4
Framework Proposto

Neste capítulo, é apresentado o framework proposto neste trabalho. O framework permite que qualquer aplicação científica que utilize OpenGL seja manipulada simultaneamente por uma ou mais pessoas usando dispositivos móveis. Com a liberdade de movimento de um dispositivo móvel, um grupo de pessoas pode interagir colaborativamente a frente de um telão ou projeção sem a necessidade de estar operando diretamente o computador que roda a visualização.

Dispondo de uma integração rápida, a aplicação científica recebe toda a interação dos usuários através de eventos de alto nível, que representam ações que o usuário executou através de gestos ou movimentos no dispositivo móvel.

Figura 4.1: Framework cliente-servidor.

Baseado no modelo cliente-servidor, o framework sugere um esquema de renderização remota onde de um lado, no papel de servidor, está um computador potente rodando a aplicação de visualização científica, onde toda a parte de renderização e manipulação do modelo acontece. Este computador dispõe de bastante memória e placas de vídeo (*GPU*) de última geração para processar o grande volume de dados que normalmente esse tipo de aplicação requer. Do outro lado, no papel de cliente, está o celular/tablet rodando um aplicativo que irá se conectar via rede Wi-Fi ao servidor. Este aplicativo é responsável por montar na tela a interface proposta pelo servidor, monitorar os eventos de toque e exibir o retorno visual renderizado pelo servidor.

Um dos desafios deste framework é manter a qualidade visual da renderização sem comprometer o desempenho e interatividade da visualização. A principal limitação para atingir este objetivo é a conexão entre o cliente e o servidor. Os dados de uma imagem sem compressão são grandes

demais para serem transmitidos até mesmo por conexões de rede cabeadas. Para exemplificar, uma imagem com uma resolução de 1280x720 *pixels* a 25 quadros por segundo (FPS) precisaria de uma conexão de 69 MBytes/s: $largura\ de\ banda = tamanho\ de\ um\ quadro \times quadros\ por\ segundo = (1280 \times 720 \times 3)\frac{bytes}{quadro} \times 25\frac{quadros}{s} \simeq 69\frac{MBytes}{s}$. Entre as conexões disponíveis nos dispositivos móveis, Wi-Fi é a mais rápida podendo atingir uma velocidade nominal de 300 Mbits/s $\simeq$ 37,5 MBytes/s. Isso nos leva a concluir que precisamos comprimir os dados da imagem com uma taxa de compressão de pelo menos 0.5 para alcançar 25 FPS.

O fluxo de execução principal do framework é ilustrado na Figura 4.2. Ao conectar no servidor, o cliente solicita a interface a ser exibida e ao recebê-la, a monta na tela. Em seguida, o cliente começa a solicitar quadros continuamente do servidor, e quando um evento ou gesto acontece, o mesmo é enviado ao servidor.

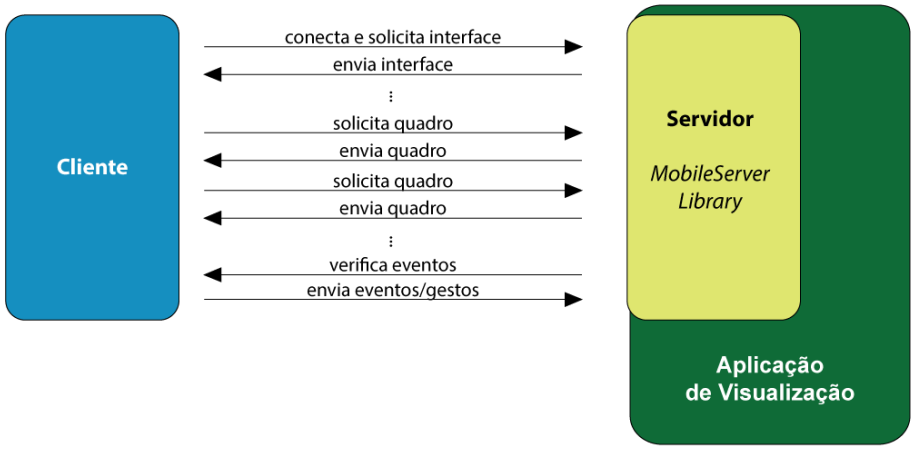

Figura 4.2: Fluxo de execução principal do Framework.

4.1 Servidor

Para reduzir o esforço de acoplamento dos dispositivos móveis nas aplicações de visualização científica já existentes, propõe-se a utilização de uma biblioteca para integração do framework no lado da aplicação. Esta biblioteca incorpora todas as funções do servidor e as executa de forma assíncrona. A biblioteca é responsável por:

- Processar e compactar os quadros a serem enviados aos clientes;
- Controlar toda a parte da comunicação TCP/IP;
- Interpretar os eventos enviados pelo cliente e disponibilizá-los através de chamadas de *callback*.

A biblioteca dispõe de uma API para inicialização e envio dos quadros renderizados. A integração é feita basicamente em dois estágios: no início do programa e após cada renderização de quadro. No primeiro estágio, o programa define uma descrição de como será montada a interface do usuário no dispositivo móvel e também define a porta TCP que o servidor ficará esperando as conexões. O servidor é então iniciado e já está pronto para uso. No segundo estágio, a biblioteca se encarrega de ler o conteúdo do buffer de cores do OpenGL e enviar o quadro para compactação. Nesta mesma etapa a aplicação verifica se há eventos disponíveis que tenham sido enviados do cliente.

Para transferir as imagens, um soquete TCP é estabelecido com o cliente. A vantagem de um soquete TCP é que a ordem correta e a integridade das imagens enviadas são garantidas. Inicialmente optamos por usar uma técnica de compressão sem perdas para obter a qualidade máxima da renderização, mas em alguns casos, que serão detalhados na Seção 4.1.2, podemos reduzir a qualidade da imagem temporariamente para garantir uma boa taxa de frames por segundo.

A compactação dos frames é feita usando LZO(10) (Lempel-Ziv-Oberhumer), um algoritmo de compressão de dados sem perda que implementa o esquema Lempel-Ziv(16)(17) com foco na velocidade de compressão e descompressão, permitindo assim a descompressão dos frames no cliente em tempo real.

Os eventos recebidos do cliente são representações de alto nível de gestos ou movimentos do dispositivo móvel que são transformados no servidor em chamadas de *callback*. Há casos em que as callbacks executam operações no contexto de renderização corrente, mas a linha de execução do servidor não tem acesso a este contexto. Para evitar conflitos entre diferentes contextos de renderização do OpenGL, as *callbacks* são somente chamadas quando a aplicação executa uma API para processar todos os eventos pendentes de notificação.

4.1.1 Multithread

Como no lado do servidor não temos a priori uma limitação de recursos definida, podemos explorar com mais liberdade o uso de threads para acelerar o processo de produção de frames.

Inicialmente, pelo menos uma linha de execução (*thread*) é necessária para isolar o trabalho do servidor do fluxo normal da aplicação científica. Esta linha de execução fica responsável por controlar a conexão TCP utilizando soquetes não bloqueantes.

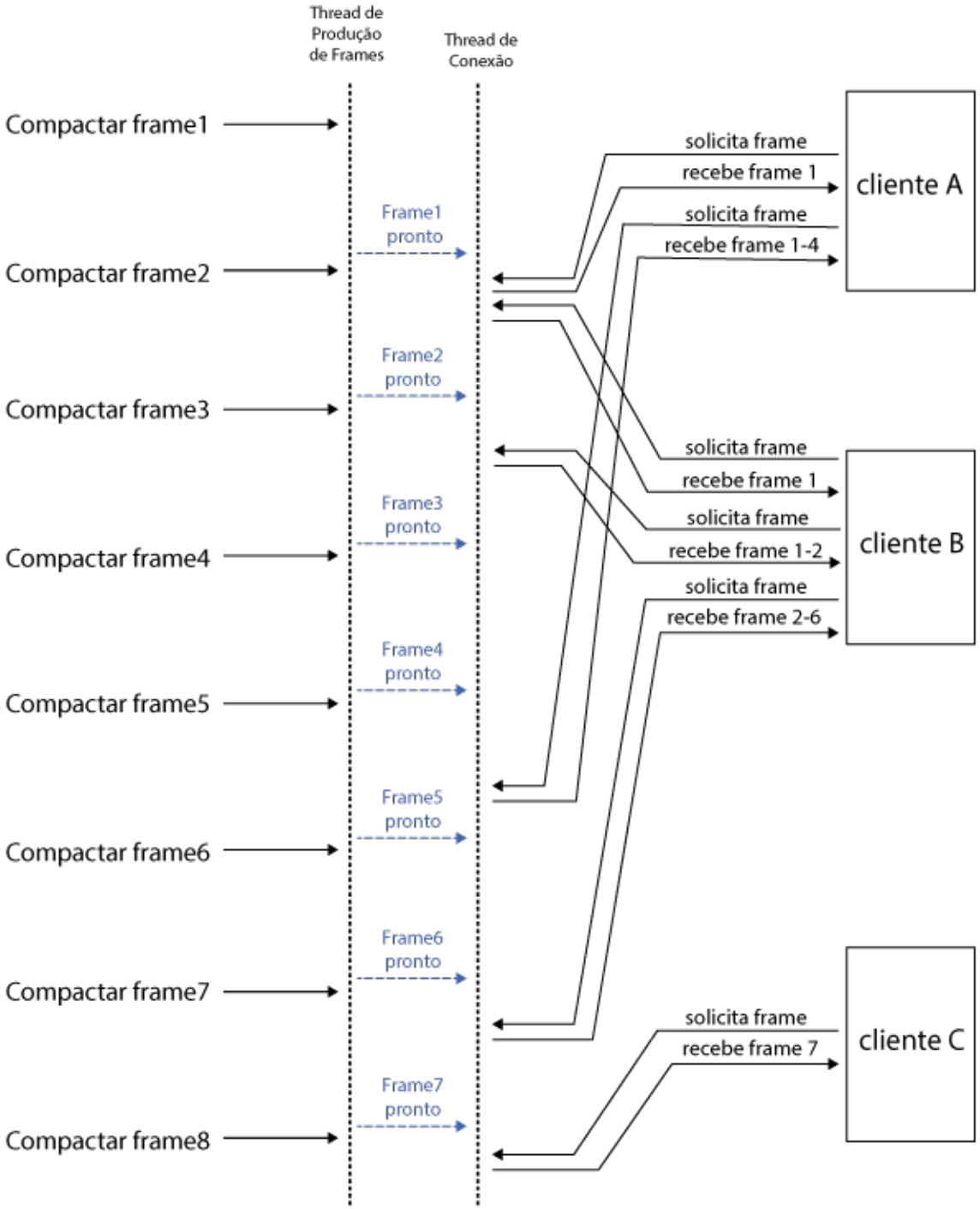

Figura 4.3: Comunicação entre a *thread* de produção e a *thread* de conexão. A *thread* de conexão responde aos pedidos de quadro imediatamente enviando o último quadro compactado disponível.

A fim de minimizar o tempo de resposta do servidor às requisições de frames, podemos separar a tarefa de compactação dos frames em outra linha de execução, assim toda requisição de frame do cliente é respondida imediatamente com o último frame compactado disponível. Dessa forma num cenário de múltiplos clientes, um cliente não tem de esperar o tempo de compactação e envio do quadro de outro cliente para ter sua requisição processada. A Figura 4.3 exemplifica como as threads se comportariam nessa abordagem. No exemplo, o cliente B solicita um quadro e recebe imediatamente o último quadro disponível (Quadro1). Em seguida, ele solicita o próximo quadro e recebe a diferença do Quadro1 para o Quadro2. Em um terceiro instante o cliente B solicita um novo quadro e recebe a diferença do último quadro que recebeu (Quadro2) para o último quadro disponível (Quadro6).

4.1.2 Quadros-diferença

Para aumentar a taxa de compactação, e em conseqüência aumentar o número de quadros transferidos por segundo pela rede, foi utilizada uma estratégia que compara o quadro atual com o último quadro enviado ao cliente. Todos os pixels que mantiveram a mesma cor do último quadro enviado são zerados, ou seja, sua cor é definida como R:0, G:0, B:0, A:0 e somente os diferentes são mantidos, maximizando a compactação quando houver pouca alteração entre os quadros.

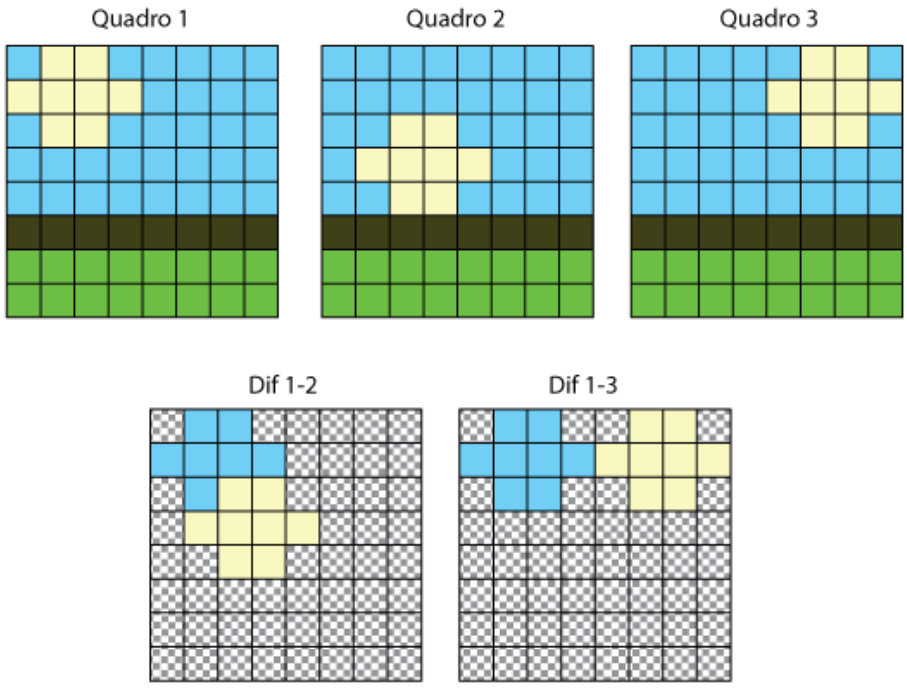

Figura 4.4: Cálculo da diferença entre os quadros.

A Figura 4.4 ilustra o funcionamento da técnica. A diferença do Quadro1 para o Quadro2 (Dif 1-2) contém somente os pixels que mudaram de cor de um quadro para o outro. Da mesma forma, o quadro-diferença Dif 1-3 contém somente as mudanças do Quadro1 para o Quadro3.

Com a técnica de quadros-diferença, cada cliente tem uma tarefa extra de produção de quadro. Dessa forma é possível também usar diferentes linhas de execução para processar o mesmo quadro para todos os clientes em paralelo.

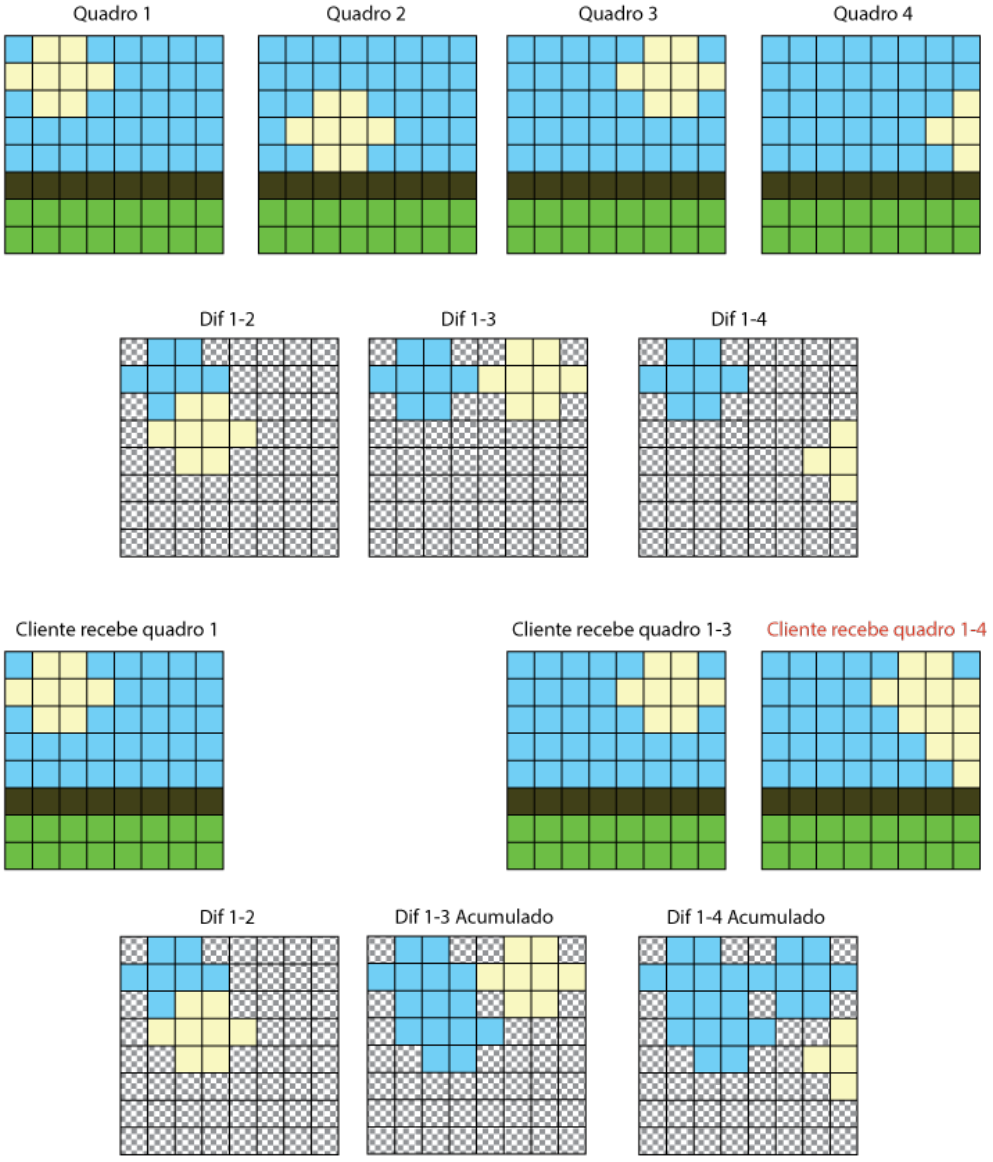

Figura 4.5: Cálculo da diferença entre quadros em uma thread separada causa artefatos na imagem final do cliente. O uso de uma máscara de bits para acumular as diferenças resolve o problema.

Como o envio dos quadros é realizado na *thread* de conexão, há momentos em que durante o processo de compactação, um novo quadro seja solicitado pelo cliente e a diferença sendo calculada no quadro atual se torna incoerente.

A Figura 4.5 ilustra o problema. Quando o cliente recebe o quadro Dif 1-3, o quadro Dif 1-4 já está em processamento no servidor. No Quadro5 o servidor iria processar a diferença Dif3-5, mas se o cliente solicitar um novo quadro antes do Quadro5 ficar pronto, o quadro Dif 1-4 seria enviado criando um artefato na imagem final. Para solucionar esse problema, uma máscara é criada para armazenar as diferenças encontradas, acumulando-as a cada novo quadro produzido. Quando um quadro é enviado ao cliente, uma nova máscara vazia é criada. Isso garante que o quadro compactado contenha sempre todas as mudanças desde o último quadro, mantendo assim uma consistência nas diferenças.

Em alguns casos, a quantidade de mudanças é tão grande que praticamente a imagem completa é enviada a cada frame. Para lidar com este problema, reduzimos o tamanho da imagem produzida de um fator f (e.g. $f = 50\%$) a fim de reduzir o tamanho em bytes de cada frame e por conseqüência manter a taxa de quadros por segundo dentro da faixa esperada. Um limite máximo (e.g. 10%) para a quantidade de mudanças é definido para controlar a transição de tamanhos. Ao passar desse limite, a imagem é reduzida, e a mesma só retorna ao tamanho natural quando a quantidade de mudanças voltar a ser menor que o limite definido.

4.2 Cliente

O cliente é representado por um aplicativo rodando nos dispositivos móveis. Este aplicativo é capaz de se conectar a qualquer aplicação de visualização, sem conhecer as funcionalidades e comportamento das mesmas. Através de uma descrição de alto nível da interface, o servidor define o *layout* e o comportamento dos elementos visuais que serão exibidos no dispositivo móvel. O cliente foi implementado usando o Moai SDK e é responsável por:

- Montar na tela a interface especificada pelo servidor.
- Monitorar os toques, identificar gestos e enviar eventos.
- Baixar e renderizar os frames produzidos no servidor.

Ao conectar no servidor, o cliente solicita a interface que será montada na tela. Esta interface, descrita em Lua, define os elementos visuais que estarão na tela e como estarão dispostos, assim como uma lista de recursos (imagens) que serão utilizados. Todas as imagens necessárias são então baixadas e em seguida a interface é montada na tela. Cada elemento visual na tela pode definir um conjunto de gestos que serão monitorados para notificação. Além de gestos, eventos de giroscópio e acelerômetro também podem ser monitorados

e repassados ao servidor. A Seção 4.3 dá mais detalhes sobre o formato da interface.

Algumas aplicações podem querer usar o dispositivo móvel apenas como um dispositivo de entrada (e.g. dispositivo de controle em ambientes imersivos) e nenhum *feedback* visual é exibido na tela. Mas quando o servidor especifica na interface que o *feedback* visual da renderização será mostrado na tela, o cliente cria uma co-rotina para solicitar continuamente os quadros renderizados do servidor. Os quadros recebidos são descompactados e exibidos na tela. Quando um quadro-diferença é recebido, em vez substituir na tela a imagem, o cliente compõe a imagem nova com a imagem anterior, substituindo assim somente as diferenças enviadas.

4.3 Especificação da Interface Móvel em Lua

Uma especificação de interface com um alto grau de abstração foi criada para permitir que a aplicação científica descreva facilmente o *layout* da tela no dispositivo móvel. Todos os elementos de interface citados na Seção 3.4 podem ser especificados através de uma tabela descritiva em Lua que contem a configuração visual do objeto. O formato da tabela foi inspirado no artigo de Celes et al. (1) que apresentou uma ferramenta para criação de interfaces gráficas interativas. Para exemplificar o formato, o trecho de código abaixo descreve uma imagem:

```
local imagem1 = image { name = "imagem1",         -- nome do objeto
                        imageSrc = "imagem.png", -- arquivo de imagem
                        width = 100,             -- largura da imagem
                        height = 100,            -- altura da imagem
                        notify = { tap = true }  -- eventos a serem notificados
                      }
```

Cada elemento na tela pode especificar, através do parâmetro *notify*, quais gestos ou eventos deverão ser notificados para aquele objeto. O parâmetro *notify* define uma tabela que contém uma entrada para cada evento (e.g *tap = true*). Os elementos do tipo imagem(*image*) ou fundo(*background*) podem receber os eventos de gestos (e.g *tap*, *doubleTap*, *pinchZoom*), enquanto os botões(*button*) recebem somente *press* e *release*. Os demais elementos não possuem eventos pré-definidos mas podem ser configurados para realizar notificações também se necessário.

Quando um gesto é executado ou um botão é pressionado, o evento gerado é enviado ao servidor no formato de uma chamada de função Lua.
Ex.: `rotate( { object="objectName", angle=32.5 } )`

Para informar ao cliente que o conteúdo da imagem no fundo da tela será renderizado pelo servidor e enviado pela rede, o parâmetro `background.source` deve ser definido como `"network"`.

No final da especificação é preciso retornar uma tabela contendo a lista de objetos definidos e quais recursos (imagens) serão usadas por eles.

O trecho de código abaixo ilustra a construção da interface mostrada na Figura 4.6:

```
local fundo = background{ name = "fundo",
                          source = "background.png",
                          style = "stretch",
                          notify = {
                              tap = true,
                              doubleTap = true,
                          }
                        }
local titulo = text{ name = "titulo",
                     text = "Interface de exemplo",
                     width = 1000,
                     height = 200,
                     x = 250,
                     y = 100,
                     font = "times.ttf",
                     color = {0,0,0}
                   }
local botaoPlay = button{ name = "botaoPlay",
                          imageSrc = "play.png",
                          width = 200,
                          height = 200,
                          x = 100,
                          y = 368,
                          notify = { release = true }
                        }
local botaoPause = button{ name = "botaoPause",
                           imageSrc = "pause.png",
                           width = 200,
                           height = 200,
                           x = 542,
                           y = 368,
                           notify = { release = true }
                         }
local botaoStop = button{ name = "botaoStop",
                          imageSrc = "stop.png",
                          width = 200,
                          height = 200,
                          x = 984,
                          y = 368,
                          notify = { release = true }
                        }
local interface = { objects = { fundo,
                                titulo,
                                botaoPlay,
                                botaoPause,
                                botaoStop
                    },
                        resources = { "play.png",
                                      "pause.png",
```

```
                                    "stop.png",
                                    "background.png",
                                    "times.ttf"
                    }
                  }
    return interface
```

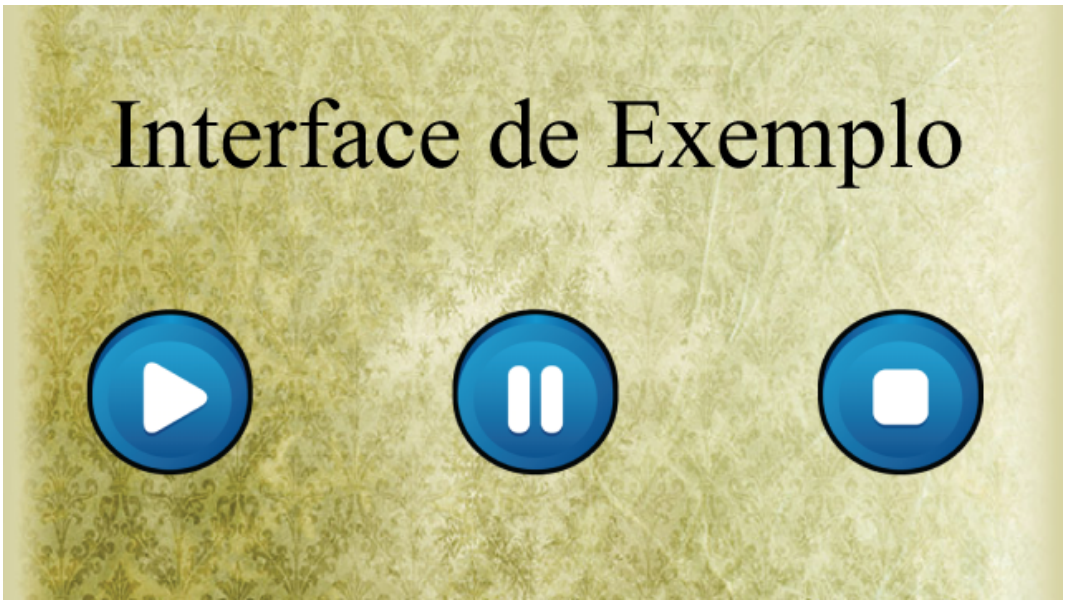

Figura 4.6: Exemplo de uma interface simples.

Este arquivo de interface quando recebido pelo cliente será executado dinamicamente e montado na tela. Como a interface é especificada em Lua e é carregado dinamicamente, a aplicação visualizadora também pode programar comportamentos de tela mais complexos usando diretamente as extensões de API criadas e/ou as APIs do Moai. Por exemplo, um botão pode habilitar ou desabilitar outros com o seguinte código:

```
botaoPlay.onPress = function()
                        botaoPause:enable()
                        botaoPlay:disable()
                    end
```