Cube Editor

Documentação Técnica — C++ & Lua

1. A Aplicação

O Cube Editor é uma aplicação gráfica interativa construída sobre OpenGL e GLUT que exibe um cubo tridimensional com seis faces independentes sobre um fundo de céu estrelado animado. O usuário pode rotacionar o cubo com as teclas WASD, selecionar faces com o mouse, aplicar e misturar cores aditivamente (vermelho, azul, verde formando roxo, amarelo, ciano e branco), carregar imagens para qualquer face e controlar zoom e rotação da textura aplicada. Um monitor de performance pode ser ativado em tempo de compilação, abrindo uma segunda janela com métricas de FPS, tempo de frame, uso de memória e estatísticas de cache de texturas.

O projeto foi desenvolvido como exercício de integração entre duas linguagens, mantendo uma separação clara de responsabilidades: C++ cuida de toda a camada gráfica e estrutural, enquanto Lua concentra as regras de negócio dinâmicas, lógica de animação e processamento de entrada.

2. Divisão de Responsabilidades entre as Linguagens

C++ é responsável pela inicialização do contexto OpenGL, criação das janelas GLUT, renderização do cubo 3D (vértices, quads, projeção de perspectiva), color picking por face via framebuffer, carregamento de texturas com stb_image, gerenciamento do estado Lua e pela ponte que conecta as duas linguagens.

Lua controla toda a lógica que se beneficia de ser modificável sem recompilação: a geração e animação das estrelas do fundo (usando um gerador congruencial linear determinístico para reprodutibilidade), o mapeamento de teclas WASD a incrementos de rotação, a lógica de mistura aditiva de cores e o controle de padrão e foto de cada face do cubo. Esses quatro domínios vivem, respectivamente, nos arquivos *background.lua*, *controle.lua*, *mixer.lua* e *faces.lua*.

3. Por que Lua

A escolha de Lua como linguagem de scripting foi motivada por dois fatores que se reforçam mutuamente. O primeiro é técnico: Lua foi projetada desde o início para ser embutida em aplicações C e C++. Sua API C pública é enxuta, estável e documentada, e a própria implementação de referência é escrita em C padrão, o que torna a integração direta sem a necessidade de wrappers pesados ou geradores de código. O segundo fator é pessoal: sempre houve interesse em trabalhar com Lua, seja pela sua adoção no desenvolvimento de jogos e ferramentas interativas, seja pela elegância de seu modelo de dados baseado em tabelas. Este projeto foi a oportunidade de colocar esse interesse em prática de forma concreta, explorando a integração real entre as duas linguagens num contexto gráfico.

4. A Interface entre C++ e Lua

A integração é feita inteiramente pela API C do Lua (liblua), sem bibliotecas de binding de terceiros. A classe **LuaBridge** encapsula um ponteiro opaco para o *lua_State*, seguindo o padrão PIMPL para que os cabeçalhos Lua não precisem ser expostos fora do módulo bridge. Todos os arquivos do projeto que dependem do bridge incluem apenas *lua_bridge.h*, que não declara nada do Lua diretamente.

O ciclo de uma chamada segue sempre o mesmo protocolo de pilha Lua: empurrar a função global com *lua_getglobal*, empurrar os argumentos com *lua_push**, chamar com *lua_pcall* especificando número de argumentos e retornos esperados, ler os retornos com *lua_to** e limpar a pilha com *lua_pop*. Erros de pcall são impressos em stderr e a pilha é corrigida, garantindo que o estado Lua permaneça válido mesmo diante de falhas em scripts.

Exemplo — leitura da tabela flat de estrelas:

```
// C++ chama getStarPositions(t) em Lua
lua_getglobal(L, "getStarPositions");
lua_pushnumber(L, t);             // argumento: tempo em segundos
lua_pcall(L, 1, 1, 0);             // 1 arg, 1 retorno
int n = (int)lua_rawlen(L, -1);    // tamanho da tabela retornada
for (int i = 1; i <= n; ++i) {
    lua_rawgeti(L, -1, i);
    out.push_back((float)lua_tonumber(L, -1));
    lua_pop(L, 1);
}
lua_pop(L, 1);                     // remove a tabela
```

A função Lua retorna uma tabela flat com 5 valores por estrela (x, y, r, g, b). O C++ itera sobre ela e chama glVertex2f / glColor3f diretamente, sem alocação adicional.

Os scripts Lua são carregados em tempo de execução com *luaL_dofile*, procurando primeiro no subdiretório *lua/* e depois no diretório de trabalho atual. Isso permite editar qualquer script sem recompilar a aplicação. A inicialização do interpretador (*luaL_newstate* + *luaL_openlibs*) e o fechamento (*lua_close*) são gerenciados pelo destrutor de LuaBridge, seguindo RAII.

Nenhum recurso do C++ ou do Lua precisou ser modificado para viabilizar a integração: a API pública do Lua já oferece tudo o que é necessário. O único cuidado arquitetural relevante foi isolar o lua_State atrás de PIMPL para evitar que o include path do Lua vazasse para o restante do projeto, mantendo os headers de interface limpos.