Capítulo 1: Introdução

Este é o código-fonte de `weaver`, uma *engine* (ou motor) para desenvolvimento de jogos feita em C utilizando-se da técnica de programação literária.

Um motor é um conjunto de bibliotecas e programas utilizado para facilitar e abstrair o desenvolvimento de um jogo. Jogos de computador, especialmente jogos em 3D são programas sofisticados demais e geralmente é inviável começar a desenvolver um jogo do zero. Um motor fornece uma série de funcionalidades genéricas que facilitam o desenvolvimento, tais como gerência de memória, renderização de gráficos bidimensionais e tridimensionais, um simulador de física, detector de colisão, suporte à animações, som, fontes, linguagem de script e muito mais.

Programação literária é uma técnica de desenvolvimento de programas de computador que determina que um programa deve ser especificado primariamente por meio de explicações didáticas de seu funcionamento. Desta forma, escrever um software que realiza determinada tarefa não deveria ser algo diferente de escrever um livro que explica didaticamente como resolver tal tarefa. Tal livro deveria apenas ter um rigor maior combinando explicações informais em prosa com explicações formais em código-fonte. Programas de computador podem então extrair a explicação presente nos arquivos para gerar um livro ou manual (no caso, este PDF) e também extrair apenas o código-fonte presente nele para construir o programa em si. A tarefa de montar o programa na ordem certa é de responsabilidade do programa que extrai o código. Um programa literário deve sempre apresentar as coisas em uma ordem acessível para humanos, não para máquinas.

Por exemplo, para produzir este PDF, utiliza-se um programa chamado TEX, o qual por meio do formato MAGITEX instalado, compreende código escrito em um formato específico de texto e o formata de maneira adequada. O TEX gera um arquivo no formato DVI, o qual é convertido para PDF. Para produzir o motor de desenvolvimento de jogos em si utiliza-se sobre os mesmos arquivos fonte um programa chamado CTANGLE, que extrai o código C (além de um punhado de códigos GLSL) para os arquivos certos. Em seguida, utiliza-se um compilador como GCC ou CLANG para produzir os executáveis. Felizmente, há `Makefiles` para ajudar a cuidar de tais detalhes de construção.

Os pré-requisitos para se compreender este material são ter uma boa base de programação em C e ter experiência no desenvolvimento de programas em C para Linux. Alguma noção do funcionamento de OpenGL também ajuda.

1.1 - Copyright e licenciamento

Weaver é desenvolvida pelo programador Thiago "Harry" Leucz Astrizi. Abaixo segue a licença do software:

A tradução não-oficial da licença é:

A versão completa da licença pode ser obtida junto ao código-fonte Weaver ou consultada no link mencionado.

1.2 - Filosofia Weaver

Estes são os princípios filosóficos que guiam o desenvolvimento deste software. Qualquer coisa que vá de encontro à eles devem ser tratados como *bugs*.

1- Software é conhecimento sobre como realizar algo escrito em linguagens formais de computadores. O conhecimento deve ser livre para todos. Portanto, Weaver deverá ser um software livre e deverá também ser usada para a criação de jogos livres.

A arte de um jogo pode ter direitos de cópia. Ela deveria ter uma licença permisiva, pois arte é cultura, e portanto, também não deveria ser algo a ser tirado das pessoas. Mas weaver não tem como impedi-lo de licenciar a arte de um jogo da forma que for escolhida. Mas como Weaver funciona injetando estaticamente seu código em seu jogo e Weaver está sob a licença GPL, isso significa que seu jogo também deverá estar sob esta mesma licença (ou alguma outra compatível).

Basicamente isso significa que você pode fazer quase qualquer coisa que quiser com este software. Pode copiá-lo. Usar seu código-fonte para fazer qualquer coisa que queira (assumindo as responsabilidades). Passar para outras pessoas. Modificá-lo. A única coisa não permitida é produzir com ele algo que não dê aos seus usuários exatamente as mesmas liberdades.

As seguintes quatro liberdades devem estar presentes em Weaver e nos jogos que ele desenvolve:

Liberdade 0: A liberdade para executar o programa, para qualquer propósito.

Liberdade 1: A liberdade de estudar o software.

Liberdade 2: A liberdade de redistribuir cópias do programa de modo que você possa ajudar ao seu próximo.

Liberdade 3: A liberdade de modificar o programa e distribuir estas modificações, de modo que toda a comunidade se beneficie.

2- Weaver deve estar bem-documentado.

As quatro liberdades anteriores não são o suficiente para que as pessoas realmente possam estudar um software. Código ofuscado ou de difícil compreensão dificulta que as pessoas a exerçam. Weaver deve estar completamente documentada. Isso inclui explicação para todo o código-fonte que o projeto possui. O uso de MAGITEX e CWEB é um reflexo desta filosofia.

Algumas pessoas podem estranhar também que toda a documentação do código-fonte esteja em português. Estudei por anos demais em universidade pública e minha educação foi paga com dinheiro do povo brasileiro. Por isso acho que minhas contribuições devem ser pensadas sempre em como retribuir à isto. Por isso, o português brasileiro será o idioma principal na escrita deste software.

Infelizmente, isso tamém conflita com o meu desejo de que este projeto seja amplamente usado no mundo todo. Geralmente espera-se que código e documentação esteja em inglês. Para lidar

com isso, pretendo que a documentação on-line e guia de referência das funções esteja em inglês. Os nomes de funções e de variáveis estarão em inglês. Mas as explicações aqui serão em português.

Com isso tento conciliar as duas coisas, por mais difícil que isso seja.

3- Weaver deve ter muitas opções de configuração para que possa atender à diferentes necessidades.

É terrível quando você tem que lidar com abominações como:

Arquivo: `/tmp/dummy.c`**:**

```
  CreateWindow("nome da classe", "nome da janela", WS_BORDER | WS_CAPTION |
Ψ      WS_MAXIMIZE, 20, 20, 800, 600, handle1, handle2, handle3, NULL);
```

Cada projeto deve ter um arquivo de configuração e muito da funcionalidade pode ser escolhida lá. Escolhas padrão sãs devem ser escolhidas e estar lá, de modo que um projeto funcione bem mesmo que seu autor não mude nada nas configurações. E concentrando configurações em um arquivo, retiramos complexidade das funções. As funções não precisam então receber mais de 10 argumentos diferentes e não é necessário também ficar encapsulando os 10 argumentos em um objeto de configuração, o qual é mais uma distração que solução para a complexidade.

Em todo projeto Weaver haverá um arquivo de configuração `conf/conf.h`, que modifica o funcionamento do motor. Como pode ser deduzido pela extensão do nome do arquivo, ele é basicamente um arquivo de cabeçalho C onde poderão ter vários `#define` s que modificarão o funcionamento de seu jogo.

4- Weaver não deve tentar resolver problemas sem solução. Ao invés disso, é melhor propor um acordo mútuo entre usuários.

Computadores tornam-se coisas complexas porque pessoas tentam resolver neles problemas insolúveis. É como tapar o sol com a peneira. Você na verdade consegue fazer isso. Junte um número suficientemente grande de peneiras, coloque uma sobre a outra e você consegue gerar uma sombra o quão escura se queira. Assim são os sistemas modernos que usamos nos computadores.

Como exemplo de tais tentativas de solucionar problemas insolúveis, temos a tentativa de fazer com que Sistemas Operacionais proprietários sejam seguros e livres de vírus, garantir privacidade, autenticação e segurança sobre HTTP e até mesmo coisas como o gerenciamento de memória. Pode-se resolver tais coisas apenas adicionando camadas e mais camadas de complexidade, e mesmo assim, não funcionará em realmente 100% dos casos.

Quando um problema não tem uma solução satisfatória, isso jamais deve ser escondido por meio de complexidades que tentam amenizar ou sufocar o problema. Ao invés disso, a limitação natural da tarefa deve ficar clara para o usuário, e deve-se trabalhar em algum tipo de comportamento que deve ser seguido pela engine e pelo usuário para que se possa lidar com o problema combinando os esforços de humanos e máquinas naquilo que cada um dos dois é melhor em fazer.

5- Um jogo feito usando Weaver deve poder ser instalado em um computador simplesmente distribuindo-se um instalador, sem necessidade de ir atrás de dependências.

Este é um exemplo de problema insolúvel mencionado anteriormente. Para isso a API Weaver é inserida estaticamente em cada projeto Weaver ao invés de ser na forma de bibliotecas compartilhadas. Mesmo assim ainda haverão dependências externas. Iremos então tentar minimizar elas e garantir que as duas maiores distribuições Linux no DistroWatch sejam capazes de rodar os jogos sem dependências adicionais além daquelas que já vem instaladas por padrão.

6- Wever deve ser fácil de usar. Mais fácil que a maioria das ferramentas já existentes.

Isso é obtido mantendo as funções o mais simples possíveis e fazendo-as funcionar seguindo padrões que são bons o bastante para a maioria dos casos. E caso um programador saiba o que está fazendo, ele deve poder configurar tais padrões sem problemas por meio do arquivo `conf/conf.h`.

Desta forma, uma função de inicialização poderia se chamar `Winit()` e não precisar de nenhum argumento. Coisas como gerenciar a projeção das imagens na tela devem ser transparentessem precisar de uma função específica após os objetos que compõe o ambiente serem definidos.

1.3 - Instalando Weaver

Para instalar Weaver em um computador, assumindo que você está fazendo isso à partir do

código-fonte, basta usar o comando `make` e `make install` (o segundo comando como *root*).

Atualmente, os seguintes programas são necessários para se compilar Weaver:

ctangle ou **notangle**: Extrai o código C dos arquivos de `cweb/`.

clang ou **gcc**: Um compilador C que gera executáveis à partir de código C.

make: Interpreta e executa comandos do Makefile.

Os dois primeiros programas podem vir em pacotes chamados de **cweb** ou **noweb**. Adicionalmente, os seguintes programas são necessários para se gerar a documentação:

TEX e MAGITEX: Usado para ler o código-fonte CWEB e gerar um arquivo DVI.

dvipdf: Usado para converter um arquivo `.dvi` em um `.pdf`.

graphviz: Gera representações gráficas de grafos.

Além disso, para que você possa efetivamente usar Weaver criando seus próprios projetos, você também poderá precisar de:

emscripten: Compila código C para Javascript e assim rodar em um navegador.

opengl: Permite gerar executáveis nativos com gráficos em 3D.

xlib: Permite gerar executáveis nativos gráficos.

xxd: Gera representação hexadecimal de arquivos. Insere o código dos shaders no programa. Por motivos obscuros, algumas distribuições trazem este último programa no mesmo pacote do **vim**.

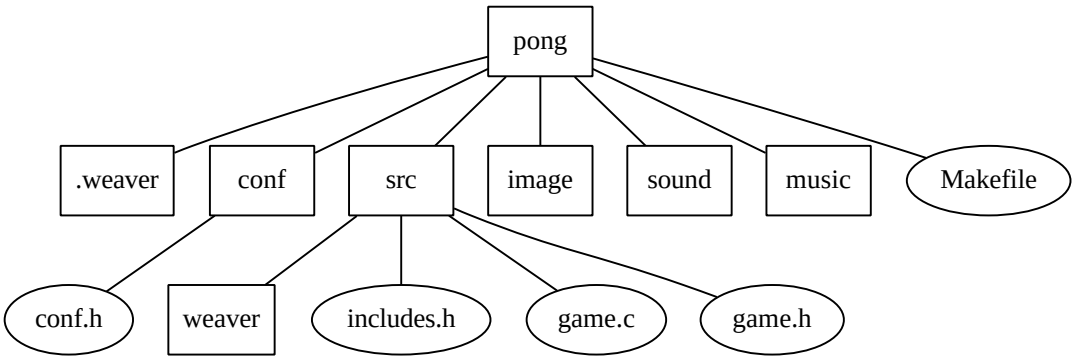
1.4 - O programa `weaver`

Weaver é uma engine para desenvolvimento de jogos que na verdaade é formada por várias coisas diferentes. Quando falamos em código do Weaver, podemos estar nos referindo à código de algum dos programas executáveis usados para se gerenciar a criação de seus jogos, podemos estar nos referindo ao código da API Weaver que é inserida em cada um de seus jogos ou então podemos estar nos referindo ao código de algum de seus jogos.

Para evitar ambigüidades, quando nos referimos ao programa executável, nos referiremos ao **programa Weaver**. Seu código-fonte será apresentado inteiramente neste capítulo. O programa é usado simplesmente para criar um novo projeto Weaver. E um projeto é um diretório com vários arquivos de desenvolvimento contendo código-fonte e multimídia. Por exemplo, o comando abaixo cria um novo projeto de um jogo chamado `pong`:

```
weaver pong
```

A árvore de diretórios exibida parcialmente abaixo é o que é criado pelo comando acima (diretórios são retângulos e arquivos são círculos):



Quando nos referimos ao código que é inserido em seus projetos, falamos do código da **API Weaver**. Seu código é sempre inserido dentro de cada projeto no diretório `src/weaver/`. Você terá aceso à uma cópia de seu código em cada novo jogo que criar, já que tal código é inserido estaticamente em seus projetos.

Já o código de jogos feitos com Weaver são tratados por **projetos Weaver**. É você quem escreve o seu código, ainda que a engine forneça como um ponto de partida o código inicial de

inicialização, criação de uma janela e leitura de eventos do teclado e mouse.

1.4.1- Casos de Uso do Programa Weaver

Além de criar um projeto Weaver novo, o programa Weaver tem outros casos de uso. Eis a lista deles:

Caso de Uso 1: Mostrar mensagem de ajuda de criação de novo projeto: Isso deve ser feito toda vez que o usuário estiver fora do diretório de um Projeto Weaver e ele pedir ajuda explicitamente passando o parâmetro `--help` ou quando ele chama o programa sem argumentos (caso em que assumiremos que ele não sabe o que fazer e precisa de ajuda).

Caso de Uso 2: Mostrar mensagem de ajuda do gerenciamento de projeto: Isso deve ser feito quando o usuário estiver dentro de um projeto Weaver e pedir ajuda explicitamente com o argumento `--help` ou se invocar o programa sem argumentos (caso em que assumimos que ele não sabe o que está fazendo e precisa de ajuda).

Caso de Uso 3: Mostrar a versão de Weaver instalada no sistema: Isso deve ser feito toda vez que Weaver for invocada com o argumento `--version`.

Caso de Uso 4: Atualizar um projeto Weaver existente: Para o caso de um projeto ter sido criado com a versão 0.4 e tenha-se instalado no computador a versão 0.5, por exemplo. Para atualizar, basta passar como argumento o caminho absoluto ou relativo de um projeto Weaver. Independente de estarmos ou não dentro de um diretório de projeto Weaver. Atualizar um projeto significa mudar os arquivos com a API Weaver para que reflitam versões mais recentes.

Caso de Uso 5: Criar novo módulo em projeto Weaver: Para isso, devemos estar dentro do diretório de um projeto Weaver e devemos passar como argumento um nome para o módulo que não deve começar com pontos, traços, nem ter o mesmo nome de qualquer arquivo de extensão `.c` presente em `src/` (pois para um módulo de nome XXX, serão criados arquivos `src/XXX.c` e `src/XXX.h`).

Caso de Uso 6: Criar um novo projeto Weaver: Para isso ele deve estar fora de um diretório Weaver e deve passar como primeiro argumento um nome válido e não-reservado para seu novo projeto. Um nome válido deve ser qualquer um que não comece com ponto, nem traço, que não tenha efeitos negativos no terminal (tais como mudar a cor de fundo) e cujo nome não pode conflitar com qualquer arquivo necessário para o desenvolvimento (por exemplo, não deve-se poder criar um projeto chamado `Makefile`).

1.4.2- Variáveis do Programa Weaver

O comportamento de Weaver deve depender das seguintes variáveis:

`inside_weaver_directory` : Indicará se o programa está sendo invocado de dentro de um projeto Weaver.

`argument` : O primeiro argumento, ou NULL se ele não existir

`project_version_major` : Se estamos em um projeto Weaver, qual o maior número da versão do Weaver usada para gerar o projeto. Exemplo: se a versão for 0.5, o número maior é 0. Em versões de teste, o valor é sempre 0.

`project_version_minor` : Se estamos em um projeto Weaver, o valor do menor número da versão do Weaver usada para gerar o projeto. Exemplo, se a versão for 0.5, o número menor é 5. Em versões de teste o valor é sempre 0.

`weaver_version_major` : O número maior da versão do Weaver sendo usada no momento.

`weaver_version_minor` : O número menor da versão do Weaver sendo usada no momento.

`arg_is_path` : Se o primeiro argumento é ou não um caminho absoluto ou relativo para um projeto Weaver.

`arg_is_valid_project` : Se o argumento passado seria válido como nome de projeto Weaver.

`arg_is_valid_module` : Se o argumento passado seria válido como um novo módulo no projeto Weaver atual.

`project_path` : Se estamos dentro de um diretório de projeto Weaver, qual o caminho para a sua base (onde há o Makefile)

`have_arg` : Se o programa é invocado com argumento.

`shared_dir` : Deverá armazenar o caminho para o diretório onde estão os arquivos compartilhados da instalação de Weaver. Por padrão, será igual à "`/usr/share/weaver`", mas caso exista a variável de ambiente `WEAVER_DIR`, então este será considerado o endereço dos arquivos compartilhados.

`author_name` , `project_name` e `year` : Conterão respectivamente o nome do usuário que está invocando Weaver, o nome do projeto atual (se estivermos no diretório de um) e o ano atual. Isso será importante para gerar as mensagens de Copyright em novos projetos Weaver.

`return_value` : Que valor o programa deve retornar caso o programa seja interrompido no momento atual.

1.4.3- Estrutura Geral do Programa Weaver

Todas estas variáveis serão inicializadas no começo, e se precisar serão desalocadas no fim do programa, que terá a seguinte estrutura:

Arquivo: `src/weaver.c`**:**

<Seção a ser Inserida: Cabeçalhos Incluídos no Programa Weaver>

<Seção a ser Inserida: Macros do Programa Weaver>

<Seção a ser Inserida: Funções auxiliares Weaver>

```
int main(int argc, char **argv){
  int return_value = 0; /* Valor de retorno. */
  bool inside_weaver_directory = false, arg_is_path = false,
    arg_is_valid_project = false, arg_is_valid_module = false,
    have_arg = false; /* Variáveis booleanas. */
  unsigned int project_version_major = 0, project_version_minor = 0,
    weaver_version_major = 0, weaver_version_minor = 0,
    year = 0;
  char *argument = NULL, *project_path = NULL, *shared_dir = NULL,
    *author_name = NULL, *project_name = NULL; /* Strings UTF-8 */
```

<Seção a ser Inserida: Inicialização>

<Seção a ser Inserida: Caso de uso 1: Imprimir ajuda de criação de projeto>

<Seção a ser Inserida: Caso de uso 2: Imprimir ajuda de gerenciamento>

<Seção a ser Inserida: Caso de uso 3: Mostrar versão>

<Seção a ser Inserida: Caso de uso 4: Atualizar projeto Weaver>

<Seção a ser Inserida: Caso de uso 5: Criar novo módulo>

<Seção a ser Inserida: Caso de uso 6: Criar novo projeto>

```
END_OF_PROGRAM:
```

<Seção a ser Inserida: Finalização>

```
  return return_value;
}
```

1.4.4- Macros do Programa Weaver

O programa precisará de algumas macros. A primeira delas deverá conter uma string com a versão do programa. A versão pode ser formada só por letras (no caso de versões de teste) ou por um número seguido de um ponto e de outro número (sem espaços) no caso de uma versão final do programa.

Para a segunda macro, observe que na estrutura geral do programa vista acima existe um rótulo chamado `END_OF_PROGRAM` logo na parte de finalização. Uma das formas de chegarmos

lá é por meio da execução normal do programa, caso nada dê errado. Entretanto, no caso de um erro, nós podemos também chegar lá por meio de um desvio incondicional após imprimirmos a mensagem de erro e ajustarmos o valor de retorno do programa. A responsabilidade de fazer isso será da segunda macro.

Por outro lado, podemos também querer encerrar o programa previamente, mas sem que tenha havido um erro. A responsabilidade disso é da terceira macro que definimos.

Seção: Macros do Programa Weaver:

```
#define VERSION "Alpha"
#define ERROR() {perror(NULL); return_value = 1; goto END_OF_PROGRAM;}
#define END() goto END_OF_PROGRAM;
```

1.4.5- Cabeçalhos do Programa Weaver

Seção: Cabeçalhos Incluídos no Programa Weaver:

```
#include <sys/types.h> // stat, getuid, getpwuid, mkdir
#include <sys/stat.h> // stat, mkdir
#include <stdbool.h> // bool, true, false
#include <unistd.h> // get_current_dir_name, getcwd, stat, chdir, getuid
#include <string.h> // strcmp, strcat, strcpy, strncmp
#include <stdlib.h> // free, exit, getenv
#include <dirent.h> // readdir, opendir, closedir
#include <libgen.h> // basename
#include <stdarg.h> // va_start, va_arg
#include <stdio.h> // printf, fprintf, fopen, fclose, fgets, fgetc, perror
#include <ctype.h> // isanum
#include <time.h> // localtime, time
#include <pwd.h> // getpwuid
```

1.4.6- Inicialização e Finalização do Programa Weaver

Inicializar Weaver significa inicializar as 14 variáveis que serão usadas para definir o seu comportamento.

1.4.6.1- Inicializando Variáveis `inside_weaver_directory` e `project_path`

A primeira das variáveis é `inside_weaver_directory`, que deve valer `false` se o programa foi invocado de fora de um diretório de projeto Weaver e `true` caso contrário.

Como definir se estamos em um diretório que pertence à um projeto Weaver? Simples. São diretórios que contém dentro de si ou em um diretório ancestral um diretório oculto chamado `.weaver`. Caso encontremos este diretório oculto, também podemos aproveitar e ajustar a variável `project_path` para apontar para o local onde ele está. Se não o encontrarmos, estaremos fora de um diretório Weaver e não precisamos mudar nenhum valor das duas variáveis, pois elas deverão permanecer com o valor padrão `NULL`.

Em suma, o que precisamos é de um loop com as seguintes características:

Invariantes: A variável `complete_path` deve sempre possuir o caminho completo do diretório `.weaver` se ele existisse no diretório atual.

Inicialização: Inicializamos tanto o `complete_path` para serem válidos de acordo com o diretório em que o programa é invocado.

Manutenção: Em cada iteração do loop nós verificamos se encontramos uma condição de finalização. Caso contrário, subimos para o diretório pai do qual estamos, sempre atualizando as variáveis para que o invariante continue válido.

Finalização: Interrompemos a execução do loop se uma das duas condições ocorrerem:

a) `complete_path == "/.weaver"` : Neste caso não podemos subir mais na árvore de diretórios, pois estamos na raiz do sistema de arquivos. Não encontramos um diretório `.weaver`. Isso significa que não estamos dentro de um projeto Weaver.

b) `complete_path == ".weaver"` : Neste caso encontramos um diretório `.weaver` e descobrimos que estamos dentro de um projeto Weaver. Podemos então atualizar a variável `project_path` para o diretório em que paramos.

Para manipularmos o caminho da árvore de diretórios, usaremos uma função auxiliar que recebe como entrada uma string com um caminho na árvore de diretórios e apaga todos os últimos caracteres até apagar dois "/". Assim em "/home/alice/projeto/diretorio/" ele retornaria "/home/alice/projeto" efetivamente subindo um nível na árvore de diretórios:

Seção: Funções auxiliares Weaver:

```
void path_up(char *path){
  int erased = 0;
  char *p = path;
  while(*p != '\0') p ++; // Vai até o fim
  while(erased < 2 && p != path){
    p --;
    if(*p == '/') erased ++;
    *p = '\0'; // Apaga
  }
}
```

Note que caso a função receba uma string que não possua dois "/" em seu nome, obtemos um "buffer overflow" onde percorreríamos regiões de memória indevidas preenchendo-as com zero. Esta função é bastante perigosa, mas se limitarmos as strings que passamos para somente arquivos que não estão na raíz e diretórios diferentes da própria raíz que terminam sempre com "/", então não teremos problemas pois a restrição do número de barras será cumprida. Ex: "/etc/" e "/tmp/file.txt".

Para checar se o diretório `.weaver` existe, definimos `directory_exist(x)` como uma função que recebe uma string correspondente à localização de um arquivo e que deve retornar 1 se `x` for um diretório existente, -1 se `x` for um arquivo existente e 0 caso contrário:

Seção: Funções auxiliares Weaver (continuação):

```
int directory_exist(char *dir){
  struct stat s; // Armazena status se um diretório existe ou não.
  int err; // Checagem de erros
  err = stat(dir, &s); // .weaver existe?
  if(err == -1) return 0; // Não existe
  if(S_ISDIR(s.st_mode)) return 1; // Diretório
  return -1; // Arquivo
}
```

A última função auxiliar da qual precisaremos é uma função para concatenar strings. Ela deve receber um número arbitrário de srings como argumento, mas a última string deve ser uma string vazia. E irá retornar a concatenação de todas as strings passadas como argumento.

A função irá alocar sempre uma nova string, a qual deverá ser desalocada antes do programa terminar. Como exemplo, `concatenate("tes", " ", "te", "")` retorna `"tes te"` .

Seção: Funções auxiliares Weaver (continuação):

```
char *concatenate(char *string, ...){
  va_list arguments;
  char *new_string, *current_string = string;
  size_t current_size = strlen(string) + 1;
  char *realloc_return;
  va_start(arguments, string);
```

```
  new_string = (char *) malloc(current_size);
  if(new_string == NULL) return NULL;
  strcpy(new_string, string); // Copia primeira string

  while(current_string[0] != '\0'){ // Pára quando copiamos o "".
    current_string = va_arg(arguments, char *);
    current_size += strlen(current_string);
    realloc_return = (char *) realloc(new_string, current_size);
    if(realloc_return == NULL){
      free(new_string);
      return NULL;
    }
    new_string = realloc_return;
    strcat(new_string, current_string); // Copia próxima string
  }
  return new_string;
}
```

É importante lembrarmos que a função `concatenate` sempre deve receber como último argumento uma string vazia ou teremos um *buffer overflow*. Esta função também é perigosa e deve ser usada sempre tomando-se este cuidado.

Por fim, podemos escrever agora o código de inicialização. Começamos primeiro fazendo `complete_path` ser igual à `./.weaver/`:

Seção: Inicialização:

```
char *path = NULL, *complete_path = NULL;
path = getcwd(NULL, 0);
if(path == NULL) ERROR();
complete_path = concatenate(path, "/.weaver", "");
free(path);
if(complete_path == NULL) ERROR();
```

Agora iniciamos um loop que terminará quando `complete_path` for igual à `/.weaver` (chegamos no fim da árvore de diretórios e não encontramos nada) ou quando realmente existir o diretório `.weaver/` no diretório examinado. E no fim do loop, sempre vamos para o diretório-pai do qual estamos:

Seção: Inicialização (continuação):

```
while(strcmp(complete_path, "/.weaver")){ // Testa se chegamos ao fim
  if(directory_exist(complete_path) == 1){ // Testa se achamos o diretório
    inside_weaver_directory = true;
    complete_path[strlen(complete_path)-7] = '\0'; // Apaga o '.weaver'
    project_path = concatenate(complete_path, "");
    if(project_path == NULL){ free(complete_path); ERROR(); }
    break;
  }
  else{
    path_up(complete_path);
    strcat(complete_path, "/.weaver");
  }
}
free(complete_path);
```

Como alocamos memória para `project_path` armazenar o endereço do projeto atual se estamos em um projeto Weaver, no final do programa teremos que desalocar a memória:

Seção: Finalização:

```
if(project_path != NULL) free(project_path);
```

1.4.6.2- Inicializando variáveis `weaver_version_major` e `weaver_version_minor`

Para descobrirmos a versão atual do Weaver que temos, basta consultar o valor presente na macro `VERSION`. Então, obtemos o número de versão maior e menor que estão separados por um ponto (se existirem). Note que se não houver um ponto no nome da versão, então ela é uma versão de testes. Mesmo neste caso o código abaixo vai funcionar, pois a função `atoi` iria retornar 0 nas duas invocações por encontrar respectivamente uma string sem dígito algum e um fim de string sem conteúdo:

Seção: Inicialização (continuação):

```
{
  char *p = VERSION;
  while(*p != '.' && *p != '\0') p ++;
  if(*p == '.') p ++;
  weaver_version_major = atoi(VERSION);
  weaver_version_minor = atoi(p);
}
```

1.4.6.3- Inicializando variáveis `project_version_major` e `project_version_minor`

Se estamos dentro de um projeto Weaver, temos que inicializar informação sobre qual versão do Weaver foi usada para atualizá-lo pela última vez. Isso pode ser obtido lendo o arquivo *.weaver/version* localizado dentro do diretório Weaver. Se não estamos em um diretório Weaver, não precisamos inicializar tais valores. O número de versão maior e menor é separado por um ponto.

Seção: Inicialização (continuação):

```
if(inside_weaver_directory){
  FILE *fp;
  char *p, version[10];
  char *file_path = concatenate(project_path, ".weaver/version", "");
  if(file_path == NULL) ERROR();
  fp = fopen(file_path, "r");
  free(file_path);
  if(fp == NULL) ERROR();
  p = fgets(version, 10, fp);
  if(p == NULL){ fclose(fp); ERROR(); }
  while(*p != '.' && *p != '\0') p ++;
  if(*p == '.') p ++;
  project_version_major = atoi(version);
  project_version_minor = atoi(p);
  fclose(fp);
}
```

1.4.6.4- Inicializando `have_arg` e `argument`

Uma das variáveis mais fáceis e triviais de se inicializar. Basta consultar `argc` e `argv`.

Seção: Inicialização (continuação):

```
have_arg = (argc > 1);
if(have_arg) argument = argv[1];
```

1.4.6.5- Inicializando `arg_is_path`

Agora temos que verificar se no caso de termos um argumento, se ele é um caminho para um projeto Weaver existente ou não. Para isso, checamos se ao concatenarmos `/.weaver` no argumento encontramos o caminho de um diretório existente ou não.

Seção: Inicialização (continuação):

```
if(have_arg){
  char *buffer = concatenate(argument, "/.weaver", "");
  if(buffer == NULL) ERROR();
  if(directory_exist(buffer) == 1){
    arg_is_path = 1;
  }
  free(buffer);
}
```

1.4.6.6- Inicializando `shared_dir`

A variável `shared_dir` deverá conter onde estão os arquivos compartilhados da instalação de Weaver. Se existir a variável de ambiente `WEAVER_DIR`, este será o caminho. Caso contrário, assumiremos o valor padrão de `/usr/share/weaver`.

Seção: Inicialização (continuação):

```
{
  char *weaver_dir = getenv("WEAVER_DIR");
  if(weaver_dir == NULL){
    shared_dir = concatenate("/usr/share/weaver/", "");
    if(shared_dir == NULL) ERROR();
  }
  else{
    shared_dir = concatenate(weaver_dir, "");
    if(shared_dir == NULL) ERROR();
  }
}
```

E isso requer que tenhamos que no fim do programa desalocar a memória alocada para `shared_dir`:

Seção: Finalização (continuação):

```
if(shared_dir != NULL) free(shared_dir);
```

1.4.6.7- Inicializando `arg_is_valid_project`

A próxima questão que deve ser averiguada é se o que recebemos como argumento, caso haja argumento, pode ser o nome de um projeto Weaver válido ou não. Para isso, três condições precisam ser satisfeitas:

1) O nome base do projeto deve ser formado somente por caracteres alfanuméricos (embora uma barra possa aparecer para passar o caminho completo de um projeto).

2) Não pode existir um arquivo com o mesmo nome do projeto no local indicado para a criação.

3) O projeto não pode ter o nome de nenhum arquivo que costuma ficar no diretório base de um projeto Weaver (como "Makefile"). Do contrário, na hora da compilação comandos como "`gcc game.c -o Makefile`" poderiam ser executados e sobrescreveriam arquivos importantes.

Para isso, usamos o seguinte código:

Seção: Inicialização (continuação):

```
if(have_arg && !arg_is_path){
  char *buffer;
  char *base = basename(argument);
  int size = strlen(base);
  int i;
  // Checando caracteres inválidos no nome:
  for(i = 0; i < size; i ++){
    if(!isalnum(base[i])){
      goto NOT_VALID;
    }
  }
  // Checando se arquivo existe:
  if(directory_exist(argument) != 0){
    goto NOT_VALID;
  }
  // Checando se conflita com arquivos de compilação:
  buffer = concatenate(shared_dir, "project/", base, "");
  if(buffer == NULL) ERROR();
  if(directory_exist(buffer) != 0){
    free(buffer);
    goto NOT_VALID;
  }
  free(buffer);
  arg_is_valid_project = true;
}
NOT_VALID:
```

1.4.6.8- Inicializando `arg_is_valid_module`

Checar se o argumento que recebemos pode ser um nome válido para um módulo só faz sentido se estivermos dentro de um diretório Weaver e se um argumento estiver sendo passado. Neste caso, o argumento é um nome válido se ele contiver apenas caracteres alfanuméricos e se não existir no projeto um arquivo `.c` ou `.h` em `src/` que tenha o mesmo nome do argumento passado:

Seção: Inicialização (continuação):

```
if(have_arg && inside_weaver_directory){
  char *buffer;
  int i, size;
  size = strlen(argument);
  // Checando caracteres inválidos no nome:
```

```
  for(i = 0; i < size; i ++){
    if(!isalnum(argument[i])){
      goto NOT_VALID_MODULE;
    }
  }
  // Checando por conflito de nomes:
  buffer = concatenate(project_path, "src/", argument, ".c", "");
  if(buffer == NULL) ERROR();
  if(directory_exist(buffer) != 0){
    free(buffer);
    goto NOT_VALID_MODULE;
  }
  buffer[strlen(buffer) - 1] = 'h';
  if(directory_exist(buffer) != 0){
    free(buffer);
    goto NOT_VALID_MODULE;
  }
  free(buffer);
  arg_is_valid_module = true;
}
NOT_VALID_MODULE:
```

1.4.6.9- Inicializando `author_name`

A variável `author_name` deve conter o nome do usuário que está invocando o programa. Esta informação é útil para gerar uma mensagem de Copyright nos arquivos de código fonte de novos módulos.

Para obter o nome do usuário, começamos obtendo o seu UID. De posse dele, obtemos todas as informações de login com um `getpwuid`. Se o usuário tiver registrado um nome em `/etc/passwd`, obtemos tal nome na estrutura retornada pela função. Caso contrário, assumiremos o login como sendo o nome:

Seção: Inicialização (continuação):

```
{
  struct passwd *login;
  int size;
  char *string_to_copy;
  login = getpwuid(getuid()); // Obtém dados de usuário
  if(login == NULL) ERROR();
  size = strlen(login -> pw_gecos);
  if(size > 0)
    string_to_copy = login -> pw_gecos;
  else
    string_to_copy = login -> pw_name;
  size = strlen(string_to_copy);
  author_name = (char *) malloc(size + 1);
  if(author_name == NULL) ERROR();
  strcpy(author_name, string_to_copy);
}
```

Depois, precisaremos desalocar a memória ocupada por `author_name`:

Seção: Finalização (continuação):

```
if(author_name != NULL) free(author_name);
```

1.4.6.10- Inicializando `project_name`

Só faz sendido falarmos no nome do projeto se estivermos dentro de um projeto Weaver. Neste caso, o nome do projeto pode ser encontrado em um dos arquivos do diretório base de tal projeto em `.weaver/name`:

Seção: Inicialização (continuação):

```
if(inside_weaver_directory){
  FILE *fp;
  char *c, *filename = concatenate(project_path, ".weaver/name", "");
  if(filename == NULL) ERROR();
  project_name = (char *) malloc(256);
  if(project_name == NULL){
    free(filename);
    ERROR();
  }
  fp = fopen(filename, "r");
  if(fp == NULL){
    free(filename);
    ERROR();
  }
  c = fgets(project_name, 256, fp);
  fclose(fp);
  free(filename);
  if(c == NULL) ERROR();
  project_name[strlen(project_name)-1] = '\0';
  project_name = realloc(project_name, strlen(project_name) + 1);
  if(project_name == NULL) ERROR();
}
```

Depois, precisaremos desalocar a memória ocupada por `project_name`:

Seção: Finalização (continuação):

```
if(project_name != NULL) free(project_name);
```

1.4.6.11- Inicializando `year`

O ano atual é trivial de descobrir usando a função `localtime`:

Seção: Inicialização (continuação):

```
{
  time_t current_time;
  struct tm *date;

  time(&current_time);
  date = localtime(&current_time);
  year = date -> tm_year + 1900;
}
```

1.4.7- Caso de uso 1: Imprimir ajuda de criação de projeto

O primeiro caso de uso sempre ocorre quando Weaver é invocado fora de um diretório de projeto e a invocação é sem argumentos ou com argumento `--help`. Nesse caso assumimos que o usuário não sabe bem como usar o programa e imprimimos uma mensagem de ajuda. A mensagem de ajuda terá uma forma semelhante a esta:

```
   .  .   You are outside a Weaver Directory.
  ./  \.  The following command uses are available:
  \\  //
  \\()//  weaver
  .={}=.      Print this message and exits.
 / /‘’\ \
 ‘ \  / ’  weaver PROJECT_NAME
   ‘’        Creates a new Weaver Directory with a new
             project.
```

O que é feito com o código abaixo:

Seção: Caso de uso 1: Imprimir ajuda de criação de projeto:

```
if(!inside_weaver_directory && (!have_arg || !strcmp(argument, "--help"))){
 printf("    .  .    You are outside a Weaver Directory.\n"
 "   .|  |.   The following command uses are available:\n"
 "   ||  ||\n"
 "   \\\\()//  weaver\n"
 "   .={}=.     Print this message and exits.\n"
 "  / /‘’\\ \\\n"
 "  ‘ \\  / ’  weaver PROJECT_NAME\n"
 "    ‘’       Creates a new Weaver Directory with a new\n"
 "             project.\n");
 END();
}
```

1.4.8- Caso de uso 2: Imprimir ajuda de gerenciamento

O segundo caso de uso também é bastante simples. Ele é invocado quando já estamos dentro de um projeto Weaver e invocamos Weaver sem argumentos ou com um `--help`. Assumimos neste caso que o usuário quer instruções sobre a criação de um novo módulo. A mensagem que imprimiremos é semelhante à esta:

```
      \            You are inside a Weaver Directory.
       \______/     The following command uses are available:
       /\____/\
      / /\__/\ \      weaver
   __/_/_/\/\_\_\___     Prints this message and exits.
     \ \ \/\/ / /
      \ \/__\/ /       weaver NAME
       \/____\/          Creates NAME.c and NAME.h, updating
       /     \          the Makefile and headers
      /
```

O que é obtido com o código:

Seção: Caso de uso 2: Imprimir ajuda de gerenciamento:

```
if(inside_weaver_directory && (!have_arg || !strcmp(argument, "--help"))){
 printf("      \\              You are inside a Weaver Directory.\n"
 "       \\______/       The following command uses are available:\n"
```

```
  "       /\\____/\\\n"
  "      / /\\__/\\ \\      weaver\n"
  "    __/_/_/\\/\\\_\\_\\___    Prints this message and exits.\n"
  "     \\ \\ \\\/\\/ / /\n"
  "      \\ \\/__\\/ /      weaver NAME\n"
  "       \\/____\\/        Creates NAME.c and NAME.h, updating\n"
  "       /     \\        the Makefile and headers\n"
  "      /\n");
  END();
}
```

1.4.9- Caso de uso 3: Mostrar versão instalada de Weaver

Um caso de uso ainda mais simples. Ocorrerá toda vez que o usuário invocar Weaver com o argumento `--version`:

Seção: Caso de uso 3: Mostrar versão:

```
if(have_arg && !strcmp(argument, "--version")){
  printf("Weaver\t%s\n", VERSION);
  END();
}
```

1.4.10- Caso de Uso 4: Atualizar projetos Weaver já existentes

Este caso de uso ocorre quando o usuário passar como argumento para Weaver um caminho absoluto ou relativo para um diretório Weaver existente. Assumimos então que ele deseja atualizar o projeto passado como argumento. Talvez o projeto tenha sido feito com uma versão muito antiga do motor e ele deseja que ele passe a usar uma versão mais nova da API.

Naturalmente, isso só será feito caso a versão de Weaver instalada seja superior à versão do projeto ou se a versão de Weaver instalada for uma versão instável para testes. Entende-se neste caso que o usuário deseja testar a versão experimental de Weaver no projeto. Fora isso, não é possível fazer *downgrades* de projetos, passando da versão 0.2 para 0.1, por exemplo.

Versões experimentais sempre são identificadas como tendo um nome formado somente por caracteres alfabéticos. Versões estáveis serão sempre formadas por um ou mais dígitos, um ponto e um ou mais dígitos (o número de versão maior e menor). Como o número de versão é interpretado com um `atoi` , isso significa que se estamos usando uma versão experimental, então o número de versão maior e menor serão sempre identificados como zero.

Pela definição que fizemos até agora, isso significa também que projetos em versões experimentais de Weaver sempre serão atualizados, independente da versão ser mais antiga ou mais nova.

Uma atualização consiste em copiar todos os arquivos que estão no diretório de arquivos compartilhados Weaver dentro de `project/src/weaver` para o diretório `src/weaver` do projeto em questão.

Mas para copiarmos os arquivos precisamos primeiro de uma função capaz de copiar um único arquivo. A função `copy_single_file` tenta copiar o arquivo cujo caminho é o primeiro argumento para o diretório cujo caminho é o segundo argumento. Se ela conseguir, retorna 1 e retorna 0 caso contrário.

Seção: Funções auxiliares Weaver (continuação):

```
int copy_single_file(char *file, char *directory){
  int block_size, bytes_read;
  char *buffer, *file_dst;
  FILE *orig, *dst;
```

```
  // Inicializa 'block_size':
     <Seção a ser Inserida: Descobre tamanho do bloco do sistema de arquivos>
  buffer = (char *) malloc(block_size); // Aloca buffer de cópia
  if(buffer == NULL) return 0;
  file_dst = concatenate(directory, "/", basename(file), "");
  if(file_dst == NULL) return 0;
  orig = fopen(file, "r"); // Abre arquivo de origem
  if(orig == NULL){
    free(buffer);
    free(file_dst);
    return 0;
  }
  dst = fopen(file_dst, "w"); // Abre arquivo de destino
  if(dst == NULL){
    fclose(orig);
    free(buffer);
    free(file_dst);
    return 0;
  }
  while((bytes_read = fread(buffer, 1, block_size, orig)) > 0){
    fwrite(buffer, 1, bytes_read, dst); // Copia origem -> buffer -> destino
  }
  fclose(orig);
  fclose(dst);
  free(file_dst);
  free(buffer);
  return 1;
}
```

O mais eficiente é que o buffer usado para copiar arquivos tenha o mesmo tamanho do bloco do sistema de arquivos. Para obter o valor correto deste tamanho, usamos o seguinte trecho de código:

Seção: Descobre tamanho do bloco do sistema de arquivos:

```
{
  struct stat s;
  stat(directory, &s);
  block_size = s.st_blksize;
  if(block_size <= 0){
    block_size = 4096;
  }
}
```

De posse da função que copia um só arquivo, definimos uma função que copia todo o conteúdo de um diretório para outro diretório:

Seção: Funções auxiliares Weaver (continuação):

```
int copy_files(char *orig, char *dst){
  DIR *d = NULL;
  struct dirent *dir;
  d = opendir(orig);
  if(d){
    while((dir = readdir(d)) != NULL){ // Loop para ler cada arquivo
```

```
        char *file;
        file = concatenate(orig, "/", dir -> d_name, "");
        if(file == NULL){
          return 0;
        }
    #if (defined(__linux__) || defined(_BSD_SOURCE)) && defined(DT_DIR)
      // Se suportamos DT_DIR, não precisamos chamar a função 'stat':
      if(dir -> d_type == DT_DIR){
    #else
      struct stat s;
      int err;
      err = stat(file, &s);
      if(err == -1) return 0;
      if(S_ISDIR(s.st_mode)){
    #endif
        // Se concluirmos estar lidando com subdiretório via 'stat' ou 'DT_DIR':
        char *new_dst;
        new_dst = concatenate(dst, "/", dir -> d_name, "");
        if(new_dst == NULL){
          return 0;
        }
        if(strcmp(dir -> d_name, ".") && strcmp(dir -> d_name, "..")){
          if(!directory_exist(new_dst)) mkdir(new_dst, 0755);
          if(copy_files(file, new_dst) == 0){
            free(new_dst);
            free(file);
            closedir(d);
            return 0; // Não fazemos nada para diretórios '.' e '..'
          }
        }
        free(new_dst);
      }
      else{
        // Se concluimos estar diante de um arquivo usual:
        if(copy_single_file(file, dst) == 0){
          free(file);
          closedir(d);
          return 0;
        }
      }
    free(file);
   } // Fim do loop para ler cada arquivo
   closedir(d);
 }
 return 1;
}
```

A função acima presumiu que o diretório de destino tem a mesma estrutura de diretórios que a origem.

De posse de todas as funções podemos escrever o código do caso de uso em que iremos realizar a atualização:

Seção: Caso de uso 4: Atualizar projeto Weaver:

```
if(arg_is_path){
  if((weaver_version_major == 0 && weaver_version_minor == 0) ||
     (weaver_version_major > project_version_major) ||
     (weaver_version_major == project_version_major &&
      weaver_version_minor > project_version_minor)){
    char *buffer, *buffer2;
    // |buffer| passa a valer  SHARED_DIR/project/src/weaver
    buffer = concatenate(shared_dir, "project/src/weaver/", "");
    if(buffer == NULL) ERROR();
    // |buffer2| passa a valer PROJECT_DIR/src/weaver/
    buffer2 = concatenate(argument, "/src/weaver/", "");
    if(buffer2 == NULL){
      free(buffer);
      ERROR();
    }
    if(copy_files(buffer, buffer2) == 0){
      free(buffer);
      free(buffer2);
      ERROR();
    }
    free(buffer);
    free(buffer2);
  }
  END();
}
```

1.4.11- Caso de Uso 5: Adicionando um módulo ao projeto Weaver

Se estamos dentro de um diretório de projeto Weaver, e o programa recebeu um argumento, então estamos inserindo um novo módulo no nosso jogo. Se o argumento é um nome válido, podemos fazer isso. Caso contrário,devemos imprimir uma mensagem de erro e sair.

Criar um módulo basicamente envolve:

a) Criar arquivos `.c` e `.h` base, deixando seus nomes iguais ao nome do módulo criado.

b) Adicionar em ambos um código com copyright e licenciamento com o nome do autor, do projeto e ano.

c) Adicionar no `.h` código de macro simples para evitar que o cabeçalho seja inserido mais de uma vez e fazer com que o `.c` inclua o `.h` dentro de si.

d) Fazer com que o `.h` gerado seja inserido em `src/includes.h` e assim suas estruturas sejam acessíveis de todos os outros módulos do jogo.

A parte de imprimir um código de copyright será feita usando a nova função abaixo:

Seção: Funções auxiliares Weaver (continuação):

```
void write_copyright(FILE *fp, char *author_name, char *project_name, int year){
  char license[] = "/*\nCopyright (c) %s, %d\n\nThis file is part of %s.\n\n%s\
is free software: you can redistribute it and/or modify\nit under the terms of\
the GNU General Public License as published by\nthe Free Software Foundation,\
either version 3 of the License, or\n(at your option) any later version.\
\n\n%s is distributed in the hope that it will be useful,\nbut WITHOUT ANY\
```

```
 WARRANTY; without even the implied warranty of\nMERCHANTABILITY or FITNESS\
 FOR A PARTICULAR PURPOSE.  See the\nGNU General Public License for more\
 details.\n\nYou should have received a copy of the GNU General Public License\
\nalong with %s. If not, see <http://www.gnu.org/licenses/>.*/\n\n";
 fprintf(fp, license, author_name, year, project_name, project_name,
       project_name, project_name);
}
```

Já o código de criação de novo módulo passa a ser:

Seção: Caso de uso 5: Criar novo módulo:

```
if(inside_weaver_directory && have_arg){
  if(arg_is_valid_module){
    char *filename;
    FILE *fp;
    // Criando modulo.c
    filename = concatenate(project_path, "src/", argument, ".c", "");
    if(filename == NULL) ERROR();
    fp = fopen(filename, "w");
    if(fp == NULL){
      free(filename);
      ERROR();
    }
    write_copyright(fp, author_name, project_name, year);
    fprintf(fp, "#include \"%s.h\"", argument);
    fclose(fp);
    filename[strlen(filename)-1] = 'h'; // Criando modulo.h
    fp = fopen(filename, "w");
    if(fp == NULL){
      free(filename);
      ERROR();
    }
    write_copyright(fp, author_name, project_name, year);
    fprintf(fp, "#ifndef _%s_h_\n", argument);
    fprintf(fp, "#define _%s_h_\n\n\n#endif", argument);
    fclose(fp);
    free(filename);

    // Atualizando src/includes.h para inserir modulo.h:
    fp = fopen("src/includes.h", "a");
    fprintf(fp, "#include \"%s.h\"\n", argument);
    fclose(fp);
  }
  else{
    fprintf(stderr, "ERROR: This module name is invalid.\n");
    return_value = 1;
  }
  END();
}
```

1.4.12- Caso de Uso 6: Criando um novo projeto Weaver

Criar um novo projeto Weaver consiste em criar um novo diretório com o nome do projeto, copiar para lá tudo o que está no diretório `project` do diretório de arquivos compartilhados e criar um diretório `.weaver` com os dados do projeto. Além disso, criamos um `src/game.c` e `src/game.h` adicionando o comentário de Copyright neles e copiando a estrutura básica dos arquivos do diretório compartilhado `basefile.c` e `basefile.h`. Também criamos um `src/includes.h` que por hora estará vazio, mas será modificado na criação de futuros módulos.

A permissão dos diretórios criados será `drwxr-xr-x` (`0755` em octal).

Seção: Caso de uso 6: Criar novo projeto:

```
if(! inside_weaver_directory && have_arg){
 if(arg_is_valid_project){
   int err;
   char *dir_name;
   FILE *fp;

   err = mkdir(argument, S_IRWXU | S_IRWXG | S_IROTH);
   if(err == -1) ERROR();
   err = chdir(argument);
   if(err == -1) ERROR();
   mkdir(".weaver", 0755); mkdir("conf", 0755);
   mkdir("src", 0755); mkdir("src/weaver", 0755);
   mkdir("image", 0755);  mkdir("sound", 0755);
   mkdir("music", 0755);

   dir_name = concatenate(shared_dir, "project", "");
   if(dir_name == NULL) ERROR();
   if(copy_files(dir_name, ".") == 0){
     free(dir_name);
     ERROR();
   }
   free(dir_name); //Criando arquivo com número de versão:
   fp = fopen(".weaver/version", "w");
   fprintf(fp, "%s\n", VERSION);
   fclose(fp); // Criando arquivo com nome de projeto:
   fp = fopen(".weaver/name", "w");
   fprintf(fp, "%s\n", basename(argv[1]));
   fclose(fp);

   fp = fopen("src/game.c", "w");
   if(fp == NULL) ERROR();
   write_copyright(fp, author_name, argument, year);
   if(append_basefile(fp, shared_dir, "basefile.c") == 0) ERROR();
   fclose(fp);

   fp = fopen("src/game.h", "w");
   if(fp == NULL) ERROR();
   write_copyright(fp, author_name, argument, year);
   if(append_basefile(fp, shared_dir, "basefile.h") == 0) ERROR();
   fclose(fp);

   fp = fopen("src/includes.h", "w");
   write_copyright(fp, author_name, argument, year);
```

```
    fclose(fp);
  }
  else{
    fprintf(stderr, "ERROR: %s is not a valid project name.", argument);
    return_value = 1;
  }
  END();
}
```

A única coisa ainda não-definida é a função usada acima `append_basefile`. Esta é uma função bastante específica para concatenar o conteúdo de um arquivo para o outro dentro deste trecho de código. Não é uma função geral, pois ela recebe como argumento um ponteiro para o arquivo de destino aberto e reebe como argumento o diretório em que está a origem e o nome do arquivo de origem ao invés de ter a forma mais intuitiva `cat(origem, destino)`.

Definimos abaixo a forma da `append_basefile`:

Seção: Funções auxiliares Weaver (continuação):

```
int append_basefile(FILE *fp, char *dir, char *file){
  int block_size, bytes_read;
  char *buffer, *directory = ".";
  char *path = concatenate(dir, file, "");
  if(path == NULL) return 0;
  FILE *origin;
```

<**Seção a ser Inserida: Descobre tamanho do bloco do sistema de arquivos**>

```
  buffer = (char *) malloc(block_size);
  if(buffer == NULL){
    free(path);
    return 0;
  }
  origin = fopen(path, "r");
  if(origin == NULL){
    free(buffer);
    free(path);
    return 0;
  }
  while((bytes_read = fread(buffer, 1, block_size, origin)) > 0){
    fwrite(buffer, 1, bytes_read, fp);
  }

  fclose(origin);
  free(buffer);
  free(path);

  return 1;
}
```

1.5 - O arquivo `conf`

h

Em toda árvore de diretórios de um projeto Weaver, deve existir um arquivo chamado `conf/conf.h`. Este arquivo é um arquivo de cabeçalho C que será incluído em todos os outros arquivos de código do Weaver no projeto e que permitirá que o comportamento da Engine seja modificado naquele projeto específico.

O arquivo deverá ter as seguintes macros (dentre outras):

`W_DEBUG_LEVEL` : Indica o que deve ser impresso na saída padrão durante a execução. Seu valor pode ser:

`0` : Nenhuma mensagem de depuração é impressa durante a execução do programa. Ideal para compilar a versão final de seu jogo. `1` : Mensagens de aviso que provavelmente indicam erros são impressas durante a execução. Por exemplo, um vazamento de memória foi detectado, um arquivo de textura não foi encontrado, etc. `2` : Mensagens que talvez possam indicar erros ou problemas, mas que talvez sejam inofensivas são impressas. `3` : Mensagens informativas com dados sobre a execução, mas que não representam problemas são impressas.

`W_SOURCE` : Indica a linguagem que usaremos em nosso projeto. As opções são:

`W_C` : Nosso projeto é um programa em C. `W_CPP` : Nosso projeto é um programa em C++.

`W_TARGET` : Indica que tipo de formato deve ter o jogo de saída. As opções são:

`W_ELF` : O jogo deverá rodar nativamente em Linux. Após a compilação, deverá ser criado um arquivo executável que poderá ser instalado com `make install`. `W_WEB` : O jogo deverá executar em um navegador de Internet. Após a compilação deverá ser criado um diretório chamado `web` que conterá o jogo na forma de uma página HTML com Javascript. Não faz sentido instalar um jogo assim. Ele deverá ser copiado para algum servidor Web para que possa ser jogado na Internet. Isso é feito usando Emscripten.

Opcionalmente as seguintes macros podem ser definidas também (dentre outras):

`W_MULTITHREAD` : Se a macro for definida, Weaver é compilado com suporte à múltiplas threads acionadas pelo usuário. Note que de qualquer forma vai existir mais de uma thread rodando no programa para que música e efeitos sonoros sejam tocados. Mas esta macro garante que mutexes e código adicional sejam executados para que o desenvolvedor possa executar qualquer função da API concorrentemente.

Ao longo das demais seções deste documento, outras macros que devem estar presentes ou que são opcionais serão apresentadas. Mudar os seus valores, adicionar ou removê-las é a forma de configurar o funcionamento do Weaver.

Junto ao código-fonte de Weaver deve vir também um arquivo `conf/conf.h` que apresenta todas as macros possíveis em um só lugar. Apesar de ser formado por código C, tal arquivo não será apresentado neste PDF, pois é importante que ele tenha comentários e CWEB iria remover os comentários ao gerar o código C.

O modo pelo qual este arquivo é inserido em todos os outros cabeçalhos de arquivos da API Weaver é:

Seção: Inclui Cabeçalho de Configuração:

```
#include "conf_begin.h"
#include "../../conf/conf.h"
```

Note que haverão também cabeçalhos `conf_begin.h` que cuidarão de toda declaração de inicialização que forem necessárias. Para começar, criaremos o `conf_begin.h` para inicializar as macros `W_WEB` e `W_ELF` :

Arquivo: `project/src/weaver/conf_begin.h`**:**

```
#define W_ELF 0
#define W_WEB 1
```

1.6 - Funções básicas Weaver

Vamos criar também um `weaver.h` que irá incluir automaticamente todos os cabeçalhos Weaver necessários (inclusivve este):

Arquivo: `project/src/weaver/weaver.h`**:**

```
#ifndef _weaver_h_
#define _weaver_h_
#ifdef __cplusplus
  extern "C" {
#endif
```

<Seção a ser Inserida: Inclui Cabeçalho de Configuração>

<Seção a ser Inserida: Cabeçalhos Weaver>

```
#ifdef __cplusplus
  }
#endif
#endif
```

Neste cabeçalho, iremos também declarar três funções.

A primeira função servirá para inicializar a API Weaver. Seus parâmetros devem ser o nome do arquivo em que ela é invocada e o número de linha. Esta informação será útil para imprimir mensagens de erro úteis em caso de erro.

A segunda função deve ser a última coisa invocada no programa. Ela encerra a API Weaver.

E a terceira função deve ser chamada no loop principal do programa e será responsável pro fazer coisas como desenhar na tela, ficar um tempo ociosa para não consumir 100% da CPU e coisas assim. O seu argumento representa quantos milissegundos devemos ficar nela sem fazer nada para evitar consumo de todo o tempo de CPU.

Nenhuma destas funções foi feita para ser chamada por mais de uma thread. Todas elas só devem ser usadas pela thread principal. Mesmo que você defina a macro `W_MULTITHREAD`, todas as outras funções serão seguras para threads, menos estas três.

Como não é razoável pedir para que um programador se preocupe com detalhes como o arquivo e linha de execução da função, abaixo das três funções definiremos funções de macro que tornarão tais informações transparentes e de responsabilidade do compilador. As três funções de macro são como as três funções serão executadas na prática.

Seção: Cabeçalhos Weaver (continuação):

```
void _awake_the_weaver(char *filename, unsigned long line);
void _may_the_weaver_sleep();
void _weaver_rest(unsigned long time);

#define Winit() _awake_the_weaver(__FILE__, __LINE__)
#define Wexit() _may_the_weaver_sleep()
#define Wrest(a) _weaver_rest(a)
```

Definiremos melhor a responsabilidade destas funções ao longo dos demais capítulos. Mas colocaremos aqui a definição delas no arquivo adequado. E no caso da função `_weaver_rest`, colocaremos aqui algum código mínimo que faz todos os buffers usados para desenho OpenGL devem ser limpos em cada frame de jogo (`glClear`) e que se nosso jogo é um programa executável Linux, então precisamos usar `nanosleep` para liberar a CPU um pouco (caso o jogo seja compilado para Javascript, isso ocorre usando um mecanismo diferente, então não é necessário especificar isso todo frame). Além disso, caso o jogo seja um programa nativo, nós usamos *double buffering*, e por isso precisamos do `glXSwapBuffers` ao invés de um mais simples `glFlush`.

Arquivo: `project/src/weaver/weaver.c`**:**

```
#include "weaver.h"

//API Weaver: Definições

void _awake_the_weaver(char *filename, unsigned long line){
  //API Weaver: Inicialização
}
```

```
void _may_the_weaver_sleep(void){
  //API Weaver: Finalização
  exit(0);
}

void _weaver_rest(unsigned long time){
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
#if W_TARGET == W_ELF
  struct timespec req = {0, time * 1000000};
#endif
```

<Seção a ser Inserida: **API Weaver: Loop Principal**>

```
#if W_TARGET == W_ELF
  glXSwapBuffers(_dpy, _window);
#else
  glFlush();
#endif
#if W_TARGET == W_ELF
  nanosleep(&req, NULL);
#endif
}
```

Seção: API Weaver: Loop Principal:

```
  // A definir...
```