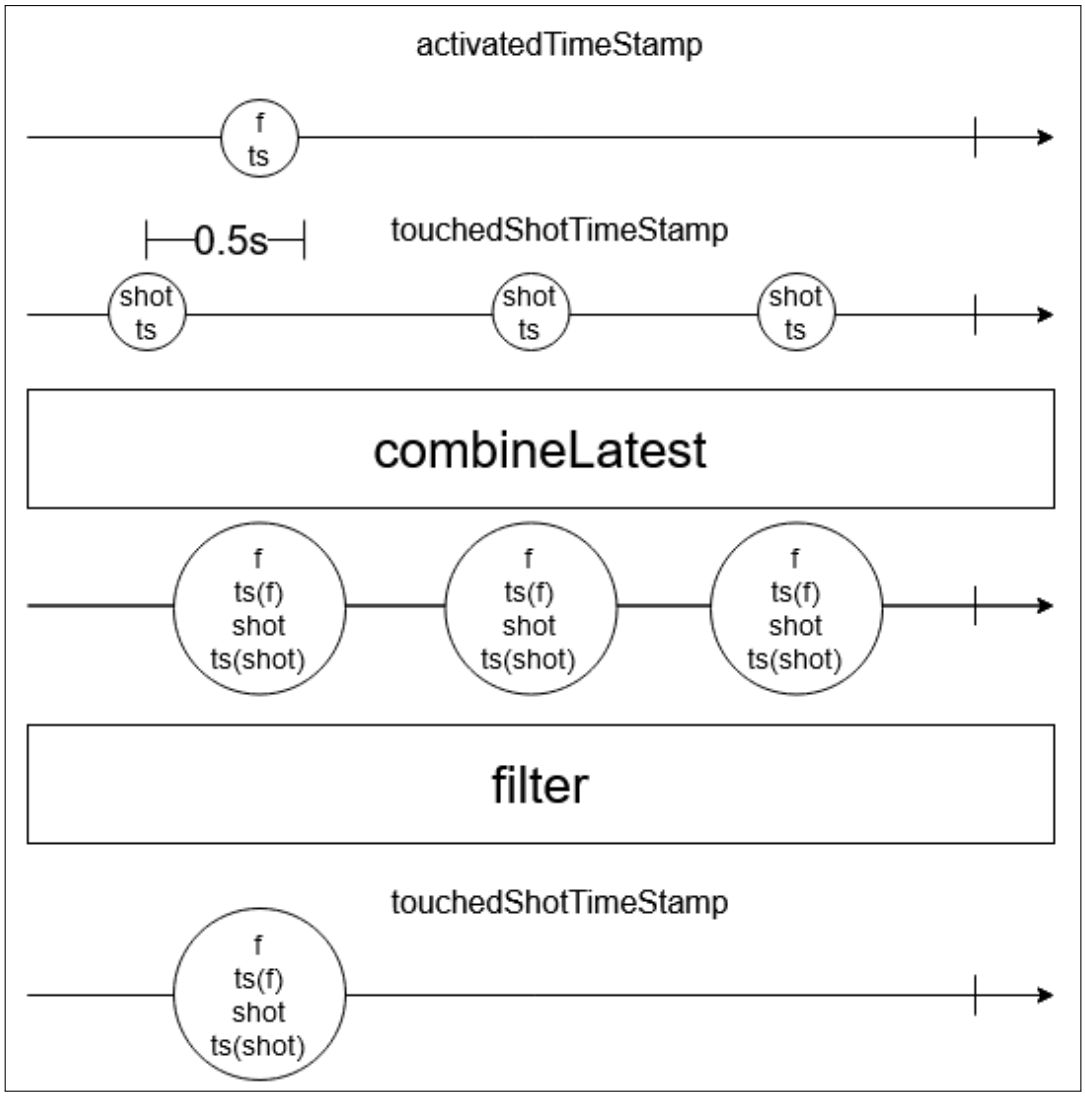


Figura 16: Escudo 2.3

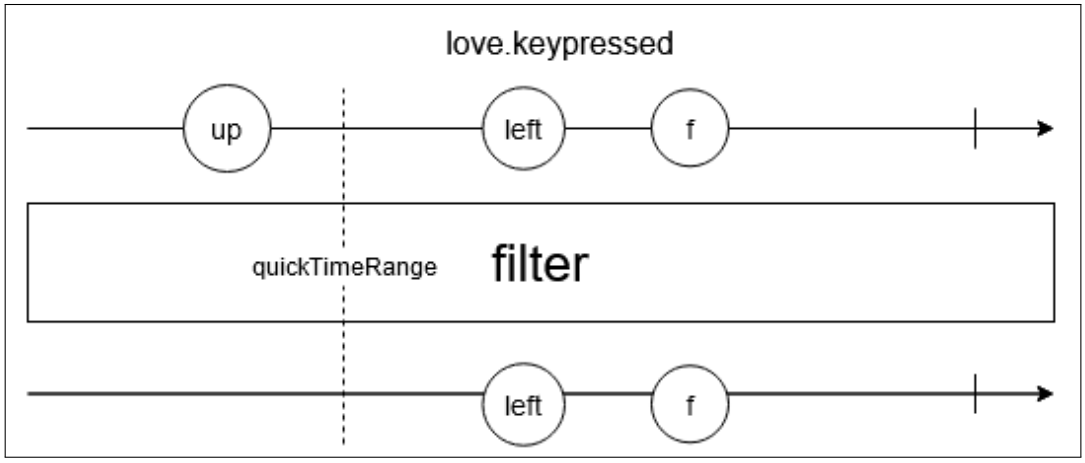


Figura 17: *Quick Time Event* 1

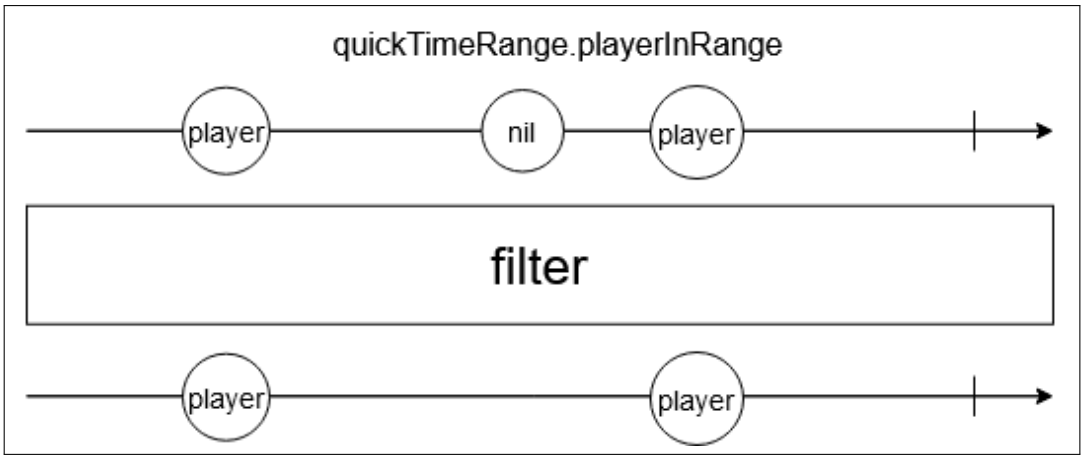


Figura 18: *Quick Time Event* 2

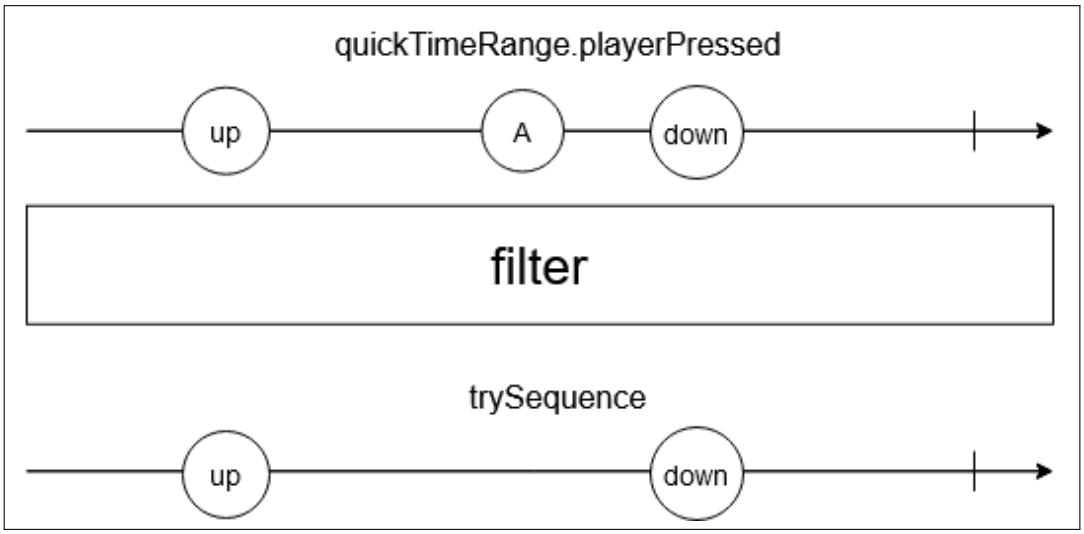


Figura 19: *Quick Time Event* 3

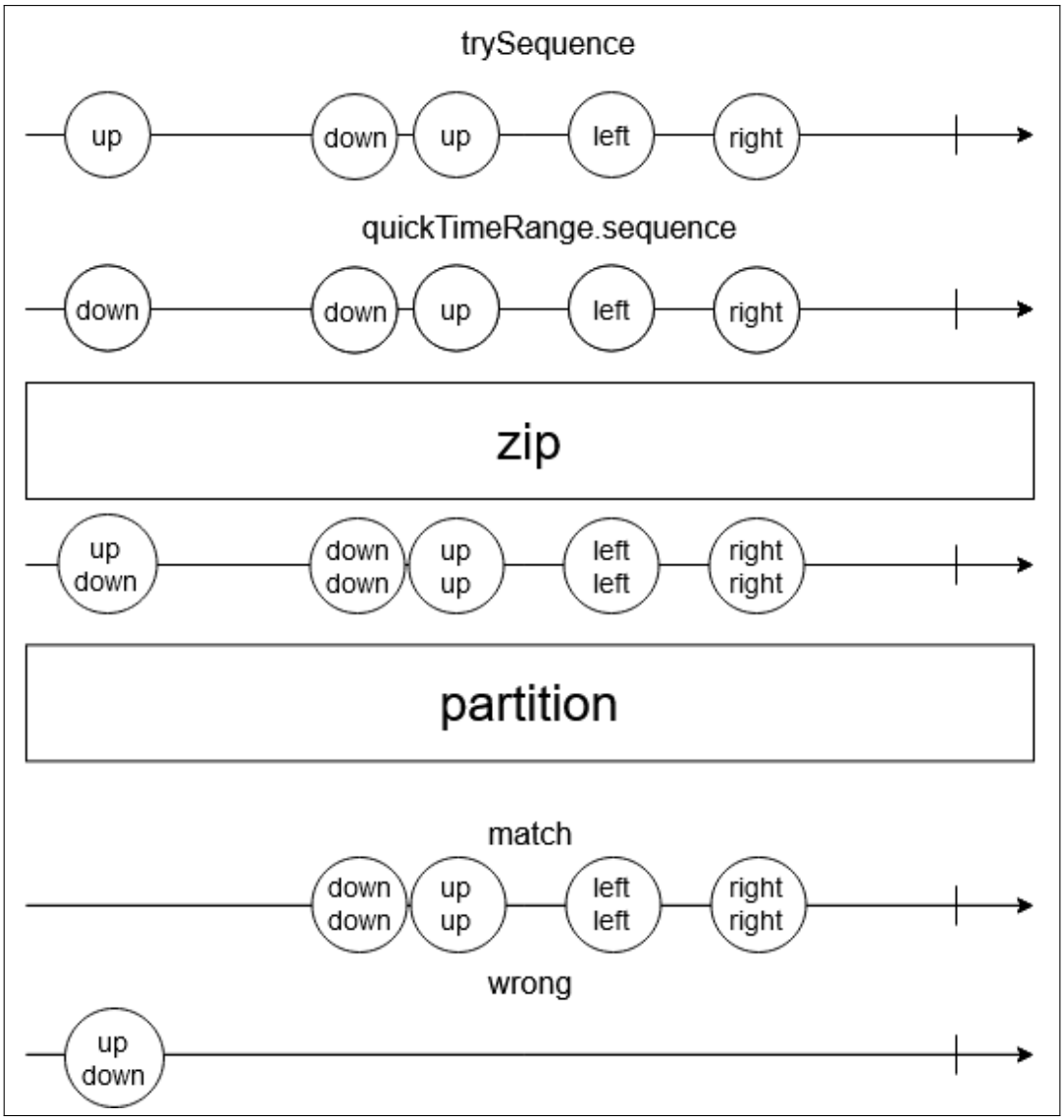


Figura 20: *Quick Time Event* 4

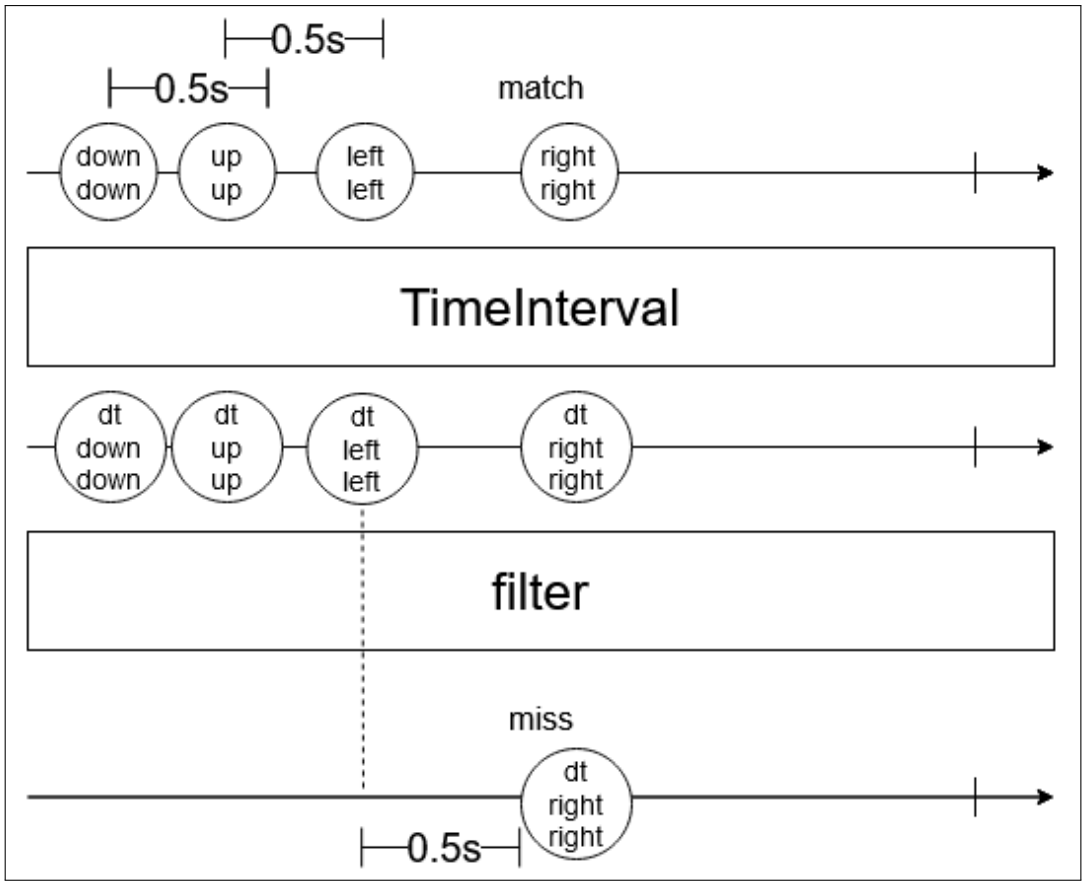


Figura 21: *Quick Time Event* 5.1

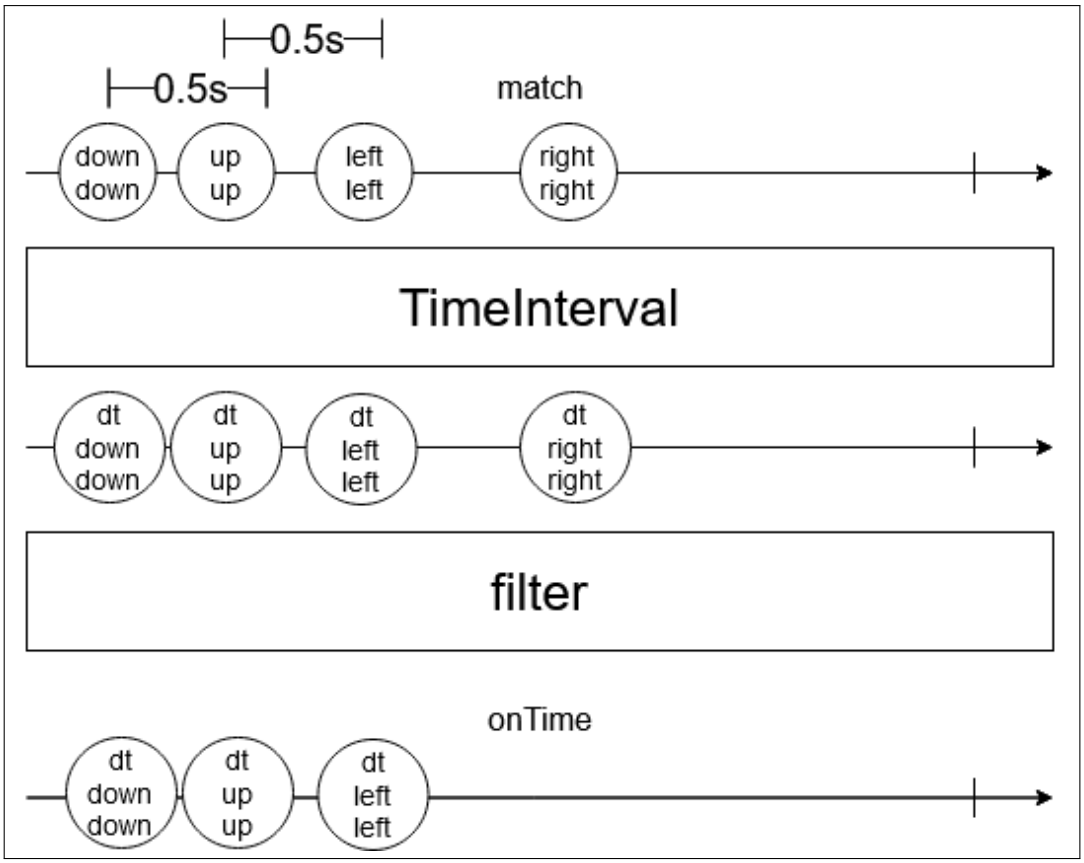


Figura 22: *Quick Time Event* 5.2

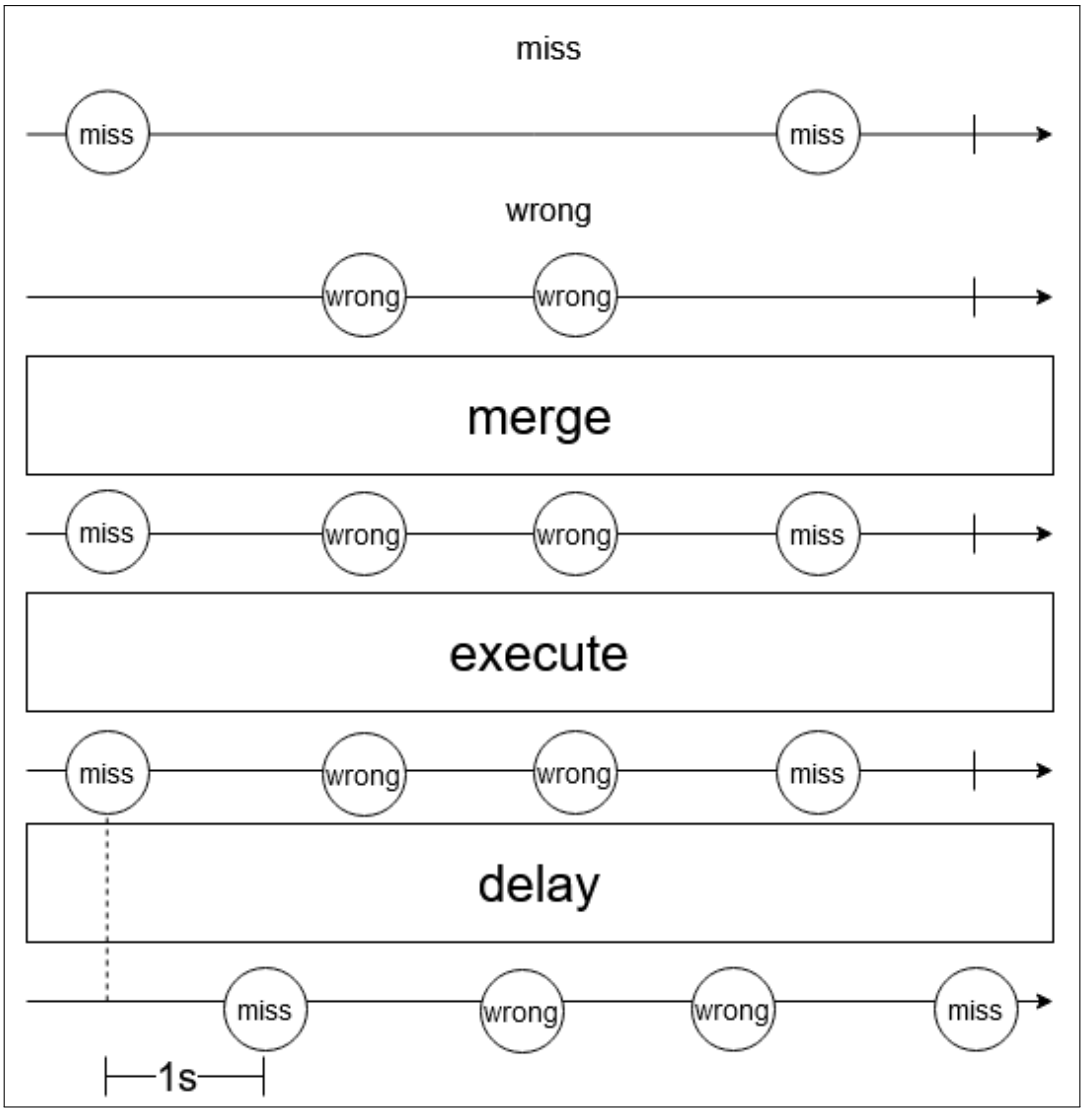


Figura 23: *Quick Time Event* 6.1

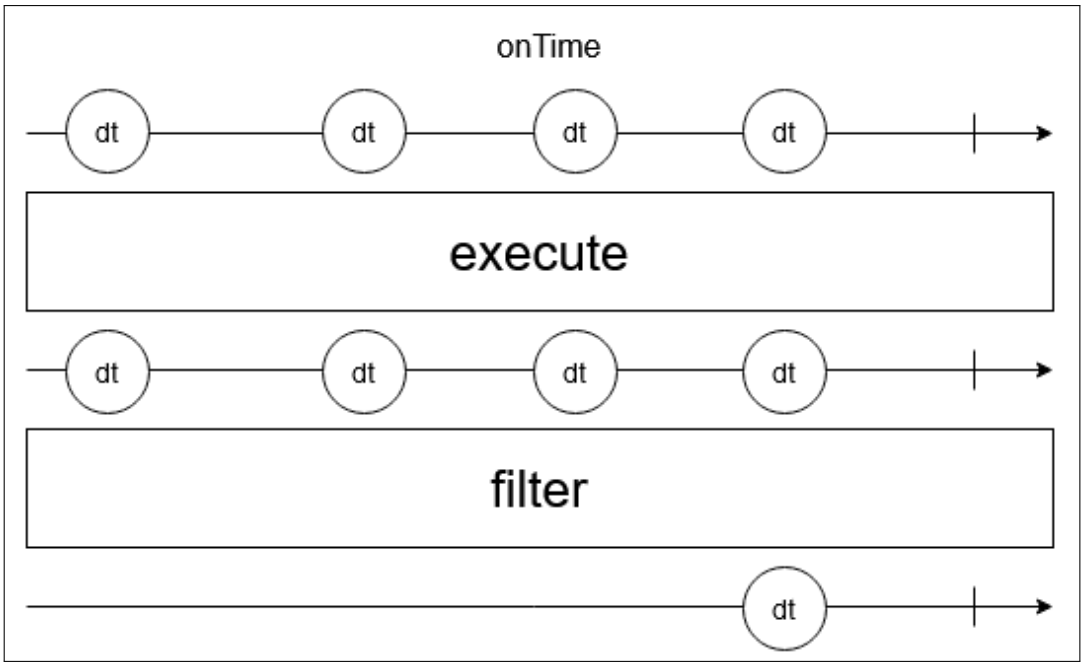


Figura 24: *Quick Time Event* 6.2

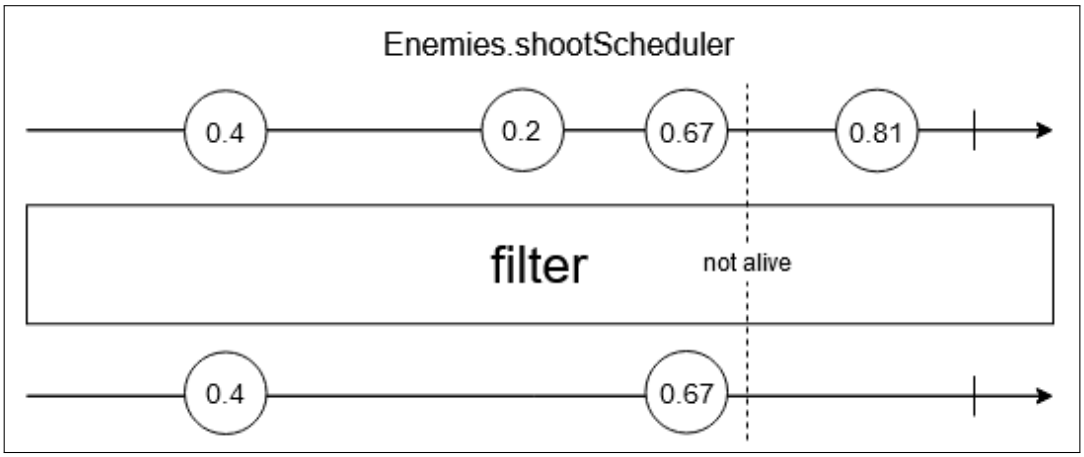


Figura 25: *Boss* 1

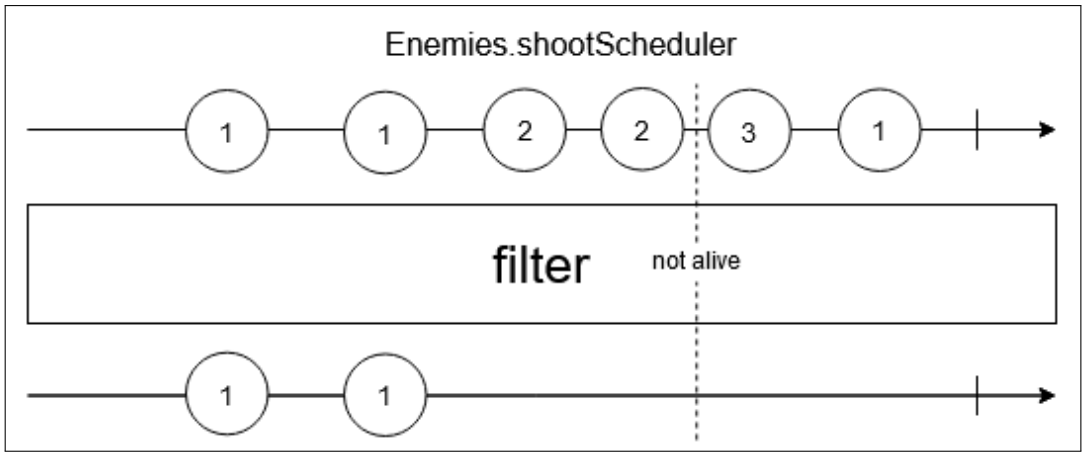


Figura 26: *Boss* 2

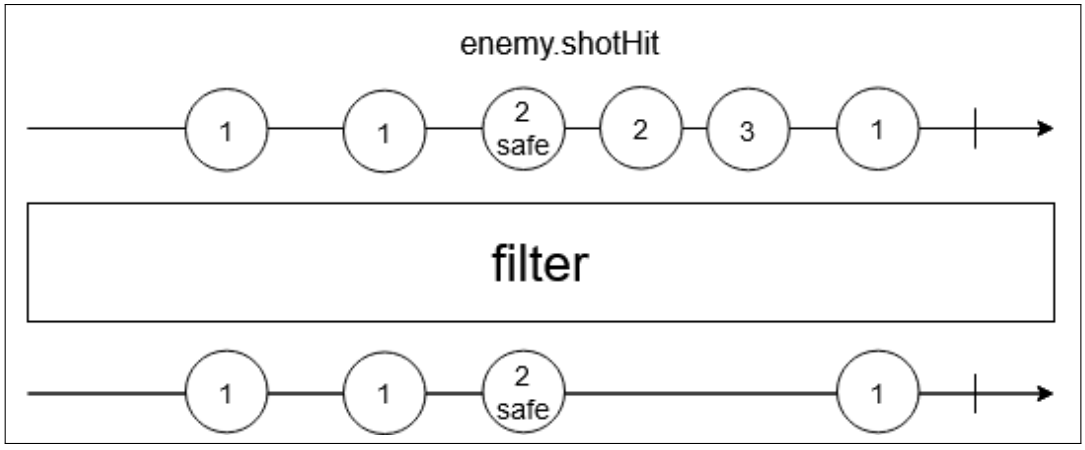


Figura 27: *Boss* 3.1

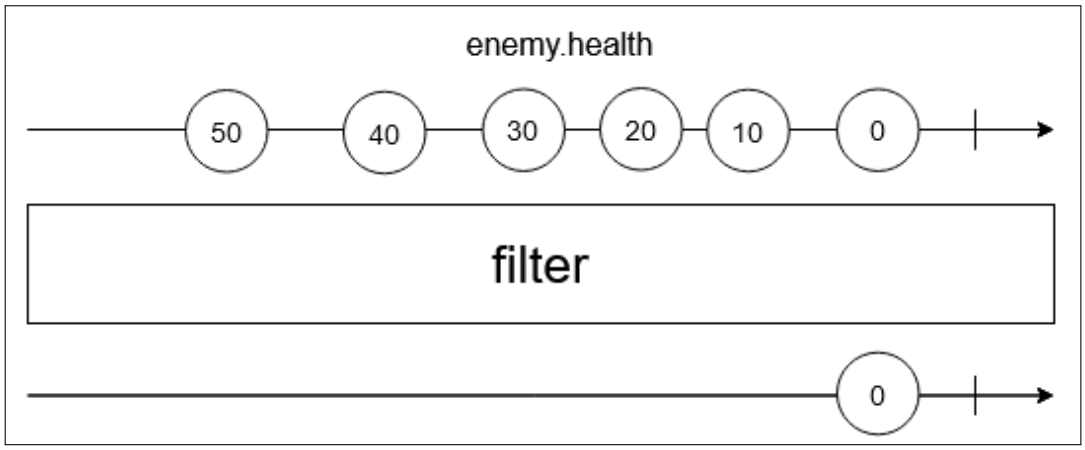


Figura 28: *Boss* 3.2

10.2 Métodos implementados em RxLua

```
1  function Observable:execute(action)
2    return Observable.create(function(observer)
3      local function onNext(...)
4        util.tryWithObserver(observer, function(...)
5          action(...)
6          return observer:onNext(...)
7        end, ...)
8      end
9
10     local function onError(e)
11       return observer:onError(e)
12     end
13
14     local function onCompleted()
15       return observer:onCompleted()
16     end
17
18     return self:subscribe(onNext, onError,
           onCompleted)
19   end)
20 end
```

Código 32: Execute em RxLua

```
1  function Observable:TimeInterval(scheduler)
2    local lastTime = scheduler:getCurrentTime()
3
4    return Observable.create(function(observer)
5      local function onNext(...)
6        local values = util.pack(...)
7        util.tryWithObserver(observer, function(...)
8          local dt = scheduler:getCurrentTime() -
               lastTime
9          lastTime = scheduler:getCurrentTime()
10         return observer:onNext(dt,util.unpack(values))
11       end, ...)
12     end
13
14     local function onError(e)
15       return observer:onError(e)
16     end
17
18     local function onCompleted()
19       return observer:onCompleted()
20     end
21
22     return self:subscribe(onNext, onError,
           onCompleted)
23   end)
24 end
```

Código 33: TimeInterval em RxLua

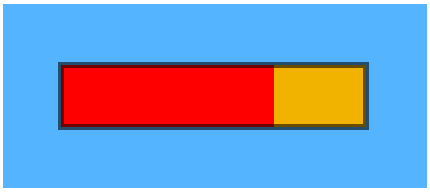
```
function Observable:Timestamp(scheduler)
  return Observable.create(function(observer)
    local function onNext(...)
      local values = util.pack(...)
      util.tryWithObserver(observer, function(...)
        return observer:onNext({
            time = scheduler:getCurrentTime(),
            other = util.unpack(values)
        })
      end, ...)
    end

    local function onError(e)
      return observer:onError(e)
    end

    local function onCompleted()
      return observer:onCompleted()
    end

    return self:subscribe(onNext, onError,
       onCompleted)
  end)
end
```

Código 34: Timestamp em RxLua

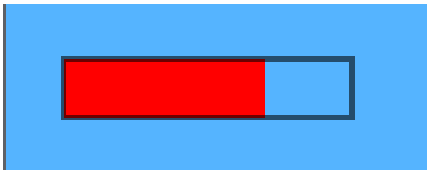
```
--- Returns the CooperativeScheduler's currentTime.
function CooperativeScheduler:getCurrentTime()
  return self.currentTime
end
```

Código 35: CooperativeScheduler getCurrentTime()

10.3 Imagens do jogo

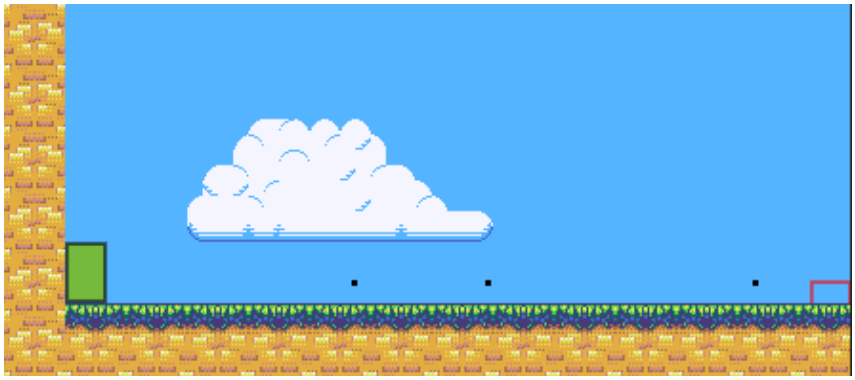


(a) Barra de vida ao sofrer danos sequenciais

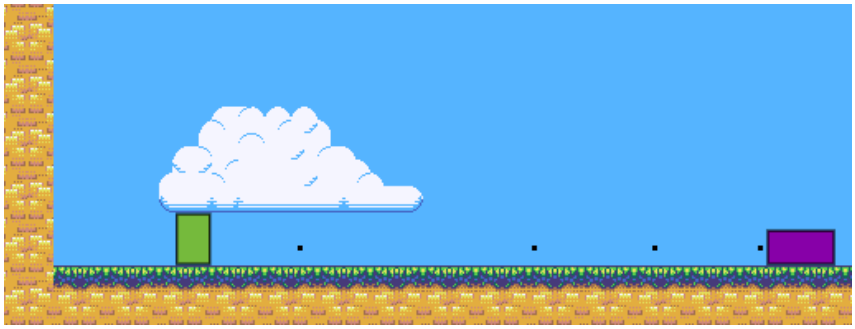


(b) Barra de vida após 1 segundo do último dano

Figura 29: Barra de vida

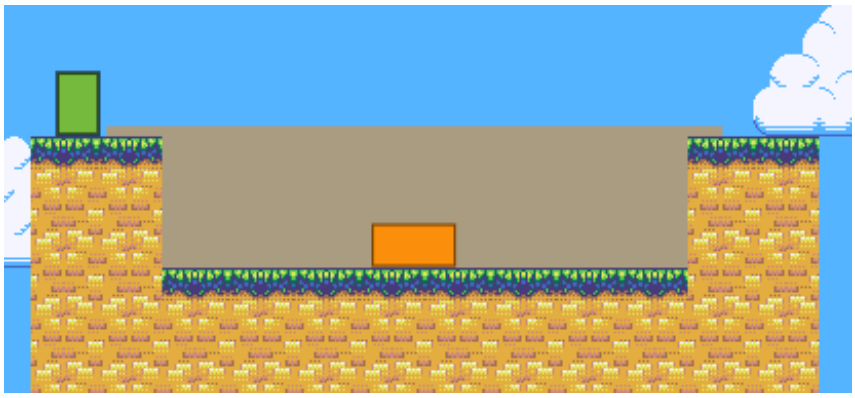


(a) Alerta de projéteis fora da tela

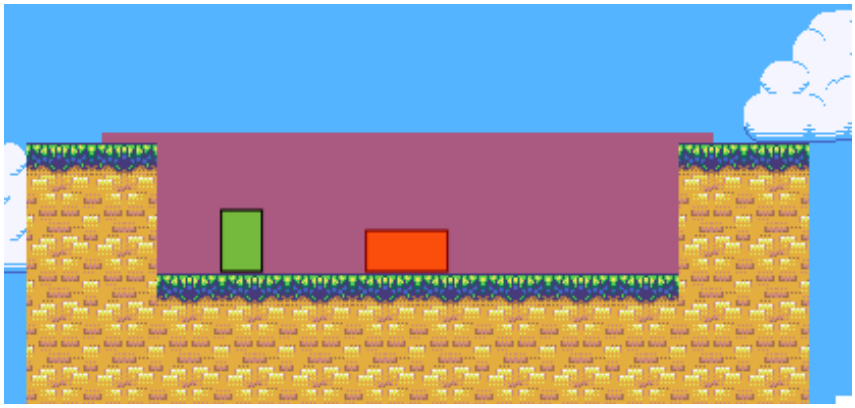


(b) Projéteis dentro da tela

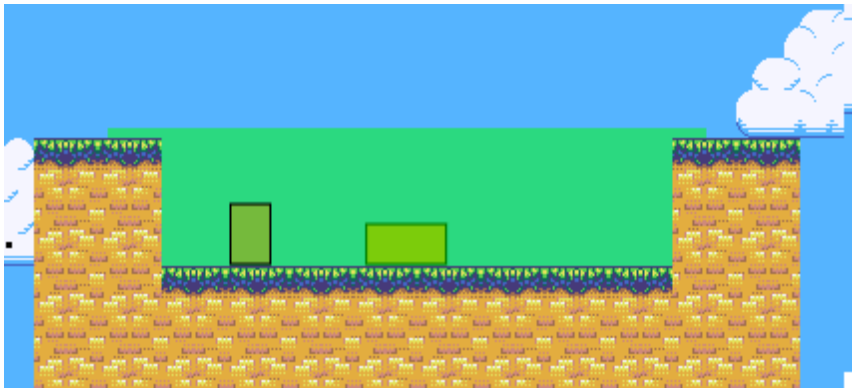
Figura 30: Alerta (projéteis)



(a) Fora do alcance

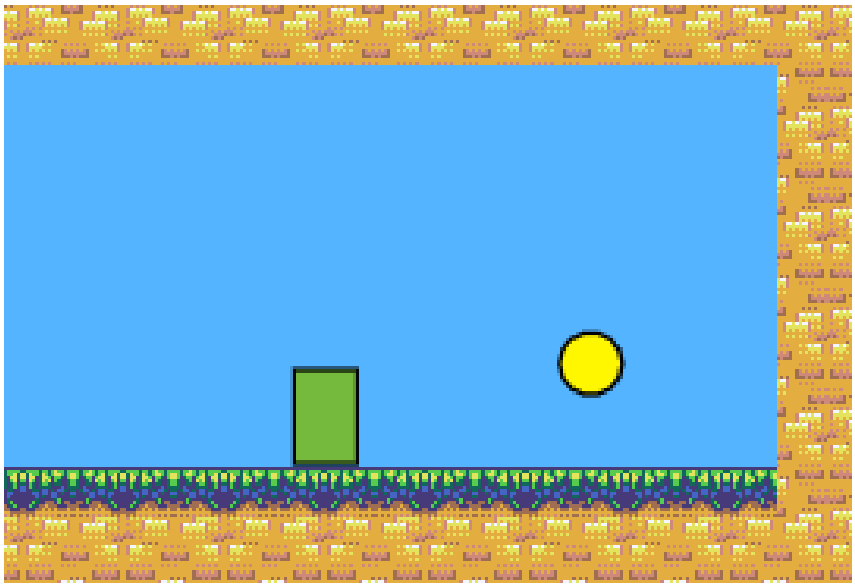


(b) Dentro do alcance sem apertar nada (perigo)

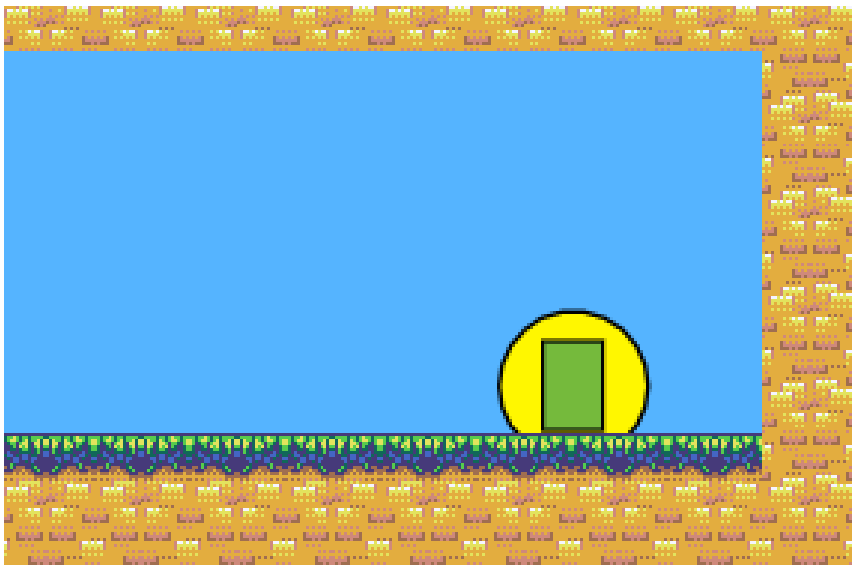


(c) Dentro do alcance apertando “C” (seguro)

Figura 31: Campo de visão

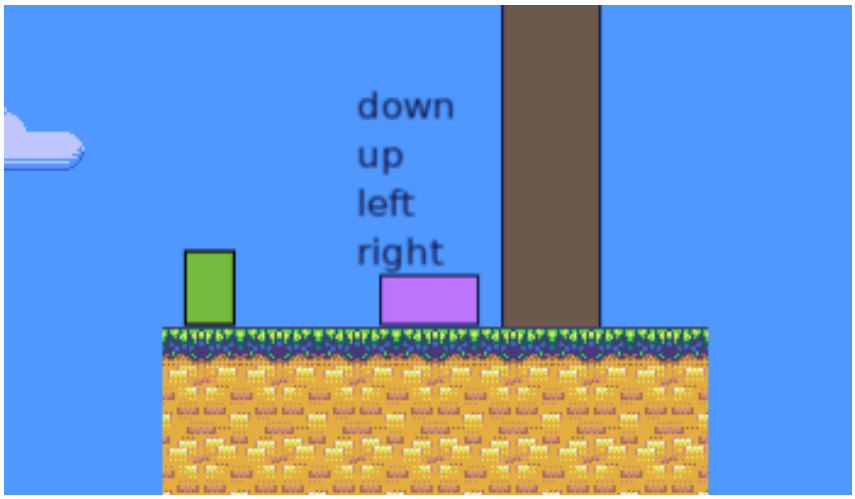


(a) Antes de coletar o escudo

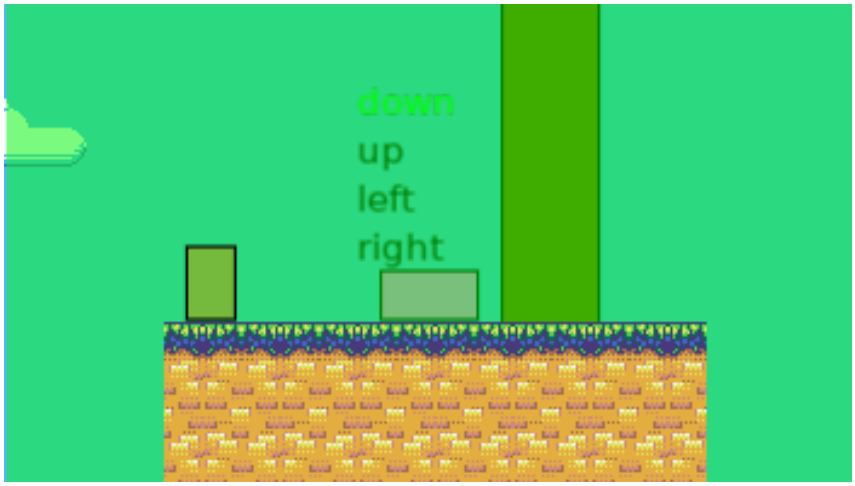


(b) Após coletar o escudo

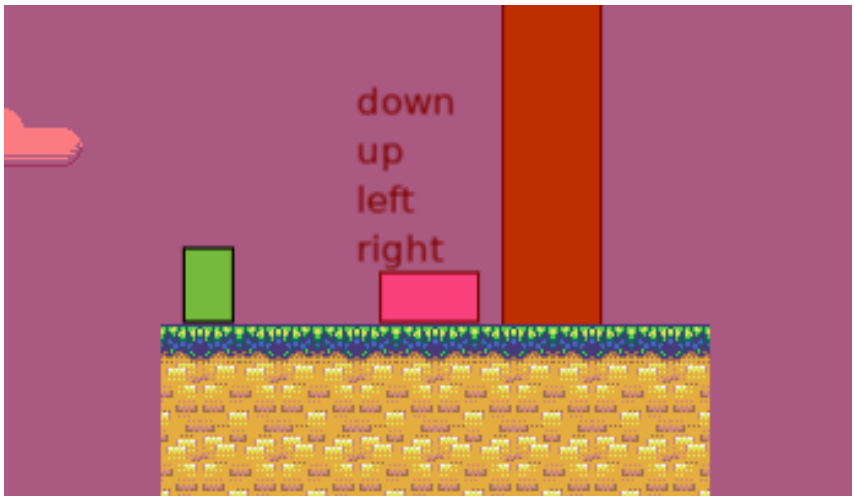
Figura 32: Escudo



(a) Esperando sequência

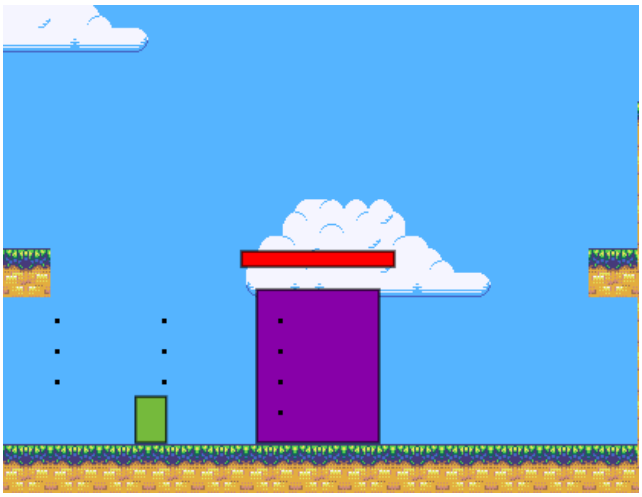


(b) Acertou o primeiro item da sequência

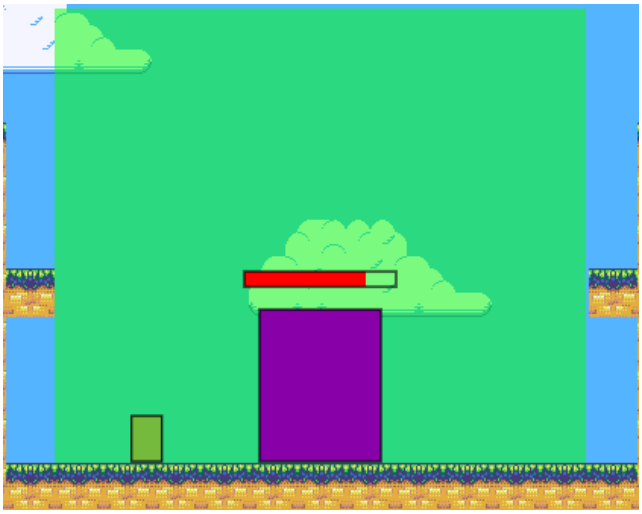


(c) Erro ou tempo limite ultrapassado

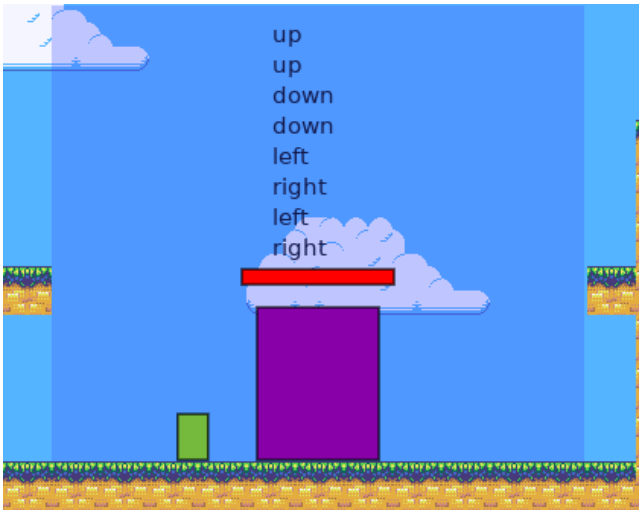
Figura 33: *Quick Time Event*



(a) Estado 1: projétil



(b) Estado 2: campo de visão



(c) Estado 1: *Quick Time Event*

Figura 34: *Boss*

11 Referências bibliográficas

[1] WePC. 2019 Video Game Industry Statistics, Trends & Data. Disponível em: https://www.wepc.com/news/video-game-statistics/. Acesso em: 13 set. 2019

[2] GAMEDESIGNING. The Evolution of Video Game Genres. Disponível em: https://www.gamedesigning.org/gaming/video-game-genres/. Acesso em: 13 set. 2019

[3] STALTZ, André. The introduction to Reactive Programming you've been missing. Disponível em: https://gist.github.com/staltz/868e7e9bc2a7b8c1f754. Acesso em: 13 set. 2019

[4] UniRx. "neuecc/UniRx" GitHub. Disponível em: https://github.com/neuecc/UniRx. Acesso em: 16 set. 2019.

[5] RxLove. "bjornbytes/RxLove". GitHub. Disponível em: https://github.com/bjornbytes/RxLove. Acesso em: 16 set. 2019.

[6] RxLua. "bjornbytes/RxLua" GitHub. Disponível em: https://github.com/bjornbytes/RxLua. Acesso em: 16 set. 2019.

[7] FRLua. "aiverson/frlua" GitHub. Disponível em: https://github.com/aiverson/frlua. Acesso em: 16 set. 2019.

[8] BAINOMUGISHA, Engineer; CARRETON, Andoni Lombide; CUTSEM, Tom Van; MOSTINCKX, Stijn; DE MEUTER, Wolfgang. 2012. A Survey on Reactive Programming. Disponível em: http://soft.vub.ac.be/Publications/2012/vub-soft-tr-12-13.pdf. Acesso em: 24 set. 2019

[9] BONÉR,Jonas; FARLEY, Dave; KUHN, Roland; THOMPSON, Martin. The Reactive Manifesto. Disponível em: https://www.reactivemanifesto.org/. Acesso em: 05 nov. 2019

[10] This is RxJS v 4.. "Reactive-Extensions/RxJS" GitHub. Disponível em: https://github.com/Reactive-Extensions/RxJS/blob/master/doc/gettingstarted/creating.md#cold-vs-hot-observables. Acesso em: 12 nov. 2019.

[11] ReactiveX. Observable. Disponível em: http://reactivex.io/documentation/observable.html. Acesso em: 13 nov. 2019

[12] ReactiveX. ReactiveX. Disponível em: http://reactivex.io. Acesso em: 13 nov. 2019

[13] ReactiveX. Scheduler. Disponível em: http://reactivex.io/documentation/scheduler.html. Acesso em: 13 nov. 2019

[14] GAMEDESIGNING. Beginner's Guide to Game Mechanics. Disponível em: https://www.gamedesigning.org/learn/basic-game-mechanics/. Acesso em: 24 abr. 2020

[15] ROGERS, Scott. Level UP: um guia para o design de grandes jogos. São Paulo: Blucher, 2013.

[16] SUTHERLAND, Jeff. Scrum: a arte de fazer o dobro do trabalho na metade do tempo. São Paulo: Leya, 2016.