5
O *framework*

Este capítulo apresenta o *framework* proposto. O *framework*, composto por uma camada Lua integrada a um motor de jogos 2D, permite que variações de jogos educativos do tipo aventura sejam facilmente implementados. A camada Lua é composta por uma série de módulos de *script*. O motor de jogos 2D utilizado nesta dissertação é o C++Play[1], que foi adaptado para facilitar o presente trabalho. Apesar do uso do C++Play ter sido fundamental para esta dissertação, a camada Lua proposta foi projetada para ser adaptável a outros motores de jogos que possam fornecer funcionalidades semelhantes às que C++Play disponibiliza.

5.1
Arquitetura da camada Lua

A lógica do jogo corresponde às regras e ao funcionamento do jogo. *Scripting* é uma forma de linguagem de programação em alto nível que permite ao usuário personalizar ainda mais o jogo, adicionando novos comportamentos dos personagens, animações, reações físicas, entre outros.

Através das classes implementadas e externadas para a linguagem de *script* Lua, diversos objetos e funções foram criadas para a utilização e criação do jogo. Todos os objetos e funções criados podem ser utilizados na linguagem Lua como se fossem objetos implementados na própria linguagem. Para isto, o motor se comunica com a camada Lua como mostra a Figura 5.1. Os componentes da Figura 5.1 são:

- **Jogador:** controla os movimentos do personagem e define a área que ele pode andar.
- **Seleciona ambiente:** controla a escolha do ambiente feita pelo jogador e verifica se jogador tem acesso ao ambiente escolhido. Um ambiente é um local do mundo do jogo (e. g. um restaurante);
- **Ambientes:** define os ambientes, carrega a imagem de *background* de determinado ambiente e determina os objetos ativos e interativos;

[1]O C++Play é um *software* ainda em desenvolvimento pelo laboratório VisionLab/ICAD da PUC-Rio, pelo menos na época da confecção da presente dissertação (Agosto/2012)

- **Professor:** o usuário define todo o jogo: imagens, posições dos objetos ativos e interativos, texto do exercício, regra que testa o resultado do exercício, porcentagem de cada exercício e a regra para liberar o acesso a determinadas áreas ou ambientes;
- **Objetos:** define os objetos que o jogador utiliza para responder os exercícios e controla a ação de cada um destes objetos;
- **Correção:** verifica, através da regra definida pelo professor, se a resposta do jogador está correta;
- **Resposta:** informa, através de uma tela de parabéns ou erro, se a resposta do exercício está correta.

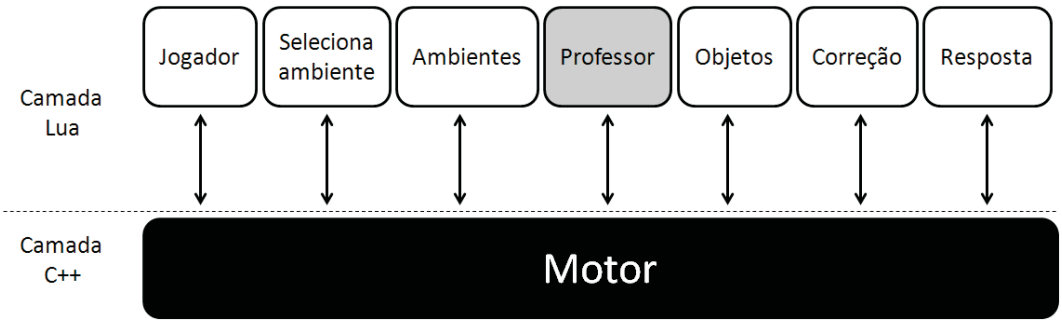

Figura 5.1: Arquitetura de integração do C++Play com Lua. O *script* "professor"é o único que o educador precisa escrever. Os outros *scripts* são mais técnicos e requerem a ajuda de um programador.

Para que a integração entre a linguagem de programação utilizada no motor de jogos (C++Play) e a linguagem de *script* Lua seja possível, o motor de jogos precisa possuir componentes disponibilizados pela linguagem Lua para que as funcionalidades do *script* possam ser acessadas no programa.

O primeiro passo para a integração entre C++ e Lua é o desenvolvimento dos objetos necessários para que a linguagem Lua possa ter acesso a rotinas implementadas no motor de jogos, pois para ter acesso aos serviços oferecidos por C++ é preciso exportar as funcionalidades de C++ para Lua.

Para a realização desta tarefa é utilizada a ferramenta *tolua* [54], a qual, através de cabeçalhos implementados em C++, exporta os arquivos fonte necessários para a inclusão no projeto. Estes arquivos têm a função de informar a linguagem de *script* sobre todos os métodos e objetos disponíveis no motor de jogos que podem ser utilizados no *script*.

Tolua [54] é uma ferramenta que simplifica a integração de C++ com Lua. Para usar *tolua*, primeiro é criado um arquivo com extensão .pkg (`C++Play_Lua_Bind.pkg`) listando as variáveis, funções, classes e métodos que

podem ser exportadas para o ambiente Lua. Depois é executado o arquivo .bat que contém o comando para gerar o código de integração a partir do arquivo com extensão .pkg. Então *tolua* analisa o arquivo e cria um arquivo C++ que liga automaticamente C++ com Lua. Assim, todos os objetos e funções criados podem ser utilizados na linguagem Lua, como se fossem objetos implementados na própria linguagem.

Com isto é possível controlar o funcionamento do programa durante sua execução, permitindo que o educador crie a lógica de execução do jogo, utilizando *scripts*.

Após a geração do código de integração (`C++Play_Lua_Bind.cpp`) o motor de jogos realiza a ligação entre C++ e Lua através do método ilustrado no código 5.1.

Código 5.1: função de *binding* gerada pelo *tolua*

```
1  void GameEngine::Lua_Init(){
2      ...
3      luaopen_C++Play_Lua_Bind(GameEngine::LUA_STATE);
4      ...
5  }
```

Com isto, durante a execução do *script*, objetos contidos no programa hospedeiro podem ser utilizados, e até mesmo instanciados, sem que seja necessária a implementação de rotinas no motor de jogos, possibilitando que um mesmo programa possa realizar diferentes tarefas sem que seja necessária a recompilação ou até mesmo mudanças no programa original.

Assim, Lua passa a ser o controlador do jogo (o cliente) e o código C++ funciona apenas como servidor, implementando de forma eficiente os serviços demandados por Lua. Nesse caso, há uma grande flexibilidade com o uso da linguagem de *script*, pois os programadores C++ ficam responsáveis por implementar o motor de jogos (estruturação e renderização de cenas, simulação física, algoritmos de inteligência artificial e gerenciamento de sons) e os "programadores" Lua ficam responsáveis por criar o roteiro, a história e o comportamento dos personagens.

Sendo assim, o fluxo de execução do jogo foi definido com os *scripts*. Os *scripts* são chamados de acordo com as interações do usuário no jogo. A cada iteração do *loop*, o motor de jogos realiza uma chamada à função *OnUpdate* implementada nos *scripts* de cada um dos objetos. Um *script* será executado toda vez que o usuário clicar em um objeto interativo, apresentando uma atividade pedagógica.

5.2 *Script* professor

Conforme ilustra a Figura 5.2, o mundo do jogo é organizado em termos de ambientes, áreas e objetos. O *script* "professor.Lua" (Apêndice A), escrito pelo educador, personaliza todos estes componentes do mundo.

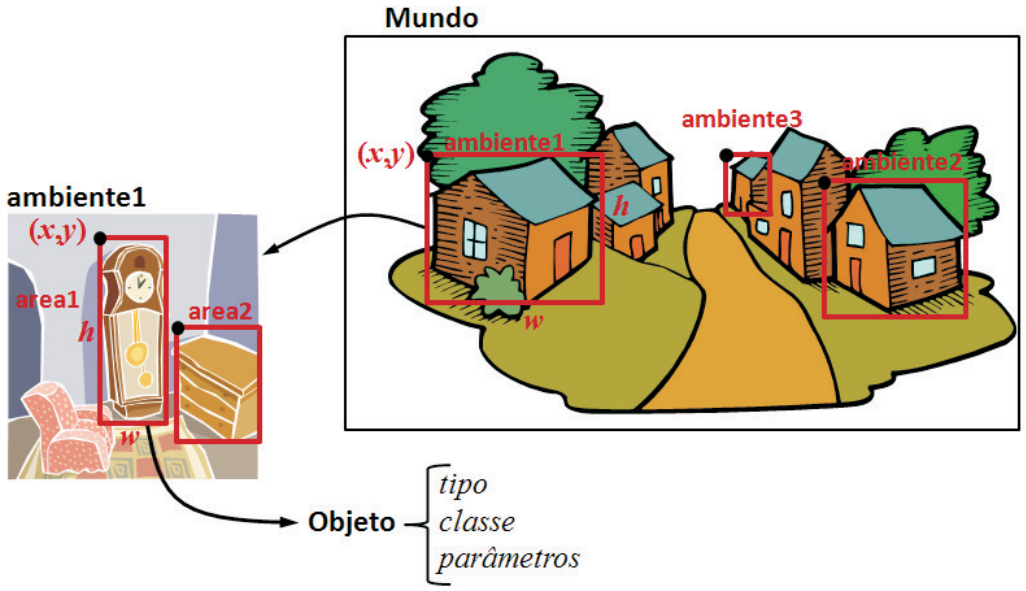

Figura 5.2: Organização em ambientes, áreas e objetos. (x,y) são as coordenadas de posicionamento, w é a largura e h é a altura (todos em pixels)

No script "professor.Lua", o educador define as imagens do jogo, determina as condições para liberação de cada ambiente/área, seleciona objetos e define os seus parâmetros. Os outros scripts definem as configurações gerais do jogo, tais como: número de ambientes, número de áreas em cada ambiente e classes de objetos (e.g. jogo de memória, *quiz*, solução de problema, ...). Cada objeto possui um tipo, classe e parâmetros. O tipo define se o objeto é ativo ou interativo, a classe define o estilo do jogo (e.g. jogo de memória, *quiz*, solução de problema) e os parâmetros definem a posição e o tamanho do objeto. Tipo, classe e parâmetros de objetos são detalhados no Capítulo 6.

O educador pode criar a regra mais apropriada para verificar se a resposta do aluno relativa a um objeto está correta. Por exemplo, para o objeto da Figura 5.3, ele pode definir a regra através do código 5.2.

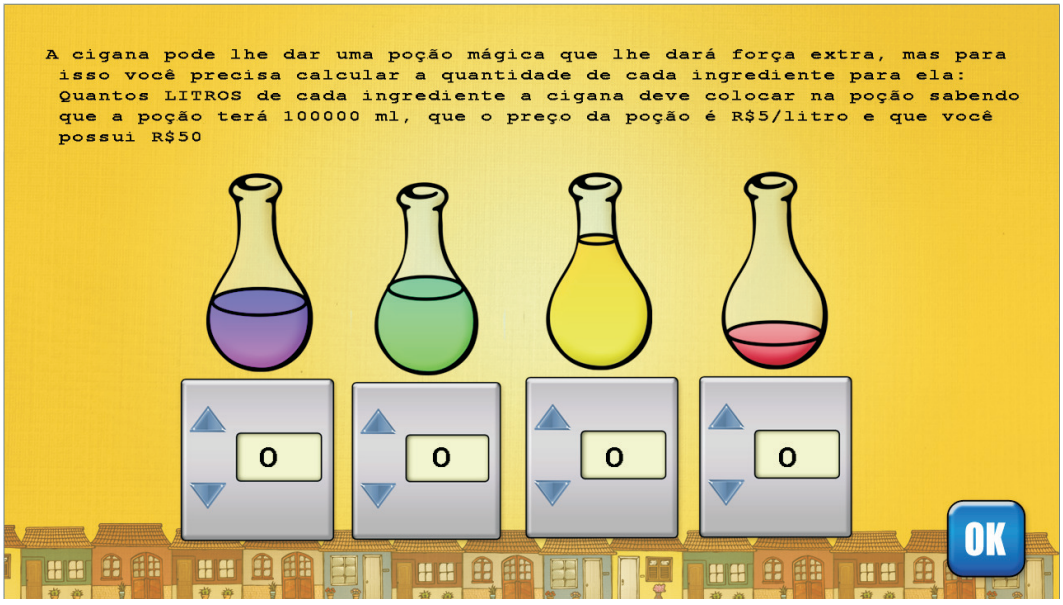

Figura 5.3: Objeto de solução de problema cuja função de verificação é definida pelo código 5.2.

Código 5.2: função para verificar se a resposta do jogador está correta

```
1  function resposta
2      for i = 1, quantidade, 1 do
3          soma = soma + respostaAluno[i]
4      end
5      if (soma == budget / preco)
6          resposta = true
7      else
8          resposta = false
9      end
10 end
```

A regra para liberar o acesso das áreas ou ambientes funciona da seguinte maneira: O educador utiliza a porcentagem de acerto de cada exercício para montar cada regra e cria as regras da maneira que desejar. Por exemplo: O ambiente 1 tem dois exercícios e cada exercício vale 50%. Para o jogador ter acesso ao segundo exercício, ele precisa acertar o exercício do objeto 1 e ele só tem acesso ao ambiente 2 se acertar todos os desafios do primeiro ambiente.

Estas características mostram que Lua oferece uma grande flexibilidade à aplicação, pois permite ao usuário controlar a aplicação externamente, apenas editando um arquivo texto (professor.lua).

5.3 Generalidade do sistema

Apesar do educador ter a flexibilidade de personalizar o jogo (através do *script* professor.Lua), criando novos desafios, novos problemas e imagens, ele está limitado às configurações gerais do jogo definidas pelos outros módulos

de *script*. O *framework* desenvolvido neste trabalho possui um número fixo de ambientes (2 ambientes), objetos interativos (2 objetos em cada ambiente) e objetos ativos (1 objeto em um dos ambientes). Estas limitações podem ser resolvidas através da programação destes outros *scripts*. Entretanto, esta tarefa requer um nível de conhecimento de programação um pouco mais elevado.

Uma outra limitação refere-se ao fato da camada Lua estar totalmente amarrada à interface das classes do C++Play. O uso desta camada com um outro motor de jogos só seria possível se o C++Play fosse transformado num "wrapper" para outro motor. Em outras palavras, a generalidade de uso da camada só poderia existir se fosse retirada toda a implementação do C++Play, mantendo apenas as assinaturas das classes e se esta implementação fosse trocada por chamadas à implementação do outro motor de jogos. Para isso, pode-se utilizar o *Adapter*, um padrão de projeto de software, utilizado para "adaptar" a interface de uma classe. A Figura 5.4 associa a comunicação entre a camada Lua e qualquer outro motor de jogos diferente do C++Play a uma situação em que é necessário conectar um plugue a uma tomada incompatível. Nesse caso, o encaixe seria impossível se não fosse a utilização de um adaptador.

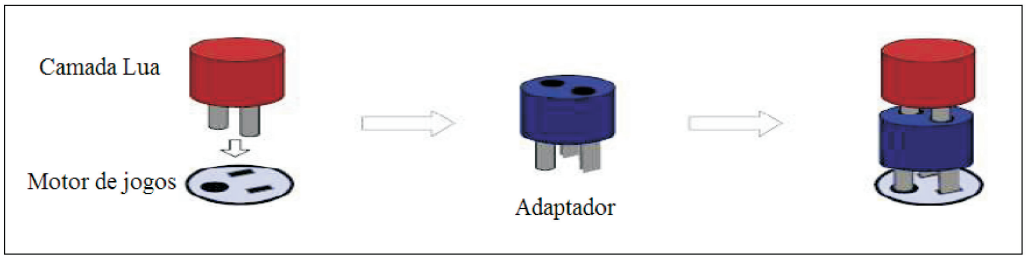

Figura 5.4: Representação do padrão *Adapter*. Para utilizar a camada Lua com outro motor de jogos, diferente do C++Play, é necessário utilizar o *Adapter* [55].

A intenção do padrão *Adapter*, segundo a referência [49], é converter a interface de uma classe em outra interface, esperada pelos clientes. O *Adapter* permite que classes com interfaces incompatíveis trabalhem em conjunto – o que, de outra forma, seria impossível [49]. Ainda segundo a referência [49], o *Adapter* deve ser usado quando for desejável utilizar uma classe existente, mas sua interface diferir da interface necessária ou quando deseja-se criar classes reutilizáveis que colaborem com classes que não tenham, necessariamente, interfaces compatíveis. O Adapter permite que classes com interfaces incompatíveis possam interagir e permite também que um objeto cliente utilize serviços de outros objetos com interfaces diferentes por meio de uma interface única. Enfim, com o *Adapter* é possível escolher outro

motor de jogos para utilizar a camada Lua implementada neste trabalho. Entretanto, este padrão não foi implementado na presente dissertação, visto que o objetivo é o desenvolvimento de um *framework* para o desenvolvimento de jogos educativos 2D de aventura.