# Capítulo 1

# Desenvolvimento de Aplicações Declarativas para TV Digital no Middleware Ginga com Objetos Imperativos NCLua

Francisco Sant'Anna, Carlos de Salles Soares Neto, Roberto Gerson de Albuquerque Azevedo e Simone Diniz Junqueira Barbosa

***Abstract***

*This short course intends to offer to its participants training in the development of declarative applications using NCLua imperative objects. These applications are based in the Ginga middleware specification, the standard middleware of the Brazilian Digital TV System. More specifically, this course is related with applications whose initial entity is written in Nested Context Language (NCL) containing media objects with imperative code written in Lua, the NCLua objects. Some basic concepts of programming in NCL and Lua are presented as a basis for the development of these applications. Special attention is dedicated to the division between declarative and imperative authoring, and the relationship between them. All concepts are illustrated by examples of increasing difficulty level, developed step by step along the course.*

***Resumo***

*O objetivo deste mini-curso é capacitar os participantes no desenvolvimento de aplicações declarativas que se utilizam de objetos imperativos NCLua com base no middleware Ginga, padrão do Sistema Brasileiro de TV Digital. Mais precisamente, o curso trata de aplicações cuja entidade inicial é escrita em Nested Context Language (NCL) e que contêm objetos de mídia de código imperativo escrito em Lua, os objetos NCLua. O curso inicia apresentando alguns conceitos básicos de NCL e de programação em Lua, cujo entendimento é necessário para o desenvolvimento dessas aplicações. Especial atenção é dedicada à divisão entre autoria declarativa e autoria imperativa, bem como ao relacionamento entre elas. Todos os conceitos são ilustrados por exemplos de nível crescente de dificuldade, desenvolvidos passo a passo ao longo do curso.*

## 1.1. Introdução

O *Middleware* Ginga, desenvolvido pela PUC-Rio e pela UFPB, é o padrão do Sistema Brasileiro de TV Digital [ABNT 2007]. O Ginga é uma camada de software que dá suporte à execução de aplicações interativas nos conversores digitais, esses últimos instalados nas casas dos telespectadores. A arquitetura do Ginga é composta tanto por um ambiente declarativo quanto por um imperativo, o que permite usar a alternativa mais adequada quando se está desenvolvendo aplicações para TV digital. É possível, inclusive, fazer uso de uma ponte para desenvolver aplicações declarativas com partes imperativas e vice-versa.

O ambiente imperativo do Ginga, também chamado de *máquina de execução*, é a parte responsável pelo suporte a aplicações desenvolvidas usando a linguagem de programação Java. O ambiente declarativo, também chamado de *máquina de apresentação*, interpreta aplicações desenvolvidas em Nested Context Language (NCL).

Linguagens declarativas são, em geral, criadas para um determinado foco ou domínio e permitem a especificação de aplicações em um nível mais alto de abstração quando comparadas a linguagens imperativas. Linguagens declarativas são voltadas para especificar a intenção final e não uma sequência passo a passo para resolver um determinado problema, como é o caso de linguagens imperativas.

NCL é uma linguagem declarativa que permite o desenvolvimento de aplicações multimídia com sincronismo espaço-temporal entre objetos de mídia, tais como vídeos, áudios, imagens e textos. Com a finalidade de ampliar a gama de aplicações que pode ser desenvolvida, a NCL também suporta objetos escritos na linguagem Lua, denominados objetos imperativos NCLua. Um objeto NCLua se comunica com o documento NCL no qual ele está embutido através de eventos, de acordo com as transições em sua máquina de estados. Na prática, cabe ao próprio exibidor de conteúdos usuais (como áudio, vídeo etc) o papel de interpretar as ações comandadas pela máquina de apresentação, tais como o início ou término da exibição. Contudo, para objetos imperativos (como os NCLua), esse papel deve ser feito pelo autor do objeto, o qual pode dar qualquer semântica para as ações por ele comandadas.

Através do uso de objetos NCLua em documentos NCL, o autor ganha em expressividade. Ele pode ir além da programação declarativa com NCL e elaborar aplicações que fogem ao escopo habitual da TV e exigem uma linguagem de *script*. Torna-se possível, por exemplo, criar aplicações com o desenho de gráficos e com o cálculo de fórmulas matemáticas.

O objetivo principal deste minicurso é descrever como desenvolver objetos NCLua. O Capítulo está organizado como se segue. A Seção 1.2 descreve o Ambiente Declarativo Ginga-NCL, fornecendo uma base conceitual mínima sobre NCL para o desenvolvimento de objetos NCLua. A Seção 1.3 apresenta um Tutorial NCLua com exemplos de níveis crescentes de dificuldade que ilustram os recursos do ambiente gradativamente. A Seção 1.4 apresenta uma Documentação de Referência NCLua para servir de consulta rápida.

## 1.2. O Ambiente Declarativo Ginga-NCL

Esta seção tem o objetivo de apresentar resumidamente os princípios básicos do ambiente declarativo Ginga-NCL. Nesse sentido, os principais conceitos e elementos da

linguagem *Nested Context Language* (NCL) são discutidos. Nesta seção são apresentados os objetos de mídia, os relacionamentos entre eles e seus principais atributos, com destaque para os objetos imperativos, foco deste minicurso. Cabe ressaltar que o objetivo aqui não é esgotar todos os conceitos de NCL e que informações mais detalhadas podem ser encontradas em [ABNT 2007; Soares Neto *et al.* 2007; Soares e Barbosa 2009].

A linguagem NCL atua como uma linguagem de cola que orquestra objetos de mídia de vários tipos diferentes, relacionando-os no tempo e no espaço, criando o que se denomina de uma apresentação multimídia. O autor da apresentação multimídia deve informar à máquina de apresentação **o que** ela deve tocar, **onde**, **como** e **quando**.

Um documento NCL é um arquivo XML cuja estrutura está dividida em duas partes principais: o cabeçalho (elemento <head>) e o corpo (elemento <body>). No cabeçalho estão as bases de informação que especificam **onde** e **como** o conteúdo deve ser exibido. No corpo está descrito **o que** é o conteúdo a ser exibido e **quando** isso deve ser feito.

Através dos objetos de mídia (elementos <media>), chamados de nós de mídia na terminologia NCL, define-se **o que** tocar, ou seja, qual conteúdo é exibido. Uma mídia pode ser um áudio, vídeo, texto, imagem e até códigos imperativos, como aqueles escritos em NCLua ou em Java. Nós de mídia são definidos dentro de um outro tipo de nó, o nó de composição ou contexto (elemento <context>).

Os nós de composição têm a função de encapsular outros nós, que podem ser tanto nós de mídia como também nós de composição, recursivamente. Para que seja possível acessar um nó interno a um contexto, é necessário definir nesse contexto um ponto de entrada ou porta (elemento <port>) que aponte para tal nó interno. O corpo do documento (elemento <body>) também é um contexto, que é tratado pela máquina de apresentação como o contexto principal do documento. Todos os outros nós (de mídia ou de composição) estão contidos no corpo do documento. Ao encapsular outros nós, um nó de composição também tem as funções de estruturar e permitir o reúso de partes de um documento NCL.

Em NCL, os nós de mídia não carregam em sua especificação a informação de **onde** eles devem ser exibidos. Isso é definido por elementos de primeira classe denominados regiões (elementos <region>). Uma região define uma área da tela onde uma mídia é apresentada. Essa separação entre a região e a mídia é importante, principalmente, para possibilitar o reúso de uma região por outros nós de mídia e até mesmo por outras aplicações NCL. As regiões são definidas através de sua posição (*top*, *bottom*, *left* e *right*), sua altura (*height*) ou sua largura (*width*). Em NCL, esses valores podem ser dados tanto em porcentagem como em pixels. Adicionalmente, toda região deve possuir um identificador único (atributo *id*), que é utilizado pelo descritor para associá-la a uma ou mais mídias.

Para associar uma mídia a uma região, são usados os descritores (elementos <descriptor>), que definem **como** nós de mídia são inicialmente apresentados. Por exemplo: uma mídia de áudio pode estar associada a um descritor que estabelece que sua apresentação ocorre com volume a 50% do original; uma imagem pode ser associada a um descritor que estabelece uma transparência de 80%; entre outros. O elemento <descriptor> possui um atributo *region* que deve referenciar um identificador de uma região. O descritor também possui um atributo *id* que é único e utilizado pelo

objeto de mídia, para se associar a esse descritor. A Figura 1.1 exemplifica a relação entre nó de mídia, descritor e região.

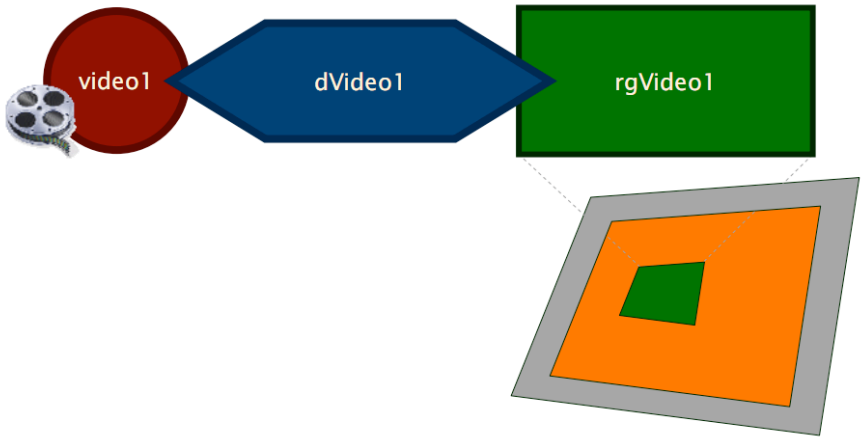


**Figura 1.1** Associação entre nó de mídia (“video1”), descritor (“dVideo1”) e região (“rgVideo1”).

Uma vez informados **o que** tocar, **onde** tocar e **como** tocar, agora resta especificar **quando** tocar os diferentes nós. Quando se pede para tocar um documento NCL, a ação resultante na máquina de apresentação é meramente a de exibir o corpo do documento (elemento <body>) como um contexto qualquer. Isso faz com que todos os nós apontados pelas portas no corpo do documento sejam inicialmente exibidos. Dessa forma, para especificar que um nó deve ser exibido no início da apresentação de um documento, é suficiente criar, no corpo do documento, uma porta que aponte para esse nó. A Listagem 1.1 mostra um exemplo de código NCL que apenas exibe o nó de mídia “video1”.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ncl id="primeiroDocumentoNCL"
     xmlns="http://www.ncl.org.br/NCL3.0/EDTVProfile">

    <!-- Cabeçalho -->
    <head>

        <regionBase>
            <!--
                Define uma região que ocupa toda a tela
            -->

            <region id="rgVideo1" width="50%" height="50%"
                        top="20%" left="30%"/>
        </regionBase>

        <descriptorBase>

            <!-- Descritor associando-se a região -->
            <descriptor id="dVideo1" region="rgVideo1"/>

        </descriptorBase>

    </head>

    <!-- Corpo -->
```

```
    <body>

        <!--
               A porta associa qual o primeiro nó de mídia a ser
                tocado, através do atributo component
        -->
        <port id="portaPrincipal" component="video1"/>

        <media id="video1" src="video.mp4"
               descriptor="dVideo1"/>

    </body>

</ncl>
```

**Listagem 1.1** Exemplo de um primeiro documento NCL.

É fácil perceber que apenas a utilização de portas no corpo do documento não é suficiente para expressar apresentações multimídia mais complexas. O autor frequentemente necessita definir, por exemplo, que o momento em que um nó deve ser exibido, na verdade, é relativo a um ou vários outros nós. Essa discussão é deixada para a seção 1.2.2.

### 1.2.1 Âncoras de conteúdo e propriedades

Um nó de mídia ou de composição também pode ter âncoras associadas. Através das âncoras é possível definir segmentos ou propriedades de um nó de mídia. Existem dois tipos de âncoras: **âncoras de conteúdo** e **âncoras de propriedade.**

Através das âncoras de conteúdo é possível marcar um segmento do nó de mídia, o que pode corresponder a um intervalo de tempo ou uma região espacial. Uma âncora de conteúdo é criada através do elemento <area>, filho do elemento <media>. Como é visto na seção 1.2.2, o sincronismo também pode ser realizado entre as âncoras de conteúdo de nós de mídia. A Listagem 1.2 mostra três exemplos de âncoras de conteúdo em um vídeo, as quais poderiam ser usadas na exibição de um vídeo com legendas. As âncoras de conteúdo marcam os intervalos de tempo no vídeo onde as legendas devem aparecer.

```
<media type = "video" id = "video1" src = "media/video1.mpg"
    descriptor = "dVideo1">

    <!--
       âncoras de conteúdo no vídeo que devem ser sincronizadas com
       a legenda
     -->
    <area id = "aVideoLegenda1" begin = "5s"  end = "9s" />
    <area id = "aVideoLegenda2" begin = "10s" end = "14s" />
    <area id = "aVideoLegenda3" begin = "5s"  end = "15s" />

</media>
```

**Listagem 1.2** Âncoras de conteúdo em um nó de mídia.

As âncoras de propriedade definem propriedades de um nó de mídia ou contexto e permitem que elas sejam manipuladas. Exemplos de propriedades são: volume de áudio e coordenadas e das dimensões de um nó de mídia que tenha representação visual

(vídeo ou imagem, por exemplo). Uma âncora de propriedade é definida pelo elemento <property>, filho do elemento <media>, <context> ou <body>. A Listagem 1.3 exemplifica a criação de quatro propriedades, relativas ao posicionamento (*top*, *left*), altura (*height*) e largura (*width*) de um nó de mídia.

```
<media type = "video" id = "video1" src = "media/video1.mpg"
    descriptor = "dVideo1">

    <!--
       âncoras de propriedade em um nó de mídia
     -->
    <property name="top"/>
    <property name="left"/>
    <property name="height"/>
    <property name="width"/>

</media>
```

**Listagem 1.3** Âncoras de propriedade em um nó de mídia.

### 1.2.2 Sincronismo baseado em elos e conectores

Uma aplicação multimídia mais elaborada requer que seja possível para o autor especificar relacionamentos entre mídias. Exemplos de relacionamentos são: quando começar a exibição da mídia X, deve terminar a exibição da mídia Y; quando terminar a exibição de X, pausa Y; e quando começar X, depois de 2 segundos inicia Y. O uso de relacionamentos é uma forma natural para um autor de uma apresentação multimídia especificar quando uma mídia deve ser exibida.

Esse relacionamento entre nós é escrito em NCL por meio de **elos** (elementos <link>). Para facilitar a definição dos elos, eles são escritos tendo como base relações hipermídia. Relações em NCL são escritas como restrições ou sentenças causais, onde uma determinada ação é realizada quando uma condição é satisfeita. Isso é especificado utilizando **conectores** (elementos <causalConnector>).

Os conectores são especificados no elemento <connectorBase> dentro do cabeçalho (elemento <head>). No perfil para a TV Digital, NCL define apenas os conectores causais. Um conector causal é definido como um conjunto de papéis de condição que devem ser satisfeitos para a ativação de elos que usam aquele conector, além de um conjunto de papéis de ações que são executadas caso elos usando aquele conector se tornem ativos. Em um conector é necessário que se defina pelo menos uma condição e uma ação. Cada condição ou ação está associada a um papel (*role*) que, posteriormente, é associado à âncora de um nó, através de ligações (elementos <bind>) nos elos. As condições e ações em NCL podem estar associadas tanto à apresentação como à seleção ou atribuição em nós de mídia.

Tanto as âncoras de conteúdo como as âncoras de propriedades são acessíveis pelos elos através do atributo *interface* no elemento <bind>. Dessa forma, é possível, por exemplo, realizar sincronismo com uma âncora de conteúdo específica daquele nó de mídia, ou ainda, consultar ou alterar o valor de uma determinada âncora de propriedade. Isso é explorado nos exemplos do minicurso, na seção seguinte.

A Listagem 1.4 exemplifica o conector "*onEndStart*", modelando o seguinte comportamento: quando a âncora que faz o papel "*onEnd*" terminar, a âncora que faz o

papel "*start*" deve ser exibida em sequência. A listagem também exemplifica o uso deste conector por um elo que associa os papéis "*onEnd*" e "*start*" ao nó "*video1*" e "*nolua*", respectivamente.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ncl id="segundoDocumentoNCL"
     xmlns="http://www.ncl.org.br/NCL3.0/EDTVProfile">

    <head>
        <!-- ... -->

        <connectorBase>
            <causalConnector id="onEndStart">
                <simpleCondition role="onEnd"/>
                <simpleAction role="start"/>
            </causalConnector>
        </connectorBase>

        <!-- ... -->
    </head>
    <body>
        <!-- ... -->

        <link xconnector="onEndStart">
            <bind component="video1" role="onEnd"/>
            <bind component="nolua" role="start"/>
        </link>

        <!-- ... -->
    </body>
</ncl>
```

**Listagem 1.4** Exemplo de um conector *onEndStart*.

A ação de atribuição (*set*) é realizada sobre âncoras de propriedades de um nó. O exemplo da Listagem 1.5 mostra o conector "*onBeginSet*", especificando a sentença: "quando uma âncora que exerce o papel "*onBegin*" iniciar, o valor passado como parâmetro a esse conector (através da variável *var*) deve ser atribuído à âncora de propriedade que realiza o papel *set*". No elo, é definido que o nó "*video1*" exerce o papel "*onBegin*" e a âncora de propriedade *playing* do nó "*nolua*" exerce o papel *set.*

Ainda na Listagem 1.5, também é exemplificado o uso de uma condição de seleção, através do conector "*onSelectionStart"*. Nesse exemplo, o elo especifica que o nó de mídia "*nolua*" iniciará sua exibição quando o nó de mídia "*video1*" for selecionado. Elos com o papel "*onSelection*" são utilizados para modelar a interatividade por parte do usuário, a qual é tratada como um caso particular de sincronismo em NCL.

Usualmente, os conectores são criados apenas uma vez, em um arquivo separado, para depois serem reusados por diversas aplicações NCL. Para facilitar este processo, a NCL permite que uma base de conectores seja importada, através do elemento <importBase>, filho de <connectorBase>. Assim, é possível definir todos os conectores em um arquivo à parte e apenas importar aquele arquivo para cada documento novo a ser criado. As ferramentas de autoria para NCL frequentemente já trazem uma base de conectores bastante ampla.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<ncl id="segundoDocumentoNCL"
     xmlns="http://www.ncl.org.br/NCL3.0/EDTVProfile">

    <head>
        <!-- ... -->

        <connectorBase>

            <causalConnector id="onBeginSet">
                <connectorParam name="var"/>
                <simpleCondition role="onBegin"/>
                <simpleAction role="set" value="$var"/>
            </causalConnector>

            <causalConnector id="onSelectionStart">
                <simpleCondition role="onSelection"/>
                <simpleAction role="start"/>
            </causalConnector>
        </connectorBase>

        <!-- ... -->
    </head>
    <body>
        <!-- ... -->

        <link xconnector="onBeginSet">
            <linkParam name="var" value="true"/>
            <bind component="video1" role="onBegin"/>
            <bind component="nolua" role="set" interface="playing"/>
        </link>

        <!-- ... -->
        <link xconnector="onSelectionStart">
            <bind component="video1" role="onSelection" />
            <bind component="nolua" role="start" />
        </link>
        <!-- ... -->
    </body>
</ncl>
```

**Listagem 1.5** Exemplo de um conector *onEndSet*.

Outras condições e ações são apresentadas, em crescente nível de dificuldade, no decorrer deste minicurso. Entretanto, foge ao escopo deste texto listar todas as ações e condições suportadas por NCL. Essa lista completa pode ser encontrada em [ABNT 2007; Soares e Barbosa 2009].

## 1.3. Tutorial NCLua

Desde o início de seu desenvolvimento, no início dos anos 90, Lua foi projetada para ser usada em conjunto com outras linguagens, não sendo comum encontrar programas escritos puramente em Lua. Nesse sentido, Lua é normalmente usada para permitir que uma aplicação principal seja estendida ou adaptada através do uso de *scripts*. A aplicação principal pode ser um videogame, onde um *script* Lua é usado para definir o comportamento de um personagem; um editor de textos, que permite que os textos sejam acessados e modificados por *scripts* Lua; ou, de maneira mais geral, aplicações que usam Lua em *scripts* de configuração. Esse tipo de uso caracteriza Lua como uma linguagem de *scripts* no seu sentido mais puro. O próprio nome da linguagem, Lua, remete à idéia de uma linguagem satélite.

Lua é uma linguagem de fácil aprendizado, que combina sintaxe procedural com declarativa, com poucos comandos primitivos. Dessa característica resulta uma implementação leve e muito eficiente quando comparada com linguagens de propósitos similares. Lua também apresenta um alto grau de portabilidade, podendo ser executada com todas as suas funcionalidades em diversas plataformas, tais como computadores pessoais, celulares, sistemas embarcados e consoles de *videogames*.

As características de Lua mencionadas — simplicidade, eficiência e portabilidade —, além de sua licença livre, casam perfeitamente com o cenário de TV Digital. Uma linguagem simples é bem-vinda onde é comum equipes formadas não só por programadores, mas também por designers e produtores de conteúdo. A portabilidade é importante quando o *middleware* deve ser desenvolvido para dispositivos com características conflitantes tais como celulares e *set-top boxes*. A eficiência e tamanho da linguagem requerem menos custos com *hardware*. Já a licença livre de *royalties* reduz a custo zero a adoção do interpretador por unidade produzida. Um indicador de como a linguagem Lua se adapta bem a esse tipo de cenário é a liderança de Lua como linguagem de *script* em *videogames*, nicho que compartilha as mesmas características descritas.

Uma apresentação mais detalhada da sintaxe e funcionalidades de Lua fica fora do escopo deste minicurso. Explicaremos os conceitos da linguagem necessários conforme forem utilizados nos exemplos deste capítulo.

### 1.3.1. Extensões de NCLua

Para se adequar ao ambiente de TV Digital e se integrar à NCL, a linguagem Lua foi estendida com novas funcionalidades. Por exemplo, um NCLua precisa se comunicar com o documento NCL para saber quando o seu objeto `<media>` correspondente é iniciado por um elo. Um NCLua também pode responder a teclas do controle remoto, ou desenhar livremente dentro da região NCL a ele destinada. Essas funcionalidades são específicas da linguagem NCL e, obviamente, não fazem parte da biblioteca padrão de Lua. O que diferencia um NCLua de um programa Lua puro é o fato de o NCLua ser controlado pelo documento NCL no qual está inserido, além de utilizar as extensões descritas a seguir.

Além da biblioteca padrão de Lua, os seguintes módulos estão disponíveis para *scripts* NCLua:

- Módulo *event*: permite que objetos NCLua se comuniquem com o documento NCL e outras entidades externas (tais como controle remoto e canal de interatividade).
- Módulo *canvas*: oferece funcionalidades para desenhar objetos gráficos na região do NCLua.
- Módulo *settings*: oferece acesso às variáveis definidas no objeto *settings* do documento NCL (objeto do tipo “application/x-ncl-settings”).
- Módulo *persistent*: exporta uma tabela com variáveis persistentes entre execuções de objetos imperativos.

A norma ABNT NBR 15606-2:2007 [ABNT 2007] e H.761 [ITU-T 2009] lista detalhadamente todas as funções suportadas por cada módulo. A Seção 1.4 apresenta um guia de consulta rápida aos dois primeiros módulos apresentados acima.

As seguintes funções da biblioteca padrão de Lua são dependentes de plataforma e por isso não estão disponíveis para *scripts* NCLua:

- No módulo *package*: a função *loadlib*.
- No módulo *io*: todas as funções.
- No módulo *os*: as funções *clock*, *execute*, *exit*, *getenv*, *remove*, *rename*, *tmpname* e *setlocale*.
- No módulo *debug*: todas as funções.

### 1.3.2. Programação Orientada a Eventos

O *Middleware* Ginga possui um modelo próprio de execução e comunicação de objetos imperativos embutidos em documentos NCL. No caso de objetos NCLua, os mecanismos de integração com o documento NCL se fazem através do paradigma de programação orientada a eventos.

Nesse paradigma, é através da difusão e recepção de eventos que são feitas a comunicação com o documento NCL e toda interação com entidades externas à aplicação, tais como o canal de interatividade, controle remoto e temporizadores. O módulo *event* de NCLua é usado para esse fim e seu entendimento é essencial para desenvolver qualquer aplicação que utilize objetos NCLua.

A Figura 1.2 ilustra ao centro um NCLua, envolto por diversas entidades com as quais ele pode interagir. Para se comunicar com um NCLua, uma entidade externa deve inserir um evento na fila indicada na figura, que é então redirecionado às funções tratadoras de eventos, definidas pelo programador do *script* NCLua. Enquanto cada tratador processa um evento (um de cada vez), nenhum outro evento da fila é tratado. Sendo assim, fica a cargo do programador escrever tratadores que executem o mais rápido possível, de maneira a evitar o congestionamento da fila. Um NCLua também pode se comunicar com entidades externas postando eventos de dentro de seus tratadores, como mostram as setas saindo do NCLua[1].

A principal vantagem do uso do paradigma de eventos é a sua característica de acoplamento fraco entre as entidades do sistema. Como se pode observar pela figura, a remoção ou adição de uma entidade não acarreta mudanças internas em nenhuma outra entidade. Essa característica vai ao encontro dos requisitos de mínima intrusão do ambiente de autoria.

[1] A fila de eventos é controlada pelo sistema e não é visível a um NCLua.

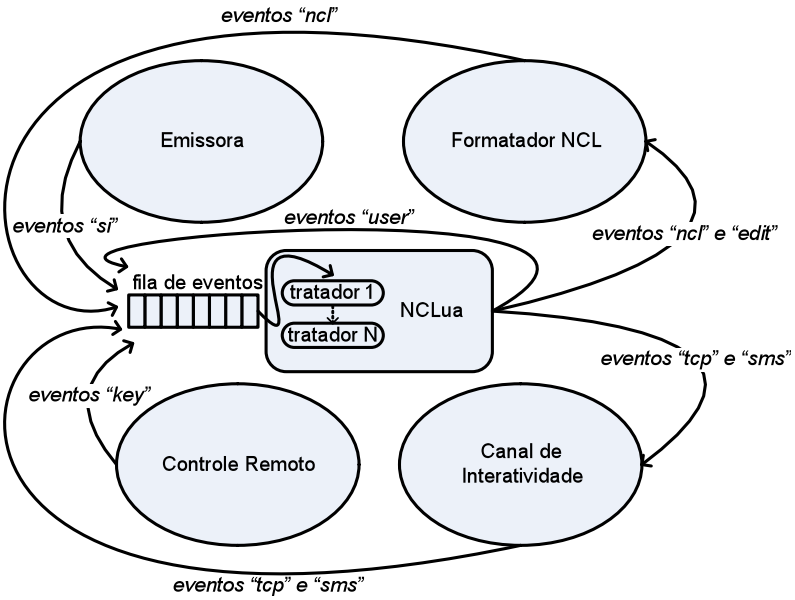


**Figura 1.2** Paradigma de programação orientado a Eventos.

Para ser informado quando eventos externos são recebidos, um NCLua deve registrar pelo menos uma função de tratamento em seu corpo através de uma chamada à função *event.register* (o nome da função a ser registrada é qualquer). O código de um NCLua segue uma estrutura comum a todos os *scripts*, como a seguir:

```
...                              -- código de inicialização
function tratador (evt)
   ...                           -- código de um tratador
end

event.register(tratador)         -- registro de pelo menos um tratador
```

**Listagem 1.6** Registro da função de tratamento de eventos em NCLua.

O código de inicialização, a definição do tratador e seu registro são executados antes que o documento NCL (ou qualquer entidade externa ao *script*) sinalize qualquer evento ao NCLua, inclusive o de início de apresentação do objeto. Após esse processo de carga do *script*, efetuado pelo sistema, apenas o código do tratador é chamado toda vez que ocorre um evento externo. O código de inicialização pode ser utilizado para criar objetos e funções auxiliares que serão usadas pelos tratadores.

Eventos são representados por tabelas Lua com chaves e valores descrevendo seus atributos. Como exemplo, a função tratadora pode receber um evento indicando que a tecla vermelha do controle remoto foi pressionada pelo telespectador (parâmetro *evt* do tratador, na Listagem 1.7).

```
evt = {
  class  = 'key',
  type   = 'press',
  key    = 'RED'
}
```

**Listagem 1.7** Representação de evento em NCLua

A função *event.post* é utilizada para que um NCLua poste eventos e possa, por exemplo, enviar dados pelo canal de interatividade ou sinalizar seu estado ao documento NCL. No exemplo a seguir (Listagem 1.8), o NCLua sinaliza ao documento o seu fim natural.

```
event.post {
  class  = 'ncl',
  type   = 'presentation',
  action = 'stop',
}
```

**Listagem 1.8** Exemplo de evento postado por um NCLua para sinalizar ao documento NCL o seu fim natural.

Como um NCLua (por meio de seus tratadores) deve executar rapidamente, a função de envio de eventos nunca aguarda o retorno de um valor. Caso o destino necessite retornar uma informação ao NCLua, deve fazê-lo através do envio de um novo evento.

#### 1.3.2.1. Classes de Eventos

O campo *class* de uma tabela que representa um evento é obrigatório e tem a finalidade de separar os eventos em categorias. A classe identifica não somente a origem de eventos passados aos tratadores, mas também o seu destino, caso o evento seja gerado e postado por um *script* NCLua.

As seguintes classes de eventos estão definidas:

- Classe *ncl*: Usada na comunicação entre um NCLua e o documento NCL que contém o objeto de mídia.
- Classe *key*: Representa o pressionamento de teclas do controle remoto pelo usuário.
- Classe *tcp*: Permite acesso ao canal de interatividade por meio do protocolo TCP.
- Classe *sms*: Usada para envio e recebimento de mensagens SMS em dispositivos móveis.
- Classe *edit*: Permite que os comandos de edição ao vivo sejam disparados a partir de *scripts* NCLua.
- Classe *si*: Fornece acesso a um conjunto de informações multiplexadas em um fluxo de transporte e transmitidas periodicamente por difusão.
- Classe *user*: Através dessa classe, aplicações podem estender sua funcionalidade criando seus próprios eventos.

Observe, pela Figura 1.2, que há eventos apenas de entrada, apenas de saída, e eventos que são usados em ambos os sentidos.

O modelo orientado a eventos de NCLua foi projetado para suportar outras entidades externas, estendendo o modelo básico, bastando para isso definir novas classes de eventos [Soares, et. al, 2007].

### 1.3.3. Interagindo com o Documento NCL

Assim como qualquer objeto de mídia, um NCLua interage com o documento NCL através de elos.

Em elos que acionam um NCLua, a condição satisfeita faz com que o NCLua receba um evento da classe *ncl* descrevendo a ação a ser tomada. No trecho da Listagem 1.9, quando o elo for disparado com o início de "*videoId*", o tratador de eventos de NCLua receberá o evento no código do objeto NCLua.

```
<link xconnector="onBeginStart">
    <bind role="onBegin" component="videoId"/>
    <bind role="start"   component="luaId"/>
</link>
```

**Arquivo NCL que contém o objeto NCLua**

```
-- Evento recebido pelo tratador do NCLua no
--   disparo do elo:
evt = {
    class  = 'ncl',
    type   = 'presentation',
    action = 'start',
}
```

**Arquivo NCLua**

**Listagem 1.9** Exemplo de códigos NCL e NCLua que tratam um evento de apresentação de um objeto NCL.

Já em elos cuja condição depende de um NCLua, a ação do elo será disparada quando o NCLua sinalizar o evento que casa com a condição esperada. No trecho da Listagem 1.10, quando o NCLua sinalizar o evento indicado (no caso, o término da apresentação do objeto NCLua), o elo será disparado e a imagem exibida.

```
<link xconnector="onBeginStart">
    <bind role="onEnd" component="luaId"/>
    <bind role="start" component="imagemId"/>
</link>
```

**Arquivo NCL que contém o objeto NCLua**

```
-- O elo acima será disparado quando o evento a seguir
-- for postado pelo NCLua 'luaId':
event.post {
    class  = 'ncl',
    type   = 'presentation',
    action = 'stop',
}
```

**Arquivo NCLua**

**Listagem 1.10** Elo disparado pelo código do objeto NCLua

O tipo "*presentation*" indica que os eventos se referem à apresentação do NCLua. Como nenhuma âncora foi especificada, é assumida a âncora de conteúdo principal.

Como os dois exemplos indicam, a interação de um NCLua com o documento NCL se dá sempre através da classe de eventos *ncl*, seja para receber instruções do formatador, seja para notificar o estado de suas âncoras.

**Exemplo 1 - Ciclo de Vida de Objetos NCLua**

Este exemplo apresenta uma aplicação com a finalidade de ilustrar o modelo de execução de objetos imperativos NCLua em documentos NCL.

Ao iniciar a aplicação, três objetos NCLua são iniciados, cada qual com um comportamento interno diferente:

- O primeiro NCLua executa indefinidamente.
- O segundo NCLua termina a si próprio assim que é iniciado.
- O terceiro NCLua termina a si próprio três segundos após ser iniciado.

Associados a cada NCLua, no documento NCL, há um botão verde e um vermelho, indicando se o NCLua está ocorrendo ou terminado, respectivamente. As visões temporal e espacial da aplicação, mostradas na Figura 1.3, refletem o comportamento descrito.

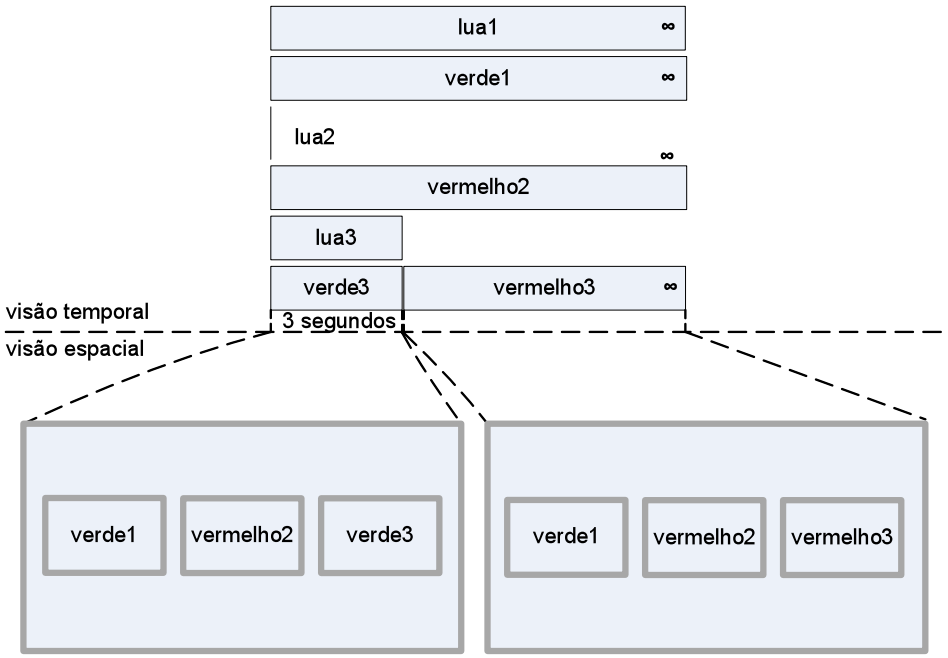


**Figura 1.3** Visão temporal e espacial do Exemplo 1.

A Figura 1.4 exibe a visão estrutural do documento NCL. O primeiro NCLua é ligado aos outros por meio de um elo "*onBeginStart*". Como o primeiro NCLua está ligado à porta de entrada do documento, os três objetos iniciam juntamente com a aplicação. Cada NCLua se liga por meio de um elo "*onBeginStart*" com seu respectivo botão verde, para que eles sejam exibidos com o início de cada NCLua. Para esconder

seu respectivo botão verde e exibir o vermelho, cada NCLua também utiliza um elo "*onEndStopStart*" com seus botões, conforme mostra a figura.

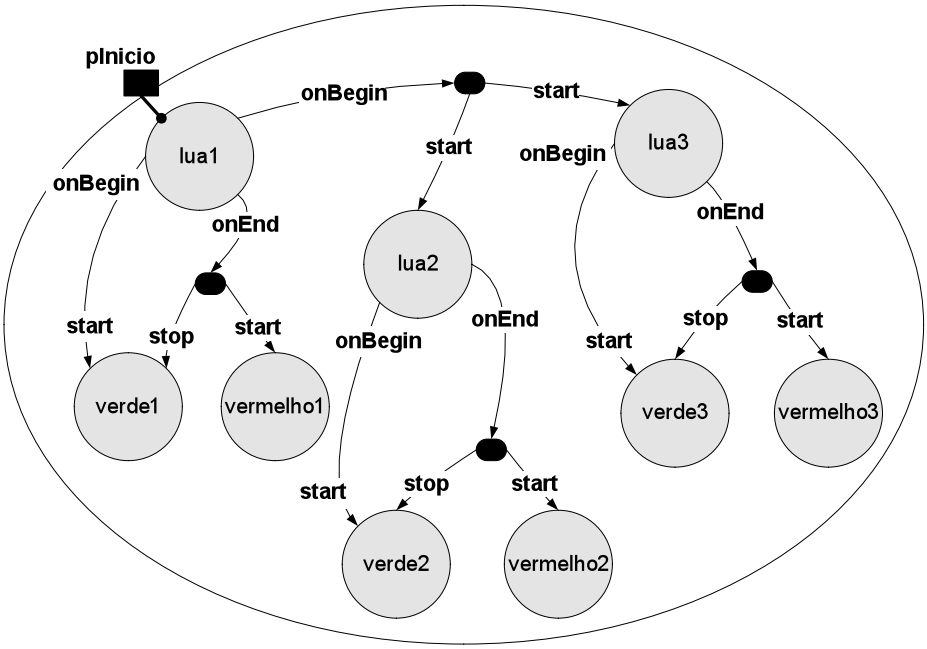


**Figura 1.4** Visão estrutural do Exemplo 1.

O documento NCL é responsável por definir e iniciar os três objetos NCLua, assim como pela "cola lógica" entre cada NCLua e seus botões correspondentes, conforme definido na Listagem 1.11.

```
<body>
    <!-- INICIA primeiro o NCLua -->
    <port id="pInicio" component="lua1"/>

    <!-- MIDIAS -->
    <media id="lua1"  src="1.lua"/>
    <media id="lua2"  src="2.lua"/>
    <media id="lua3"  src="3.lua"/>
    <media id="verde1"    src="media/verde1.png" descriptor="ds1"/>
    <media id="vermelho1" src="media/verm1.png"  descriptor="ds1"/>
    <media id="verde2"    src="media/verde2.png" descriptor="ds2"/>
    <media id="vermelho2" src="media/verm2.png"  descriptor="ds2"/>
    <media id="verde3"    src="media/verde3.png" descriptor="ds3"/>
    <media id="vermelho3" src="media/verm3.png"  descriptor="ds3"/>

    <!-- AO INICIAR lua1 -> INICIA lua2/lua3 -->
    <link xconnector="onBeginStart">
    <bind role="onBegin" component="lua1"/>
    <bind role="start"   component="lua2"/>
    <bind role="start"   component="lua3"/>
    </link>

    <!-- AO INICIAR lua[N] -> INICIA verde[N] -->
    <link xconnector="onBeginStart">
    <bind role="onBegin" component="lua1"/>
    <bind role="start"   component="verde1"/>
    </link>
    ...  <!-- Código idêntico para os outros NCLua -->
```

```
    <!-- AO TÉRMINO DE lua[N] -> PÁRA verde[N] | INICIA vermN -->
    <link xconnector="onEndStopStart">
    <bind role="onEnd" component="lua1"/>
    <bind role="stop"  component="verde1"/>
    <bind role="start" component="vermelho1"/>
    </link>
    ...  <!-- Código idêntico para os outros NCLua -->
</body>
```

**Listagem 1.11** Código NCL parcial do Exemplo 1.

Note como neste exemplo o NCL não aciona o término de nenhum objeto NCLua. Esse papel fica a cargo de cada *script* NCLua, como mostrado a seguir.

- O primeiro NCLua é um *script* vazio (sem nenhuma linha de código). Em particular, como não possui um tratador de eventos, nunca sinaliza o seu término para o documento NCL. O efeito visual é a exibição permanente do primeiro botão verde.

```
-- 1.lua
-- vazio
```

**Listagem 1.12** Código do arquivo 1.lua do Exemplo 1.

- O segundo NCLua registra um tratador de eventos que gera seu fim natural ao receber um "*start*" do documento NCL. Visualmente, instantaneamente após a exibição do segundo botão verde, é exibido o botão vermelho correspondente (o botão verde pode nem ser visto).

```
-- 2.lua:
function tratador (evt)
  if (evt.class == 'ncl') and (evt.type == 'presentation')
     and (evt.action == 'start') then
    event.post {
        class  = 'ncl',
        type   = 'presentation',
        action = 'stop' }
  end
end
event.register(tratador)
```

**Listagem 1.13** Código do arquivo 2.lua do Exemplo 1.

Tão logo o evento indicando seu início é recebido, o NCLua posta um evento para sinalizar o seu fim natural.

- O terceiro NCLua registra um tratador de eventos que cria um temporizador de três segundos que, por sua vez, gera seu fim natural ao expirar. Como efeito visual, temos a exibição do terceiro botão verde e, após três segundos, a mudança para vermelho.

```
-- 3.lua:
function tratador (evt)
  if (evt.class == 'ncl') and
     (evt.type == 'presentation') and
     (evt.action == 'start') then
```

```
    event.timer(3000,
        function()
            event.post {
                class  = 'ncl',
                type   = 'presentation',
                action = 'stop'
            }
        end)
  end
end
event.register(tratador)
```

**Listagem 1.14** Código do arquivo 3.lua do Exemplo 1.

O temporizador de três segundos (3000 milissegundos) é criado assim que o evento de início é recebido, passando a função que deve ser executada quando o temporizador expira. Essa função posta um evento idêntico ao do segundo NCLua para sinalizar seu fim natural.

Enquanto executa indefinidamente, o primeiro NCLua não consome recursos e poderia responder a eventos (apesar de não o fazer nesse caso por não possuir um tratador para tal fim). O mesmo ocorre com o terceiro NCLua enquanto aguarda os três segundos para terminar.

#### 1.3.3.1. Eventos em Âncoras de Conteúdos e Propriedades

No exemplo anterior, apenas a âncora de conteúdo principal de cada NCLua foi acionada. No entanto, âncoras de conteúdo específicas e propriedades também podem ser relacionadas entre o documento NCL e o objeto NCLua.

Dados os tipos de eventos NCL suportados, o campo *type* de um evento da classe *ncl* pode assumir os valores "*presentation*" ou "*attribution*", conforme o atributo *eventType* definido nos conectores NCL. Eventos do tipo "*selection*" são complementados pelo campo *key*.

**Eventos do tipo "presentation"**

Eventos de apresentação estão associados à apresentação de âncoras de conteúdo, que são identificadas pelo campo *label* do evento. O campo *action* indica a ação a ser realizada ou sinalizada pelo NCLua, dependendo se este está recebendo ou gerando o evento.

Um evento de apresentação possui a seguinte estrutura:

- *class*: *'ncl'*
- *type*: *'presentation'*
- *label*: [*string*] – Rótulo da âncora associada ao evento.
- *action*: [*string*] – Pode assumir os seguintes valores: 'start', 'stop', 'abort', 'pause' e 'resume'.

**Eventos do tipo "attribution"**

Eventos de atribuição estão associados às propriedades do objeto NCLua, que são identificadas pelo campo *name*.

O campo *value* é preenchido com o valor a ser atribuído à propriedade e é sempre uma string, uma vez que vem de um atributo NCL. A ação de "*start*" em um evento de atribuição corresponde ao papel "*set*" de um elo NCL.

Um evento de atribuição possui a seguinte estrutura:

- *class*: *'ncl'*
- *type*: *'attribution'*
- *name*: [*string*] Nome da propriedade associada ao evento.
- *action*: [*string*] Pode assumir os seguintes valores: 'start', 'stop', 'abort', 'pause' e 'resume'.
- *value*: [*string*] Novo valor a ser atribuído à propriedade.

O campo *action* de um evento *ncl* (seja ele de apresentação ou atribuição) pode assumir os valores correspondentes aos seus tipos, como mostrado em [ABNT 2007]. No entanto, o nome das transições é usado sem o "s" final ("*starts*" vira "*start*"), de maneira a unificar a sintaxe para eventos recebidos ou sinalizados pelo NCLua.

**Exemplo 2 - Contador de Cliques**

Vamos supor uma aplicação NCL que exibe um botão "Clique Aqui" em quatro momentos diferentes. Se o usuário selecioná-lo com o controle remoto por pelo menos três vezes, ao final da apresentação é exibida a imagem "Você Ganhou", caso contrário é exibido a imagem "Você Perdeu". A Figura 1.5 mostra as visões temporal e espacial da aplicação.

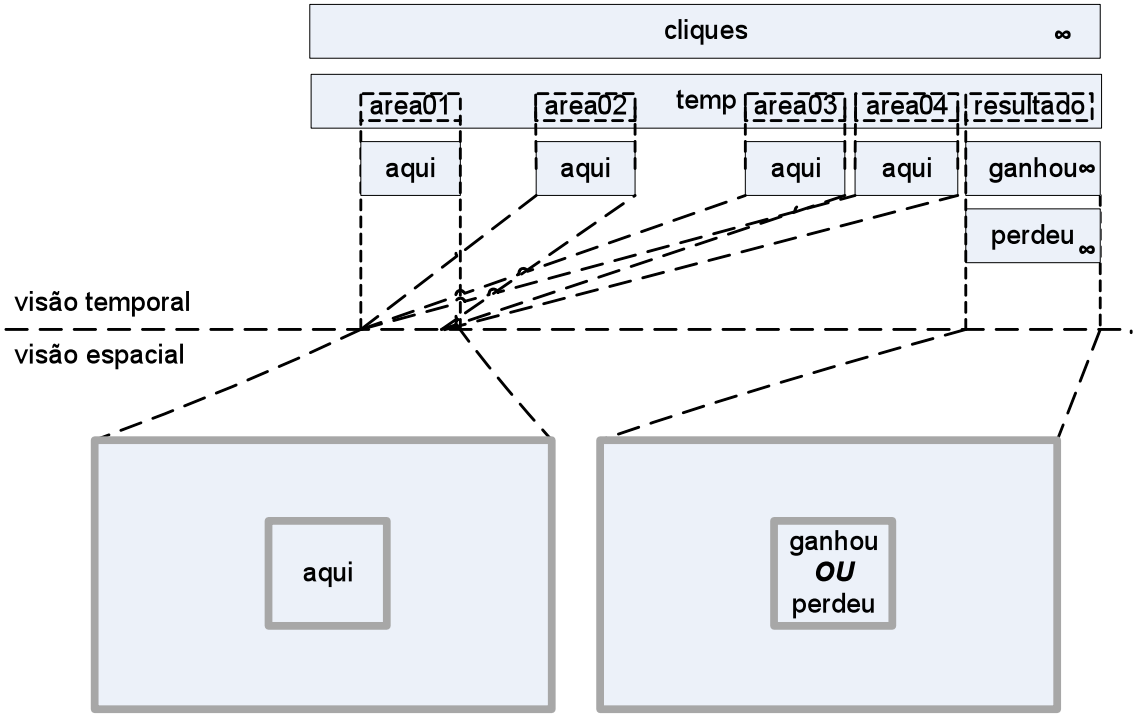


**Figura 1.5** Visões temporal e espacial do Exemplo 2.

Não é possível fazer a contagem de cliques puramente em NCL de forma simples, uma vez que não há suporte a expressões aritméticas na linguagem. Neste

exemplo, usaremos um NCLua para contar e armazenar o número de cliques em uma propriedade, que será consultada ao final para determinar o resultado.

O documento NCL mostrado a seguir define um temporizador ("*temp*") com quatro âncoras temporais ("*area01*" até "*area04*") durante as quais o botão "Clique aqui" é exibido. O temporizador também define uma âncora temporal para exibir o resultado após o botão ser exibido quatro vezes. Além do temporizador e dos botões, o documento define um NCLua responsável pela contagem dos cliques e exporta a propriedade "*contador*" para manter esse valor. A Figura 1.6 mostra a visão estrutural da aplicação.

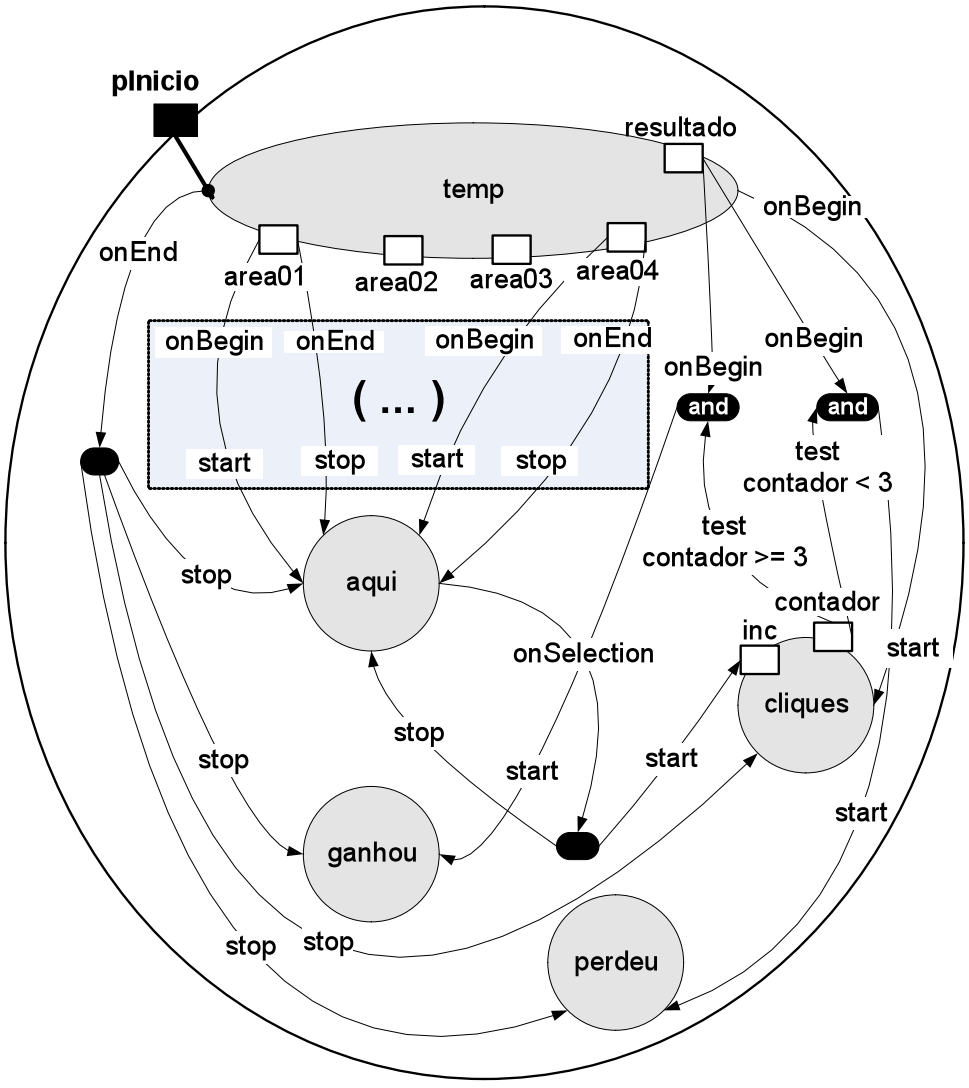


**Figura 1.6** Visão estrutural do Exemplo 2.

A porta de entrada da aplicação é o temporizador, que também dispara o NCLua por meio de um elo "*onBeginStart*". Cada âncora de exibição do botão "Clique Aqui" possui um elo "*onBeginStart*" e "*onEndStop*" para o botão. Toda vez que o botão é selecionado pelo usuário (o que só pode acontecer enquanto está sendo exibido), a âncora "*inc*" do NCLua é iniciada e o botão é escondido, conforme o elo "*onSelectionStopStart*" saindo do próprio botão. O tratamento dado à âncora "*inc*" e à propriedade "*contador*" são especificados no código do NCLua. Ao final da âncora "*resultado*" do temporizador, dois elos testam se o valor da propriedade "*contador*" é

maior ou menor que três cliques. Dependendo do resultado, a imagem “Você ganhou” ou “Você perdeu” é exibida.

O documento NCL para a aplicação é exibido na Listagem 1.15.

```
<body>
  <!-- INÍCIO PELO TEMPORIZADOR -->
  <port id="pInicio" component="temp"/>
   <!-- TEMPORIZADOR -->
  <media id="temp" type="application/x-ginga-time">
    <!-- ÂNCORAS PARA EXIBIR O BOTÃO "CLIQUE AQUI" -->
    <area id="area01" begin="3s"  end="6s"/>
    <area id="area02" begin="10s" end="13s"/>
    <area id="area03" begin="17s" end="20s"/>
    <area id="area04" begin="24s" end="27s"/>
    <!-- ÂNCORA PARA EXIBIR O RESULTADO -->
    <area id="resultado" begin="35s"/>
  </media>

  <!-- NCLua -->
  <media id="cliques" src="cliques.lua">
    <area id="inc" label="inc"/>
    <property name="contador"/>
  </media>

  <!-- "CLIQUE AQUI"/"VOCÊ GANHOU"/"VOCÊ PERDEU" -->
  <media id="aqui"   descriptor="dsBotao" src="media/aqui.jpg"/>
  <media id="ganhou" descriptor="dsBotao"
         src="media/ganhou.jpg"/>
  <media id="perdeu" descriptor="dsBotao"
         src="media/perdeu.jpg"/>

  <!-- TEMPORIZADOR -> NCLua -->
  <link xconnector="onBeginStart">
    <bind role="onBegin" component="temp"/>
    <bind role="start"   component="cliques"/>
  </link>

  <!-- AREA[N] -> BOTÃO "CLIQUE AQUI" -->
  <link xconnector="onBeginStart">
    <bind role="onBegin" component="temp"
          interface="area01"/>
    <bind role="start" component="aqui"/>
  </link>
  ... <!-- Código idêntico para as outras âncoras -->

  <!-- AO FINAL DE AREA[N] -> PÁRA BOTÃO "CLIQUE AQUI" -->
  <link xconnector="onEndStop">
    <bind role="onEnd" component="temp"
          interface="area01"/>
    <bind role="stop" component="aqui"/>
  </link>
  ... <!-- Código idêntico para as outras âncoras -->

  <!-- AO SELECIONAR "CLIQUE AQUI" -> CLIQUES.INC -->
  <link xconnector="onSelectionStopStart">
    <bind role="onSelection" component="aqui"/>
    <bind role="stop"        component="aqui"/>
    <bind role="start"       component="cliques"
                             interface="inc"/>
  </link>

  <!-- TESTE DO RESULTADO -->
  <link xconnector="onCondGteBeginStart">
```

```
  <linkParam name="var" value="3"/>
    <bind role="onBegin  component="temp"
                         interface="resultado"/>
    <bind role="attNodeTest" component="cliques"
                             interface="contador"/>
    <bind role="start" component="ganhou"/>
  </link>
  <link xconnector="onCondLtBeginStart">
    <linkParam name="var" value="3"/>
    <bind role="onBegin" component="temp"
                         interface="resultado"/>
    <bind role="attNodeTest" component="cliques"
                             interface="contador"/>
    <bind role="start" component="perdeu"/>
  </link>
</body>
```

**Listagem 1.15** Código NCL parcial do Exemplo 2.

O código do NCLua deve lidar, em sua função tratadora de eventos, com as duas interfaces com o documento NCL: a âncora "*inc*" e a propriedade "*contador*". A Listagem 1.16 apresenta o código do *script*.

```
local contador = 0
function tratador (evt)
   contador = contador + 1
   local evtContador = {
      class = 'ncl',
      type  = 'attribution',
      name  = 'contador',
      value = contador
   }
   evtContador.action = 'start'
   event.post(evtContador)
   evtContador.action = 'stop'
   event.post(evtContador)
   event.post {
      class = 'ncl',
      type = 'presentation',
      label = 'inc',
      action = 'stop'
   }
end
event.register(tratador)
```

**Listagem 1.16** Código NCLua do Exemplo 2.

O *script* começa declarando uma variável para guardar a contagem de cliques. Essa variável não possui relação direta com a propriedade de mesmo nome do NCLua definida no documento NCL[2]. O manuseio da propriedade é feito através da postagem de eventos, conforme apresentado a seguir.

A função "*tratador*" é então definida e registrada (na última linha do código). Os parâmetros extras na chamada à função *register* servem para filtrar apenas os eventos desejados, no caso, eventos *ncl* de início de apresentação da âncora "*inc*". Quando o

[2] No entanto, quando uma variável deve refletir o valor de uma propriedade e vice-versa, é conveniente utilizar o mesmo nome nas suas definições.

botão é selecionado, iniciando a âncora "*inc*", a função tratadora é executada. A função incrementa o contador interno e posta um evento de início de atribuição (*action='start'*), para indicar a mudança na propriedade "*contador*".

Note que é necessário sinalizar também o fim da atribuição, fazendo com que a máquina de estados da propriedade "*contador*" volte ao estado "*sleeping*" e futuras atribuições surtam efeito. Pelo mesmo motivo, é necessário sinalizar o término da âncora "*inc*", postando o evento de "*stop*" da apresentação, no fim da função.

O NCLua não tem como saber o momento exato em que a propriedade "*contador*" será consultada pelo documento NCL — por isso, sempre que incrementa o valor, também atualiza a propriedade.

### 1.3.4. Desenhando na Região do Objeto

Quando um NCLua é carregado, o formatador NCL cria um objeto gráfico para representar a região associada à mídia NCLua no documento. Esse objeto é pré-carregado na variável global "*canvas*" do *script*, e é através dela que todas as operações gráficas são efetuadas. Caso o objeto de mídia NCLua não esteja associado a nenhuma região, então o valor do "*canvas*" será igual a "*nil*".

Como exemplo, caso a região a seguir esteja associada ao objeto NCLua, a variável "*canvas*" do *script* irá representá-la, com tamanho 300x100 e posicionada em (20,200).

```
<region id="luaRegion" width="300" height="100" top="200" left="20"/>
```

**Listagem 1.17** Exemplo de região NCL.

Existem diversas operações gráficas suportadas pelo módulo *canvas*, tais como desenho de linhas, textos e imagens. A lista completa de operações pode ser consultada em [ABNT 2007]. O exemplo a seguir cria um novo *canvas* representando a imagem passada, desenhando-a centralizada na região e acompanhada de uma legenda. A Figura 1.7 exibe o resultado visual da execução do *script* da Listagem 1.18.


**Figura 1.7** Resultado da execução do *script* da Listagem 1.17.

```
local regLarg, regAlt = canvas:attrSize()

local img = canvas:new('ginga.png')
local imgLarg, imgAlt = img:attrSize()
local imgX = (regLarg - imgLarg) / 2
local imgY = (regAlt - imgAlt) / 2
canvas:compose(imgX, imgY, img)

local txt = 'TV Digital se faz com Ginga'
canvas:attrFont('vera', 14)
local txtLarg, txtAlt = canvas:measureText(txt)
local txtX = (regLarg - txtLarg) / 2
local txtY = imgY + imgAlt + 2
canvas:attrColor('white')
canvas:drawText(txtX, txtY, txt)

canvas:flush()
```

**Listagem 1.18** *Script* ilustrando o uso do *canvas*.

O *script* da Listagem 1.18 começa guardando as dimensões da região inteira do NCLua nas variáveis *regLarg* e *regAlt* com a chamada à função *canvas:attrSize()*, que retorna a largura e altura do *canvas*. Em seguida, o método *canvas:new()* recebe uma imagem e retorna um novo *canvas*, "*img*", que a representa. As dimensões da imagem são então recuperadas e utilizadas para calcular a posição centralizada na qual a imagem é desenhada na região (*imgX* e *imgY*). A chamada *canvas:compose(imgX,imgY,img)* sobrepõe a imagem do Ginga à região do NCLua na posição informada[3].

Para desenhar a legenda do texto, primeiramente é medido o tamanho que o texto ocupa no *canvas*, com a chamada à *canvas:measureText()*. A posição horizontal centralizada do texto é calculada de maneira similar à da imagem. Já sua posição vertical é calculada um pouco abaixo da imagem. Por fim, a chamada à *canvas:attrColor('white')* altera o atributo de cor para branco, afetando as futuras operações sobre o *canvas*, para então desenhar o texto na posição calculada. Apenas com a chamada à função *canvas:flush()* as operações gráficas descritas são de fato efetuadas.

Como se pode deduzir ao observar o exemplo, um objeto *canvas* guarda em seu estado diversos atributos sobre os quais as primitivas gráficas devem operar. Os atributos são acessados através de métodos de prefixo *attr* acompanhado do nome do atributo (e.g. *attrColor*), e servem tanto para leitura, quando chamados sem parâmetros (como é o caso da chamada à *attrSize*() ), como para escrita, quando chamados com os novos parâmetros (como é o caso da chamada à *attrColor('white')* ).

A referência completa do módulo *canvas*, com todos os atributos e operações gráficas suportados, pode ser encontrada em [ABNT 2007]. A Seção 1.4.2 pode servir como guia de consulta rápida para tal módulo.

**Exemplo 3 – Gráficos e Controle Remoto**

[3] As coordenadas passadas para todos os métodos são sempre relativas ao canto superior esquerdo do canvas (0,0), como é comum em sistemas gráficos.

Este exemplo apresenta um jogo bastante simples, onde são exibidos os logotipos das linguagens Lua e NCL. O usuário deve mover com o controle remoto o logotipo de Lua até o de NCL, quando é então exibida uma imagem indicando o fim do jogo. O exemplo explora os eventos da classe *key* e o pacote gráfico *canvas*. A Figura 1.8 mostra as visões temporal e espacial da aplicação.

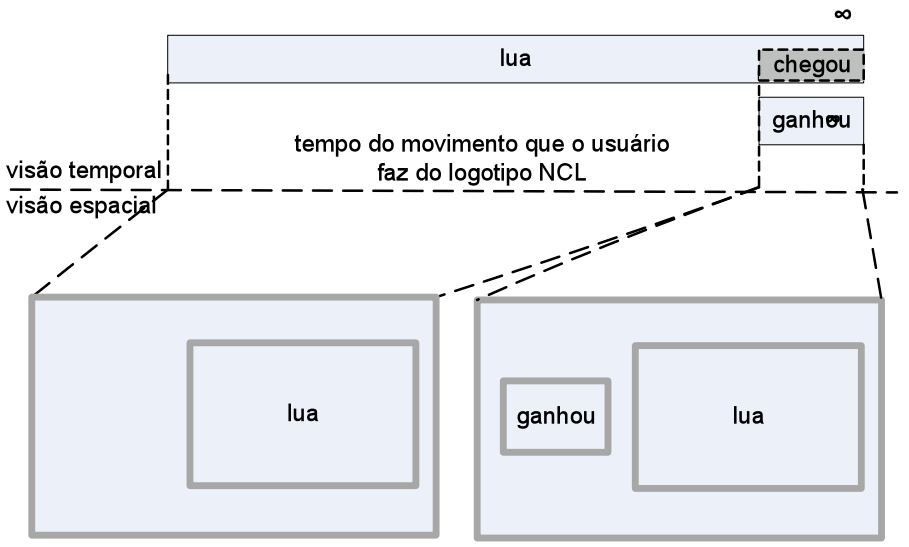


**Figura 1.8** Visões espacial e temporal do Exemplo 3.

O documento NCL possui como mídias apenas o objeto NCLua ("*lua*") com o jogo e a imagem "Você ganhou" ("*ganhou*"), que é exibida quando o logotipo de Lua encontra o de NCL (acontecimento representado pela âncora "*chegou*" do NCLua). O jogo inicia assim que o documento é carregado. A Figura 1.9 mostra a visão estrutural da aplicação.

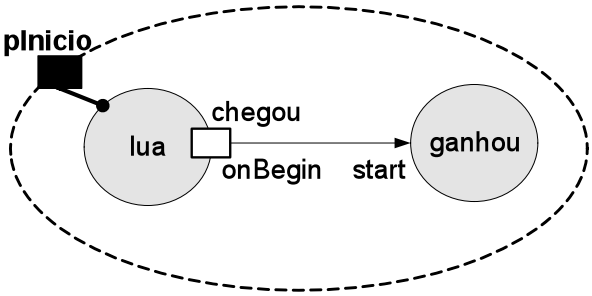


**Figura 1.9** Visão estrutural do Exemplo 3.

O código NCL é bem simples, como definido na Listagem 1.19.

```
<body>
  <port id="pInicio" component="lua"/>
  <media id="lua" src="jogo.lua" descriptor="dsLua">
    <area id="chegou"/>
  </media>
     <media id="ganhou" src="media/ganhou.jpg" descriptor="dsGanhou"/>

  <link xconnector="onBeginStart">
    <bind role="onBegin" component="lua" interface="chegou"/>
    <bind role="start" component="ganhou"/>
    </link>
```

```
</body>
```

**Listagem 1.19** Código NCL parcial do Exemplo 3.

Nos exemplos anteriores, o código da aplicação estava concentrado no documento NCL. Nesse exemplo, praticamente todo o código está dentro do NCLua. As primeiras linhas do código NCLua criam duas tabelas para representar os logotipos de Lua e NCL (Listagem 1.20).

```
local img = canvas:new('media/lua.png')
local larg, alt = img:attrSize()
local logoLua = { canvas=img, x=10,y=10,
                  larg=larg,alt=alt }

local img = canvas:new('media/ncl.png')
local larg, alt = img:attrSize()
local logoNcl = { canvas=img, x=100,y=10,
                  larg=larg,alt=alt }
```

**Listagem 1.20** Primeira parte do código NCLua do Exemplo 3.

Conforme a Listagem 1.19, o logotipo de Lua é representado pela tabela *logoLua*, guardando o *canvas* com a imagem *lua.png*, as posições x e y onde ela deve ser desenhada, e sua largura e altura. A tabela *logoNcl* guarda os mesmos atributos (mas para a imagem *ncl.png*) para representar o logotipo de NCL. Em seguida, definimos na Listagem 1.21 a função de redesenho da tela, chamada durante a execução do NCLua toda vez que o usuário movimenta o logotipo de Lua. Sempre que chamada, a função *redraw* desenha um retângulo preto ocupando a região inteira (para limpá-la), e em seguida compõe os *canvas* das duas tabelas *logoNcl* e *logoLua* sobre o *canvas* principal, em suas posições correntes.

```
function redraw ()
    canvas:attrColor('black')
    canvas:drawRect('fill', 0,0, canvas:attrSize())
    canvas:compose(logoNcl.x, logoNcl.y, logoNcl.canvas)
    canvas:compose(logoLua.x, logoLua.y, logoLua.canvas)
    canvas:flush()
end
```

**Listagem 1.21** Segunda parte do código NCLua do Exemplo 3.

Por fim, conforme a Listagem 1.22, definimos a função tratadora de eventos, responsável por responder às teclas do controle remoto e por sinalizar, ao documento NCL, o momento em que os logotipos se sobrepõem.

```
function tratador (evt)
  -- apenas eventos de tecla interessam
    if evt.class == 'key' and evt.type == 'press'
    then
      -- apenas as setas que movem o logotipo Lua interessam
      if evt.key == 'CURSOR_UP' then
        logoLua.y = logoLua.y - 10
      elseif evt.key == 'CURSOR_DOWN' then
        logoLua.y = logoLua.y + 10
      elseif evt.key == 'CURSOR_LEFT' then
```

```
        logoLua.x = logoLua.x - 10
      elseif evt.key == 'CURSOR_RIGHT' then
        logoLua.x = logoLua.x + 10
      end
    -- testa se os logotipos estão sobrepostos
    if sobrepondo(logoLua, logoNcl) then
        -- sinaliza que a ancora "chegou" esta ocorrendo
        event.post {
       class  = 'ncl',
       type   = 'presentation',
       label  = 'chegou',
       action = 'start',
      }
    end
    redraw()  -- redesenha a tela inteira
   end
end
event.register(tratador)
```

**Listagem 1.22** Terceira parte do código NCLua do Exemplo 3.

Na Listagem 1.22, o *script* inicia testando se alguma das teclas direcionais foi pressionada (campo *key* do evento de tecla), alterando a posição do logotipo de Lua de acordo com a tecla. Caso os logotipos estejam se sobrepondo, é postado o evento para sinalizar que a âncora "*chegou*" iniciou. A função "sobrepondo" foi omitida sem prejuízo de entendimento. Por fim, a tela é redesenhada, qualquer que seja a tecla que tenha sido pressionada.

Em um evento da classe *key*, o valor da tecla é uma string, guardada no campo *key*. O campo *type* pode ser '*press'* ou '*release'*, dependendo se a tecla foi pressionada ou liberada.

### 1.3.4.1. Programando com Animações

Imaginemos agora que o logotipo de Lua fosse movimentado com o passar do tempo, como em uma animação, em vez de ser guiado pelo controle remoto. Ao receber a instrução de "start", o trecho hipotético da Listagem 1.23 atualiza a posição do logotipo a cada 30 milissegundos, até que sua posição chegue a 100.

```
function tratador (evt)
   if evt.action == 'start' then
      while logoLua.x < 100 do
         logoLua.x = logoLua.x + 5
         redraw()
         sleep(30)
      end
   end
end
event.register(tratador, 'ncl', 'presentation')
```

**Listagem 1.23** Exemplo (ruim) de animação em NCLua.

O problema com esse código é que a chamada à função *sleep* bloqueia o tratador, não permitindo que outros eventos sejam processados, inviabilizando essa proposta.

Uma possível solução seria criar uma função de *update* a ser chamada a cada 30 milissegundos por um temporizador (Listagem 1.24).

```
function update ()
   logoLua.x = logoLua.x + 5
   redraw()
   if logoLua.x < 100 then
      event.timer(update, 30)
   end
end
function tratador (evt)
   if evt.action == 'start' then
      update()
   end
end
event.register(tratador, 'ncl', 'presentation')
```

**Listagem 1.24** Exemplo de animação em NCLua que utiliza um temporizador.

Pela listagem anterior, ao ser chamado, o tratador ativa a função *update*, que atualiza a posição do logotipo de Lua, chama a função de redesenho e, caso o logotipo ainda não tenha alcançado a posição 100, se programa para ser chamada novamente após 30 milissegundos de espera.

Essa solução não é tão legível quanto um *loop* localizado, mas mantém o NCLua reativo a possíveis eventos que possam chegar durante os 30 milissegundos.

### 1.3.4.2. Corrotinas de Lua

O mecanismo de corrotinas de Lua simplifica muito a programação de aplicações em que muitos objetos interagem e devem estar permanentemente sincronizados.

Apesar de serem comumente comparadas a *threads*, corrotinas são, na verdade, muito mais próximas ao conceito comum de função (ou rotina). Da mesma forma que funções, chamadas a corrotinas são síncronas, isto é, o código que chama uma corrotina sempre aguarda o seu retorno. No entanto, uma corrotina pode explicitamente suspender sua própria execução, preservando o seu estado corrente (i.e. variáveis locais, contador de instruções), antes do seu término completo. Ao se suspender, uma corrotina retorna imediatamente o controle a quem a chamou. Nesse caso, a corrotina pode, a partir de outro trecho do código, ter sua execução continuada do ponto onde parou, executando até o seu término ou até uma nova suspensão. Note que uma corrotina que não se suspende antes de terminar é exatamente uma rotina (ou função) comum.

O uso de co-rotinas em Lua é feito através das seguintes primitivas:

- *coroutine.create (f)*: Retorna uma nova corrotina a partir da função passada como parâmetro. Cria os meios pelos quais o estado de uma corrotina é preservado entre suspensões.
- *coroutine.resume (co)*: Recomeça a execução da corrotina do ponto onde parou. Também serve para iniciar a corrotina.
- *coroutine.yield ()*: Suspende a execução da corrotina em execução.
- *coroutine.status (co)*: Retorna o estado da corrotina passada, que pode ser *running*, *suspended*, *normal* ou *dead*.

Voltando ao exemplo da animação do logotipo de Lua, agora podemos usar o *loop* problemático da Listagem 1.23, bastando colocá-lo dentro de uma corrotina e trocando a chamada *sleep()* por *coroutine.yield()* (Listagem 1.25).

Na listagem, a função *update* acorda a corrotina a cada 30 milissegundos, até que ela termine, o que acontece quando o logotipo chega à posição 100 quando, então, o *loop* é encerrado.

O uso de corrotinas simula a execução sequencial de código em situações em que, na verdade, uma entidade externa à aplicação controla sua execução (o temporizador, no exemplo apresentado).

É possível ainda trocar valores entre chamadas e suspensões de corrotinas, analogamente à passagem de parâmetros de funções. Para uma descrição mais detalhada do suporte a corrotinas de Lua, o manual da linguagem deve ser consultado [Ierusalimschy 2006].

```
function animaLogoLua ()
   while lua.x < 100 do
      lua.x = lua.x + 5
      redraw()
          coroutine.yield() -- sleep(30)
       end
    end
    coAnimaLogoLua = coroutine.create(animaLogoLua)

function update ()
   coroutine.resume(coAnimaLogoLua)
   if coroutine.status(coAnimaLogoLua) ~= 'dead' then
      event.timer(30, update)
   end
end

function tratador (evt)
   if evt.action == 'start' then
          update()
       end
end
event.register(tratador, 'ncl', 'presentation')
```

**Listagem 1.25** Exemplo de uso de co-rotinas para realizar uma animação.

**Exemplo 4 – Corrida de Cavalos (Parte I)**

Imaginemos a simulação de uma corrida de cavalos. Neste exemplo, a primeira parte da simulação, o *script* para a animação de apenas um cavalo é criado. A segunda parte, apresentada no Exemplo 5, reúsa esse *script* na definição de todos os cavalos.

A Figura 1.10 apresenta uma tira de imagens, com o cavalo em diversas posições. A cada intervalo de tempo, uma imagem diferente do cavalo será mostrada na tela, dando uma sensação de realidade à animação.

**Figura 1.10** Imagem dos cavalos do Exemplo 4.

A tabela representando o cavalo é um pouco diferente da usada para representar os logotipos do exemplo anterior:

```
local img = canvas:new('media/cavalo.png')
local larg, alt = img:attrSize()
local cavalo = { canvas=img,  quadro=0, x=0, y=0,
                 larg=larg/5, alt=alt }
```

**Listagem 1.26** Definição do *canvas* do Exemplo 4.

A largura do cavalo é igual à largura da imagem dividida pelo número de quadros da tira. Além disso, a tabela também guarda o quadro a ser exibido, que varia durante a animação.

A função de redesenho (Listagem 1.27) deve exibir apenas a parte da imagem dos cavalos que representa o quadro corrente. A função *attrCrop* é usada para recortar da tira de imagens do cavalo apenas a parte que forma o quadro de movimento atual.

```
function redraw ()
   canvas:attrColor('black')
   canvas:drawRect('fill', 0,0, canvas:attrSize())
   cavalo.canvas:attrCrop((cavalo.quadro*cavalo.larg), cavalo.y,
                          cavalo.larg, cavalo.alt)
   canvas:compose( cavalo.x, cavalo.y, cavalo.canvas)

   canvas:flush()
end
```

**Listagem 1.27** Função de redesenho do Exemplo 4.

Para animar o cavalo, usaremos uma corrotina, como no exemplo anterior:

```
function animaCavalo ()
   local partida = 10     -- posição da linha de partida
   local chegada = 500    -- posição da linha de chegada
   local mudaQuadro = 20  -- muda de quadro a cada 20px
   cavalo.x = partida
   while cavalo.x < chegada do
      coroutine.yield()
      local deslocamento = 3 + math.random(1,3)
      cavalo.x = cavalo.x + deslocamento
      mudaQuadro = mudaQuadro - deslocamento
      if mudaQuadro < 0 then
         cavalo.quadro = (cavalo.quadro + 1) % 5
         mudaQuadro = 20
      end
      redraw()
   end
end
coAnimaCavalo = coroutine.create(animaCavalo)
```

**Listagem 1.28** Função de animação do cavalo do Exemplo 4.

O deslocamento do cavalo a cada 30 milissegundos varia de 4 a 6 pixels, de acordo com a chamada à função *math.random* (para que a aplicação tenha alguma imprevisibilidade). O código para a função *update* e o tratador de eventos são iguais ao do exemplo anterior, bastando trocar o nome da corrotina de animação.

Novamente, o uso de uma corrotina permite que o código do cavalo seja escrito sequencialmente, facilitando o seu desenvolvimento e entendimento.

### 1.3.5. Reúso de Código Lua

Conforme mencionado, um documento NCL não contém código Lua diretamente, mas referencia um objeto contendo o código Lua. Isso cria dois ambientes isolados de programação e também permite que um mesmo código Lua possa ser reusado em diversos objetos de mídia NCL. Para tornar essa abordagem ainda mais abrangente, elos de atribuição em NCL podem ser usados para informar parâmetros a *scripts* Lua. Âncoras em objetos de mídia NCLua permitem que o código Lua tanto sirva como condição para o disparo de elos como trate ações provenientes de elos, como já vimos.

O reúso de código Lua permite, por exemplo, que se definam componentes gráficos (ou *widgets*), tais como caixas de texto ou painéis de opção, uma única vez. Cada componente gráfico poderia ser reusado e parametrizado em documentos NCL como um objeto de mídia orquestrado, tal como qualquer outro objeto de mídia.

**Exemplo 5 – Corrida de Cavalos (Parte II)**

Este exemplo estende o Exemplo 4 para ilustrar uma corrida entre cavalos. O mesmo *script* NCLua responsável pela animação de um único cavalo é reusado na especificação de quatro objetos de mídia, demonstrando o uso de NCL como um orquestrador entre objetos de código imperativo de igual conteúdo.

A Figura 1.11 apresenta a visão estrutural do exemplo. O objeto de mídia "*cavalo1*" é a porta de início da exibição do documento. Quando "*cavalo1*" é iniciado, os outros objetos de mídia "*cavalo2*", "*cavalo3*" e "*cavalo4*" também são iniciados. Os quatro objetos de mídia referenciam o mesmo arquivo de extensão *.lua*.

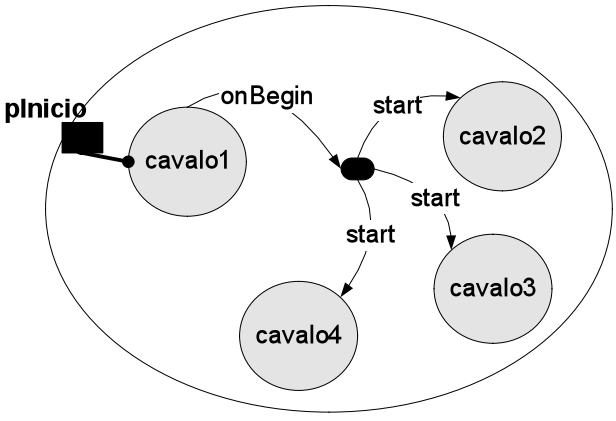


**Figura 1.11** Visão estrutural do Exemplo 5.

A Figura 1.12 ilustra as visões temporal e espacial do exemplo. Para simplificar, na visão temporal, as quatro animações em NCLua aparecem com o mesmo tempo total de exibição. Na prática, conforme é explicado no Exemplo 4, o tempo total de cada animação é imprevisível, já que cada cavalo possui um fator de deslocamento que varia de 4 a 6 pixels a cada 30 milissegundos.

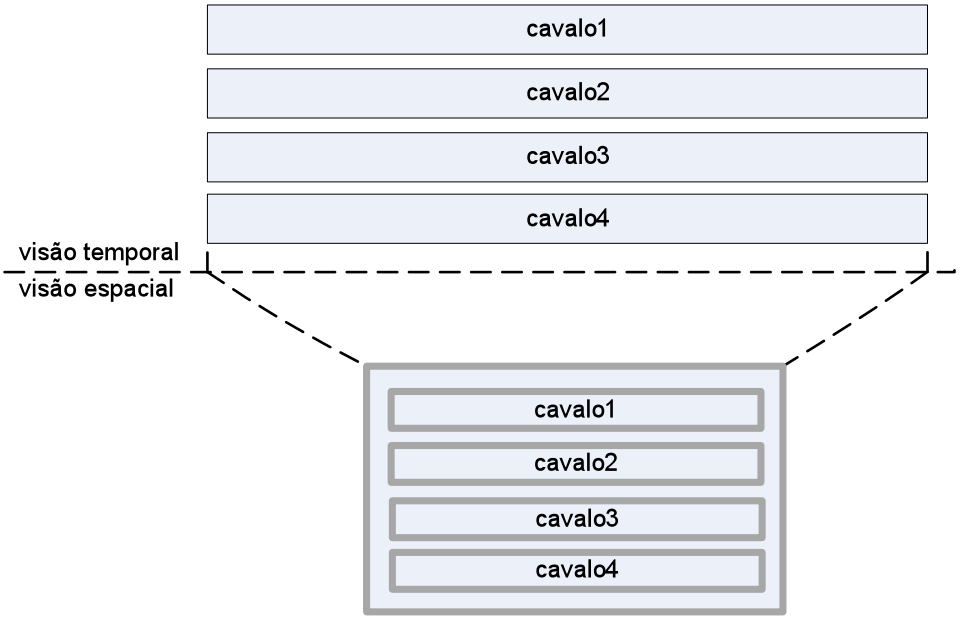


**Figura 1.12** Visões temporal e espacial do Exemplo 5.

**Exemplo 6 – Passagem de Valores**

Esta seção apresenta uma aplicação que ilustra o reúso de código Lua em documentos NCL e a passagem de valores de propriedades entre objetos NCLua.

Quando inicia a aplicação, um campo de entrada e dois campos de saída são exibidos na tela. Enquanto o usuário preenche o campo de entrada, o primeiro campo de saída é automaticamente atualizado com o texto que vai sendo digitado. Apenas quando o usuário encerra a digitação por meio do botão OK do controle remoto, ou através do ENTER no teclado alfanumérico, o texto digitado até o momento é copiado para o segundo campo de saída.

A Figura 1.13 ilustra a visão estrutural deste exemplo. O objeto NCLua "entrada" é disparado no início do documento, o que faz iniciar também os outros dois objetos NCLua "saida1" e "saida2". O foco para digitação inicia no campo de entrada, cuja propriedade "texto" armazena o valor atual digitado pelo usuário. A cada mudança no valor da propriedade "texto", seu valor é copiado para a propriedade "texto" do objeto "saida1", o que atualiza o que é exibido pelo primeiro campo de saída. Por outro lado, apenas quando a âncora "selecao" do objeto "entrada" é iniciada, seu valor é copiado para a propriedade "texto" do objeto "saida2".

Os campos de entrada e saída são implementados em dois *scripts* Lua diferentes. Ambos os códigos tratam a propriedade "*texto*", mantendo-a com o texto visualizado. No caso do campo de entrada, a propriedade "*texto*" é controlada pelo próprio NCLua, e é alterada toda vez que uma nova tecla é pressionada. No caso do campo de saída, a propriedade "*texto*" deve ser controlada pelo documento NCL, através de elos de atribuição.

Neste exemplo, os objetos NCLua podem ser vistos como caixas pretas, não importando o conteúdo dos arquivos *.lua*. Para o autor do documento NCL basta saber a interface que cada um dos NCLua oferece com suas propriedades "*texto*" e a âncora "*selecao*",

especificamente no campo de entrada. Dessa forma, os arquivos *.lua* podem ser reusados em outras aplicações.

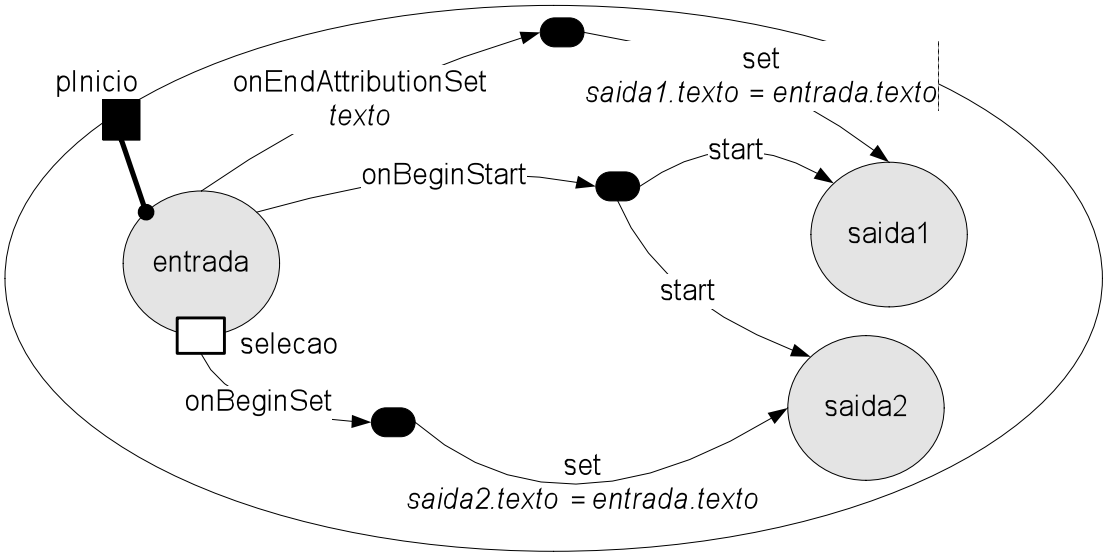


**Figura 1.13** Visão estrutural do Exemplo 6.

O campo de entrada é representado em NCL pelo seguinte código:

```
<media id="entrada" src="input.lua"  descriptor="dsInput">
    <area id="selecao"/>
    <property name="texto"/>
</media>
```

**Listagem 1.29** Campo de entrada.

O objeto possui a âncora "*selecao*" que é iniciada sempre que o usuário pressiona OK no controle remoto ou ENTER no teclado alfanumérico.

Os campos de saída são representados com o seguinte código:

```
<media id="saida1" src="output.lua" descriptor="dsOutput1">
    <property name="texto"/>
</media>
<media id="saida2" src="output.lua" descriptor="dsOutput2">
    <property name="texto"/>
</media>
```

**Listagem 1.30** Campos de saída.

A parte mais importante do documento é a que contém os elos responsáveis por copiar o conteúdo do campo de entrada para os campos de saída.

O primeiro campo de saída é atualizado sempre que a propriedade "*texto*" do campo de entrada é alterada:

```
<link xconnector="onEndAttributionSet">
    <bind role="onEndAttribution" component="entrada"
          interface="texto"/>
    <bind role="set" component="saida1" interface="texto">
```

```
        <bindParam name="var" value="$get"/>
    </bind>
    <bind role="get" component="entrada" interface="texto"/>
</link>
```

**Listagem 1.31** Elo que atualiza o primeiro campo de saída com a propriedade "texto" do campo de entrada.

A associação com o papel "*get*", definido no próprio elo, permite consultar o valor de uma propriedade de qualquer objeto, guardando-o na variável "$get". No exemplo acima, a propriedade consultada é a "*texto*" do objeto "*entrada*". A variável "$get", por sua vez, é utilizada na associação com o papel de atribuição "*set*", fazendo com que o valor da entrada seja copiado para a propriedade "*texto*" do objeto "*saida1*".

O segundo campo de saída é atualizado somente quando a âncora "*selecao*" do objeto de "*entrada*" é iniciada:

```
<link xconnector="onBeginSet">
    <bind role="onBegin" component="entrada"
          interface="selecao"/>
    <bind role="set" component="saida2" interface="texto">
        <bindParam name="var" value="$get"/>
    </bind>
    <bind role="get" component="entrada" interface="texto"/>
</link>
```

**Listagem 1.32** Elo que atualiza o segundo campo de saída quando a âncora "selecao" do campo de entrada é iniciada.

O mesmo mecanismo de cópia de valores de uma propriedade para outra também é usado para a cópia do valor digitado no campo de entrada para o segundo campo de saída.

## 1.4. Documentação de Referência NCLua

Esta seção tem o objetivo de servir como um guia de referência rápida à API NCLua. Esta referência é baseada na norma ABNT NBR 15606-2:2007 relativa ao *middleware* Ginga. Ela extrai da norma o que é referente à NCLua, ou seja, basicamente a especificação das novas bibliotecas adicionadas a Lua para TV Digital.

As bibliotecas NCLua são divididas em módulos essenciais, onde cada qual exporta um conjunto de funções de seu domínio, são eles: *event*, *canvas*, *persistent* e *settings*. Esta seção trata apenas dos dois primeiros módulos.

### 1.4.1. Módulo Event

A comunicação entre aplicações NCL e objetos NCLua é feita através do paradigma de eventos. Os objetos NCLua, através da API de tratamento de eventos disponibilizada pelo módulo Event, podem assim receber ou enviar eventos para o formatador NCL. Um objeto NCLua também pode fazer uso deste mecanismo internamente, através da classe de eventos "user".

**event.post([dst:string]; evt: event) → sent: boolean; err_msg: string;**

Posta o evento *evt* passado como parâmetro.

O parâmetro opcional *dst* informa o destinatário do evento, podendo assumir os valores:

"*in"* – informa que o evento deve ser enviado para o próprio NCLua.

"*out"* – informa que o evento deve ser enviado ao formatador NCL.

O valor padrão do *dst* quando omitido é "*out"*.

Todos os exemplos apresentados neste minicurso, de alguma forma, utilizam esta função. A Listagem 1.13 (Exemplo 1) mostra um exemplo de uso simples de como postar um evento, informando ao formatador NCL que um *script* NCLua acabou sua execução.

**event.register([pos:number]; f: function; [class: string]; […: any] )**

Registra a função passada como um tratador de eventos, isto é, sempre que ocorrer um evento, *f* será chamada.

O parâmetro *pos* é opcional e, informa em qual posição na lista de tratadores de eventos, a função f deve ser inserida. Caso *pos* não seja informado, a função é registrada na última posição da fila.

O formatador NCL garante que as funções recebem os eventos na ordem em que elas foram registradas.

A assinatura de f deve ser:

```
function f (evt)
    -- return boolean
end
```

onde *evt* é o evento que, ao ocorrer, ativa a função *f*.

A função *f* pode retornar *true*, para sinalizar que o evento foi tratado e, portanto, não deve ser enviado a outros tratadores ou *false*, informando que o evento não foi tratado e que deve ser passado para os tratadores de eventos que estão nas posições seguintes na fila de tratadores de eventos.

É recomendado que a função *f* retorne rapidamente, já que, enquanto ela estiver executando, nenhum outro evento é processado.

O parâmetro *class* é opcional e permite informar ao formatador NCL um filtro de classe para os eventos passados à função *f*.

Na Listagem 1.16 (Exemplo 2), como apenas os eventos de apresentação sobre a propriedade "inc" interessam, a função *tratador* poderia ser chamada com o seguinte filtro: `event.register(tratador,'ncl','presentation','inc','start')`.

**event.timer (time: number; f: function) → unreg: function**

Cria um timer que expira após *time* milissegundos e então chama a função *f*.

Retorna *unreg*: a função que, quando chamada pelo NCLua irá cancelar o timer.

A assinatura de *f* é simples, sem parâmetros:

```
function f () end
```

O valor de 0 milissegundos é válido. Neste caso, *f* é chamada assim que possível (nunca imediatamente, dentro de um tratador de eventos). Assim como tratadores de eventos, *f* deve retornar rapidamente, pois enquanto ela é executada nenhum evento é tratado.

A Listagem 1.14 (Exemplo 1) exemplifica o uso da função *event.time*.

---

**event.unregister (f: function)**

Faz com que a função passada como parâmetro não seja mais um tratador de eventos, isto é, novos eventos não serão mais passados a *f*.

---

**event.uptime () → ms: number**

Retorna *ms*: o número de milissegundos decorridos desde o início da aplicação.

### 1.4.2. Módulo Canvas

Um NCLua tem a possibilidade de fazer operações gráficas durante a apresentação de uma aplicação, tais como desenho de linhas, círculos, imagens, etc. Quando um NCLua é iniciado, automaticamente é instanciado um objeto gráfico que é atribuído à variável global *canvas*. Este objeto aponta para a região associada ao nó de mídia NCLua no documento NCL e é através dele que todas as operações gráficas são feitas.

---

**canvas:attrClip ( ) → x, y, width, height: number**

Acessa o atributo que limita a área do *canvas* para desenho. As primitivas de desenho e o método *compose* só operam dentro da região limitada. O valor inicial é o *canvas* inteiro. Os valores de retorno são:

* x: [number] Coordenada x da área limitada.
* y: [number] Coordenada y da área limitada.
* width: [number] Largura da área limitada.
* height: [number] Altura da área limitada.

---

**canvas:attrClip ( x, y, width, height: number )**

Altera os valores do atributo que limita a área do *canvas* para desenho. Os valores dos atributos são equivalentes aos valores de retorno de `canvas:attrClip()`.

---

**canvas:attrColor ( ) → R, G, B, A: number**

Acessa o atributo de cor do *canvas*. As primitivas gráficas utilizam a cor deste atributo do *canvas*. As cores são descritas em RGBA, onde A varia de 0 (totalmente transparente) a 255 (totalmente opaco). O valor inicial do atributo é 0,0,0,255 (preto). Os valores de retorno são:

* R: [number] Componente vermelha da cor.

* G: [number] Componente verde da cor.

* B: [number] Componente azul da cor.

* A: [number] Componente *alpha* da cor.

---

**canvas:attrColor (R, G, B, [A]: number)**

Altera o atributo de cor do *canvas*. Os valores dos argumentos são equivalentes aos valores de retorno de `canvas:attrColor()`.

O valor padrão de *alpha*, caso não seja informado, é 255 (opaco).

---

**canvas:attrColor (color_name: string)**

Altera o atributo de cor do *canvas* com cores pré-definidas em opaco (A=255), onde *color_name* é o nome da cor.

Os nomes das 16 cores NCL pré-definidas são:

*'white', 'aqua', 'lime', 'yellow', 'red', 'fuchsia', 'purple', 'maroon',*

*'blue', 'navy', 'teal', 'green', 'olive', 'silver', 'gray', 'black'*

---

**canvas:attrCrop ( ) → x, y, width, height: number**

Acessa o atributo de recorte do *canvas*. Quando o *canvas* é composto sobre outro, apenas a região de recorte é copiada para o *canvas* de destino. O valor inicial é o *canvas* inteiro. Os valores de retorno são:

* x: [number] Coordenada x da área limitada.

* y: [number] Coordenada y da área limitada.

* width: [number] Largura da área limitada.

* height: [number] Altura da área limitada.

---

**canvas:attrCrop ( x, y, width, height: number )**

Altera os atributos de recorte do *canvas*. Os valores dos argumentos são equivalentes aos valores de retorno de `canvas:attrCrop()`.

---

**canvas:attrFlip ( ) → horiz, vert: boolean**

Retorna o espelhamento do *canvas*. Valores de retorno:

* horiz: [boolean] Informa se o espelhamento for horizontal.

* vert: [boolean] Informa se o espelhamento for vertical.

---

**canvas:attrFlip ( horiz, vert: boolean )**

Configura o espelhamento do *canvas* usado em funções. Os valores dos argumentos são equivalentes aos valores de retorno de `canvas:attrFlip()`.

---

**canvas:attrFont ( ) → face: string; size: number; style: string**

Acessa o atributo de fonte do *canvas*. Retorna a fonte do *canvas*:

* face: [string] Nome da fonte.

* size: [number] Tamanho da fonte.

* style: [string] Estilo da fonte.

O tamanho é em pixels e representa a altura máxima de uma linha escrita com a fonte escolhida. Os estilos possíveis são: 'bold', 'italic' ou 'bold-italic'.

---

**canvas:attrFont ( face: string; size: number; [style: string] )**

Configura a fonte do *canvas*. Os valores dos argumentos são equivalentes aos valores de retorno de `canvas:attrFont()`.

O valor `nil` assume que nenhum dos estilos será usado. Qualquer valor passado não suportado deve obrigatoriamente gerar um erro. O valor inicial da fonte é indeterminado.

---

**canvas:attrOpacity ( ) → opacity: number**

Retorna o valo da opacidade do *canvas*, que varia de 0 (transparente) a 255 (opaco).

---

**canvas:attrOpacity ( opacity: number )**

Altera a opacidade do *canvas*. O *canvas* principal não pode ser alterado, pois é controlado pelo formatador NCL. O argumento varia de 0 (transparente) a 255 (opaco).

---

**canvas:attrRotation ( ) → degrees: number**

Retorna o atributo de rotação do *canvas*.

---

**canvas:attrRotation ( degrees: number )**

Configura o atributo de rotação do *canvas*. O *canvas* principal não pode ser alterado, pois é controlado pelo formatador NCL. O grau informado deve ser múltiplo de 90 graus.

---

**canvas:attrScale ( ) → w, h: number**

Retorna a largura *w* e altura *h* de escalonamento do *canvas*.

---

**canvas:attrScale ( w, h: number )**

Escalona o *canvas* com nova largura e altura informadas como parâmetro. Um dos valores pode ser *true*, indicando que a proporção do *canvas* deve ser mantida. O *canvas* principal não pode ser alterado, pois é controlado pelo formatador NCL. O atributo de escalonamento é independente do atributo de tamanho.

---

**canvas:attrSize ( ) → width, height: number**

Retorna o tamanho do *canvas*, onde *width* é a largura e height é a altura do *canvas*. Não é possível alterar as dimensões de um *canvas* instanciado. Portanto, para este método, apenas a leitura está disponível.

---

**canvas:clear ([x, y, w, h: number])**

Limpa o *canvas* com o cor configurada em `canvas:attrColor`. Os parâmetros são opcionais e especificam qual deve ser a região do *canvas* que será limpada. Caso não passados, assume-se que ela é todo o *canvas*.

---

**canvas:drawEllipse (mode:string; xc, yc, width, height, ang_start, ang_end: number)**

Desenha elipses e outras primitivas similares (círculos, arcos e setores). Recebe:

* mode: [string] Mode de desenho. Pode ser *'arc'* para desenhar apenas a circunferência ou *'fill'* para preenchimento interno.

* xc: [number] Posição *x* do centro da elipse.

* yc: [number] Posição *y* do centro da elipse.

* width: [number] Largura da elipse.

* height: [number] Altura da elipse.

* ang_start: [number] Ângulo de início.

* ang_end: [number] Ângulo de fim.

---

**canvas:drawLine (x1, y1, x2, y2: number)**

Desenha uma linha com extremidades em (x1,y1) e (x2,y2). Utiliza a cor especificada em *attrColor*. Recebe:

* x1: [number] Extremidade 1 da linha.

* y1: [number] Extremidade 1 da linha.

* x2: [number] Extremidade 2 da linha.

* y2: [number] Extremidade 2 da linha.

---

**canvas:drawPolygon (mode:string) → drawer: function**

Desenha e preenche polígonos. Recebe o modo de desenho (*mode*) que pode ser '*open'* para desenhar polígonos sem ligar o último ponto ao primeiro ou '*fill'* para desenhar o polígono e colorir a região interior.

A função *drawer* retornada recebe as coordenadas do próximo ponto do polígono e retorna a si mesmo como resultado. Ao receber *nil* a função efetua a operação de desenhar o polígono. Veja o exemplo:

```
canvas:drawPolygon('fill')(0,0)(100,0)(10,100)(0,100)()
```

---

**canvas:drawRect (mode:string, x, y, width, height: number)**

Desenha um retângulo no *canvas*. Utiliza a cor especificada em *attrColor*. Recebe:

* mode: [string] Modo de desenho: 'frame' ou 'fill'.
* x: [number] Coordenada do retângulo.
* y: [number] Coordenada do retângulo.
* width: [number] Largura do retângulo.
* height: [number] Altura do retângulo.

O parâmetro *mode* pode receber 'frame' para desenhar apenas a moldura do retângulo ou 'fill' para preenchê-lo.

---

**canvas:drawRoundRect (mode, x, y, width, height)**

Desenha um retângulo com cantos arredondados no *canvas*. Utiliza a cor especificada em *attrColor*. Recebe:

* mode: [string] Modo de desenho: 'frame' ou 'fill'.
* x: [number] Coordenada do retângulo.
* y: [number] Coordenada do retângulo.
* width: [number] Largura do retângulo.
* height: [number] Altura do retângulo.
* arcWidth: [number] Largura do canto arredondado.
* arcHeight: [number] Altura do canto arredondado.

O parâmetro *mode* pode receber 'frame' para desenhar apenas a moldura do retângulo ou 'fill' para preenchê-lo.

---

**canvas:drawText (x, y, text)**

Desenha o texto passado na posição (x,y) do *canvas*. Utiliza a cor especificada em *attrColor* e fonte em *attrFont*. Recebe:

* x: [number] Coordenada x do texto.
* y: [number] Coordenada y do texto.
* text: [string] Texto a ser desenhado.

---

**canvas:flush ()**

Atualiza o *canvas* após operações de desenho e de composição. É suficiente chamá-la apenas uma vez após uma sequência de operações.

---

**canvas:measureText (text) → dx, dy: number**

Retorna as dimensões do texto passado. Utiliza a fonte especificada em *attrFont*. Recebe:

* text: [string] Texto a ser medido.

Retorna:

* dx: [number] Largura do texto.

* dy: [number] Altura do texto.

---

**canvas:new (width, height: number) → canvas: object**

A partir do objeto *canvas* é possível criar novos objetos gráficos e combiná-los através de operações de composição. Este construtor instancia um *canvas* com o tamanho especificado, onde *width* é a largura e *height* a altura do *canvas*.

Inicialmente os pixels são todos transparentes.

---

**canvas:new (image_path: string) → canvas: object**

Este construtor instancia um *canvas* cujo conteúdo é a imagem passada como parâmetro, onde *image_path* é o caminho da imagem.

O novo *canvas* mantém os aspectos de transparência da imagem original.

---

**canvas:pixel (x, y, R, G, B, A: int)**

Altera a cor do pixel que ocupa posição x, y no *canvas*.

---

**canvas:pixel (x, y: int) → R, G, B, A: int**

Retorna a cor do pixel que ocupa posição x, y no *canvas*.

## Referências


ABNT NBR 15606-2:2007. Televisão digital terrestre - Codificação de dados e especificações de transmissão para radiodifusão digital – Parte 2: Ginga-NCL para receptores fixos e móveis – Linguagem de aplicação XML para codificação de aplicações. Setembro 2007.

Ierusalimschy, R. (2006) "Programming in Lua", Second Edition. http://Lua.Org.

Soares, L. F. G., Barbosa, S.D.J. (2009) *Programando em NCL*. Série Campus-SBC.

Soares, L. F. G., Rodrigues, R. F., Moreno, M. F. Ginga-NCL: the Declarative Environment of the Brazilian Digital TV System. Journal of the Brazilian Computer Society. n.4, v. 12, Março, 2007.

Soares Neto, C. S. et al. Construindo Programas Audiovisuais Interativos Utilizando a NCL 3.0 e a Ferramenta Composer. 2007.

ITU-T Recommendation H.761, 2009. Nested Context Language (NCL) and Ginga-NCL for IPTV Services. Geneva, Abril, 2009.