Weaver: Uma Engine de Desenvolvimento de Jogos

Thiago “Harry” Leucz Astrizi

31 de maio de 2016

Sumário

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 conjunto de programas denominados de CWEB, a qual foi desenvolvida por Donald Knuth e Silvio Levy. Um programa chamado CWEAVE é responsável por gerar por meio do código-fonte do programa um código LaTeX, o qual é compilado para um formato DVI, e finalmente para este formato PDF final. 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. Qualqer coisa que vá de encontro à eles devem ser tratados como *bugs*.

- **Software é conhecimento sobre como realizar algo escrito em linguagens formais de computadores. E 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.

- **Como corolário da filosofia anterior, Weaver deve estar bem-documentado.**

A escolha de CWEB para a criação de Weaver é um reflexo desta filosofia. Naturalmente, outra questão que surge é: documentado para quem?

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. O motivo de eu escrever em português neste manual é este. Mas ao mesmo tempo quero que meu programa seja acessível ao mundo todo. Isso me leva a escrever o site do projeto em inglês e dar prioridade para a escrita de tutoriais em inglês lá. Os nomes das variáveis também serão em inglês. Com isso tento conciliar as duas coisas, por mais difícil que isso seja.

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

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. Que Weaver nunca tenha funções abomináveis como a API do Windows, com 10 ou mais argumentos.

Como reflexo disso, faremos com que em todo projeto Weaver haja 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.

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

Computadores são coisas complexas primariamente porque pessoas tentam resolver neles problemas insolúveis. É como tapar o sol com a peneira. Você até 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 de computador modernos.

Como exemplo de tais tentativas de solucionar problemas insolúveis, que fazem com que arquiteturas monstruosas e ineficientes sejam construídas 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.

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.

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

Isso está por trás da decisão do código da API Weaver ser inserido estaticamente nos projetos ao invés de compilado como bibliotecas compartilhadas. À rigor, este é um exemplo de problema insolúvel mencionado anteriormente. Por causa disso, o acordo proposto é que Weaver garanta que jogos possam ser instalados por meio de pacotes nas 2 distribuições Linux mais populares segundo o Distro Watch, sem a instalação de pacotes adicionais, desde que as distribuições em si suportem interfaces gráficas. Da parte do usuário, ele deverá usar uma destas distribuições ou deverá concordar em instalar por conta própria as dependências antes de instalar um jogo Weaver.

Naturalmente, caso alguma dependência esteja faltando, a mensagem de erro na instalação deve ser tão clara como em qualquer outro pacote.

Ferramentas para gerar instaladores nas 2 distribuições mais usadas devem ser fornecidas, desde que elas possuam gerenciador de pacotes.

- **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 transparentes e nem precisar de uma função específica após os objetos que compõe o nosso ambiente 3D sejam 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`.

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

- **ctangle:** Extrai o código C dos arquivos de `cweb/`.
- **clang:** Um compilador C que gera executáveis à partir de código C. Pode-se usar o GCC (abaixo) ao invés dele.
- **gcc:** Um compilador que gera executáveis à partir de código C. Pode-se usar o CLANG (acima) ao invés dele.
- **make:** Interpreta e executa comandos do Makefile.

Adicionalmente, os seguintes programs são necessários para se gerar a documentação:

- **cweave:** Usado para gerar código LaTeX usado para gerar este PDF.
- **dvipdf:** Usado para converter um arquivo `.dvi` em um `.pdf`, que é o formato final deste manual. Dependendo da sua distribuição este programa vem em algum pacote com o nome `texlive`.
- **graphviz:** Um conjunto de programas usado para gerar representações gráficas em diferentes formatos de estruturas como grafos.
- **latex:** Usado para converter um arquivo `.tex` em um `.dvi`. Também costuma vir em pacotes chamados `texlive`.

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

- **emscripten:** Isso é opcional. Mas é necessário caso você queira usar Weaver para construir jogos que possam ser jogados em navegadores Web após serem compilados para Javascript.
- **opengl:** Você precisa de arquivos de desenvolvimento da biblioteca gráfica OpenGL caso queira gerar programas executáveis para Linux.
- **xlib:** Você precisa dos arquivos de desenvolvimento Xlib caso você queira usar Weaver para gerar programas executáveis para Linux.
- **xxd:** Um programa capaz de gerar representação hexadecimal de arquivos quaisquer. É necessário para inserir o código dos shaders no programa. Por motivos obscuros, algumas distribuições trazem este programa no mesmo pacote do **vim**.

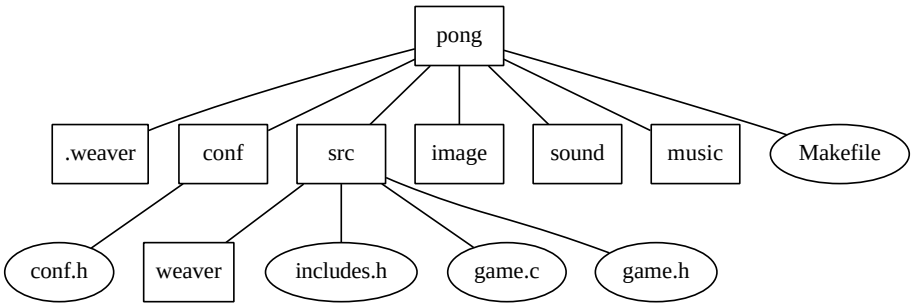
1.4 O programa `weaver`

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 pode ser encontrado junto ao código da engine em si. O programa é usado simplesmente para criar um novo projeto Weaver. E um projeto é um diretório com vários arquivos e diretórios necessários para gerar um novo jogo Weaver. 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 ue 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:

8 ⟨ src/weaver.c 8 ⟩ ≡

```
⟨ Cabeçalhos Incluídos no Programa Weaver 10 ⟩
⟨ Macros do Programa Weaver 9 ⟩
⟨ Funções auxiliares Weaver 27 ⟩
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;
      /* Usa-se o inteiro mais simples para tais valores. O padrão garante
       que o podemos representar até o número 65536 aqui. Provavelmente
       será o suficiente para toda a história deste projeto. */
  char *argument = Λ, *project_path = Λ, *shared_dir = Λ,
       *author_name = Λ, *project_name = Λ;      /* Strings UTF-8 */
  ⟨ Inicialização 12 ⟩
  ⟨ Caso de uso 1: Imprimir ajuda de criação de projeto 30 ⟩
  ⟨ Caso de uso 2: Imprimir ajuda de gerenciamento de projeto 31 ⟩
  ⟨ Caso de uso 3: Mostrar versão 32 ⟩
  ⟨ Caso de uso 4: Atualizar projeto Weaver 33 ⟩
  ⟨ Caso de uso 5: Criar novo módulo 38 ⟩
  ⟨ Caso de uso 6: Criar novo projeto 40 ⟩
finalize: ⟨ Finalização 13 ⟩
  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 *finalize* 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.

9 ⟨ Macros do Programa Weaver 9 ⟩ ≡
```
#define VERSION  "Alpha"
#define ERROR( )
  {
    perror(Λ);
    return_value = 1;
    goto finalize;
  }
#define END( ) gotofinalize;
```
This code is used in chunk 8.

1.4.5 Cabeçalhos do Programa Weaver

10 ⟨ Cabeçalhos Incluídos no Programa Weaver 10 ⟩ ≡
```
#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 */
```
This code is used in chunk 8.

1.4.6 Inicialização e Finalização do Programa Weaver

Inicializando *inside_weaver_directory* e *project_path*

Inicializar Weaver significa inicializar as 14 variáveis que serão usadas para definir o seu comportamento. A primeira delas 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 pai do `.weaver`. Se não o encontrarmos, estaremos fora de um diretório Weaver e não precisamos mudar nenhum valor das duas variáveis.

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

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

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

3. **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.

4. **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. 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 checar se o diretório `.weaver` existe, vamos assumir a existência de uma função chamada *directry_exists*(x), onde x é uma string e tal função deve retornar 1 se x for um diretório existente, -1 se x for um arquivo existente e 0 caso contrário. Para checarmos os diretórios acima dos atuais, assumiremos que existe uma função chamada *path_up*(x) que dada uma string x, apaga todos os caracteres dela de trás pra frente até remover dois "/". Desta forma, removemos em cada execução desta função o "/.weaver" presente no fim de cada string e também subimos um diretório na hierarquia na variável *complete_path*. O comportamento da função é indefinido se não existirem dois "/" na string, mas cuidaremos para que isto nunca aconteça no teste de finalização do loop.

Por fim, a tradução para a linguagem C da implementação que propomos. Vamos assumir que existe uma função *concatenate*, que recebe como argumento um número qualquer de strings como argumento, sendo que a última delas deve ser vazia. A função retorna uma nova string que é a concatenação delas, ou Λ se não é possível alocar espaço para isso.

12 ⟨ Inicialização 12 ⟩ ≡

```
char *path = Λ, *complete_path = Λ;
path = getcwd(Λ, 0);
if (path ≡ Λ) ERROR( );
complete_path = concatenate(path, "/.weaver", "");
if (complete_path ≡ Λ) {
  free(path);
  ERROR( );
}
free(path);      /* O while abaixo testa a Finalização 1: */
```

```
while (strcmp(complete_path, "/.weaver")) {
      /* O if abaixo testa a Finalização 2: */
  if (directory_exist(complete_path) ≡ 1) {
    inside_weaver_directory = true;
    complete_path[strlen(complete_path) − 7] = '\0';
       /* Apaga o .weaver */
    project_path = concatenate(complete_path, "");
    if (project_path ≡ Λ) {
      free(complete_path);
      ERROR( );
    }
    break;
  }
  else {      /* Dentro deste else está a manutenção do loop */
    path_up(complete_path);
    strcat(complete_path, "/.weaver");
  }
}
free(complete_path);
```

See also chunks 14, 15, 16, 17, 18, 20, 21, 22, 24, and 26.

This code is used in chunk 8.

¶ Isso significa que agora na finalização do projeto, temos que desalocar a memória de *path*:

13 ⟨ Finalização 13 ⟩ ≡

```
if (project_path ≠ Λ) free(project_path);
```

See also chunks 19, 23, and 25.

This code is used in chunk 8.

Inicializando *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. De qualquer forma 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:

14 ⟨ Inicialização 12 ⟩ +≡

```
{
  char *p = VERSION;
```

```
  while (*p ≠ '.' ∧ *p ≠ '\0') p++;
  if (*p ≡ '.') p++;
  weaver_version_major = atoi(VERSION);
  weaver_version_minor = atoi(p);
}
```

Inicializando *project_version_major* e *project_version_minor*

Se estamos dentro de um projeto Weaver, queremos saber qual foi a versão do Weaver usada para criar o projeto, ou então 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. Tal como em "0.5".

15 ⟨ Inicialização 12 ⟩ +≡

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

Inicializando *have_arg* e *argument*

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

16 ⟨ Inicialização 12 ⟩ +≡

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

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.

```
17 ⟨ Inicialização 12 ⟩ +≡
  if (have_arg) {
    char *buffer = concatenate(argument, "/.weaver", "");
    if (buffer ≡ Λ) ERROR( );
    if (directory_exist(buffer) ≡ 1) {
      arg_is_path = 1;
    }
    free(buffer);
  }
```

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`.

```
18 ⟨ Inicialização 12 ⟩ +≡
  {
    char *weaver_dir = getenv("WEAVER_DIR");
    if (weaver_dir ≡ Λ) {
      shared_dir = concatenate("/usr/share/weaver/", "");
      if (shared_dir ≡ Λ) ERROR( );
    }
    else {
      shared_dir = concatenate(weaver_dir, "");
      if (shared_dir ≡ Λ) ERROR( );
    }
  }
```

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

```
19 ⟨ Finalização 13 ⟩ +≡
  if (shared_dir ≠ Λ) free(shared_dir);
```

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:

```
20 ⟨ Inicialização 12 ⟩ +≡
  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 ≡ Λ) ERROR( );
    if (directory_exist(buffer) ≠ 0) {
      free(buffer);
      goto not_valid;
    }
    free(buffer);
    arg_is_valid_project = 1;
  }
  not_valid:
```

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:

```
21 ⟨ Inicialização 12 ⟩ +≡
  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 ≡ Λ) 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 = 1;
  }
  not_valid_module:
```

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, os quais serão criados e escritos pelo usuário da Engine.

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:

```
22 ⟨ Inicialização 12 ⟩ +≡
  {
    struct passwd *login;
    int size;
    char *string_to_copy;
    login = getpwuid(getuid( ));      /* Obtém dados de usuário */
    if (login ≡ Λ) 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 ≡ Λ) ERROR( );
  strcpy(author_name, string_to_copy);
}
```

¶ Depois, precisaremos desalocar a memória ocupada por *author_name*:

23 ⟨ Finalização 13 ⟩ +≡
```
if (author_name ≠ Λ) free(author_name);
```

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`:

24 ⟨ Inicialização 12 ⟩ +≡
```
if (inside_weaver_directory) {
  FILE *fp;
  char *filename = concatenate(project_path, ".weaver/name", "");
  if (filename ≡ Λ) ERROR( );
  project_name = (char *) malloc(256);
  if (project_name ≡ Λ) {
    free(filename);
    ERROR( );
  }
  fp = fopen(filename, "r");
  if (fp ≡ Λ) {
    free(filename);
    ERROR( );
  }
  fgets(project_name, 256, fp);
  fclose(fp);
  free(filename);
  project_name[strlen(project_name) − 1] = '\0';
  project_name = realloc(project_name, strlen(project_name) + 1);
  if (project_name ≡ Λ) ERROR( );
}
```

¶ Depois, precisaremos desalocar a memória ocupada por *project_name*:

25 ⟨ Finalização 13 ⟩ +≡
```
if (project_name ≠ Λ) free(project_name);
```

Inicializando *year*

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

26 ⟨ Inicialização 12 ⟩ +≡
```
{
  time_t current_time;
  struct tm *date;
  time(&current_time);
  date = localtime(&current_time);
  year = date→tm_year + 1900;
}
```

Função auxiliar: Checando se diretório ou arquivo existe

Definiremos agora a função *directory_exist* para verificarmos se um caminho de diretório passado como argumento existe ou não. Os valores de retorno possíveis desta função serão:

-1 : Arquivo existe, mas não é um diretório.

0 : Diretório ou arquivo não existe.

1 : Arquivo existe e é um diretório.

27 ⟨ Funções auxiliares Weaver 27 ⟩ ≡
```
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) {
    if (S_ISDIR(s.st_mode)) {
      return 1;
    }
    else {
      return −1;
    }
  }
  return 0;
}
```
See also chunks 28, 29, 34, 36, 39, and 41.

This code is used in chunk 8.

Função auxiliar: Apagando caracteres até apagar duas barras

Esta função é usada mais para manipular o caminho para arquivos no sistema de arquivos. Apagar a primeira barra é ficar só com o endereço do diretório, não do arquivo indicado pelo caminho. Apaga a segunda barra significa subir um nível na árvore de diretórios:

28 ⟨ Funções auxiliares Weaver 27 ⟩ +≡

```
void path_up(char *path)
{
  int erased = 0;
  char *p = path;
  while (*p ≠ '\0') p++;
  while (erased < 2 ∧ p ≠ path) {
    p--;
    if (*p ≡ '/') erased++;
    *p = '\0';
  }
}
```

Função auxiliar: Concatenando strings

Esta é uma das funções auxiliares mais usadas. El recebe um número variável de argumentos, todos strings sendo que o último é a string vazia. Então, ela aloca espaço para uma nova string e retorna um ponteiro para ela, sendo que a nova string é a concatenação de todos os argumentos. Se algo falhar, Λ é retornado:

29 ⟨ Funções auxiliares Weaver 27 ⟩ +≡

```
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 ≡ Λ) return Λ;
  strcpy(new_string, string);
  while (current_string[0] ≠ '\0') {
    current_string = va_arg(arguments, char *);
    current_size += strlen(current_string);
    realloc_return = (char *) realloc(new_string, current_size);
    if (realloc_return ≡ Λ) {
      free(new_string);
      return Λ;
    }
```

```
    new_string = realloc_return;
    strcat(new_string, current_string);
  }
  return new_string;
}
```

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.
```

30 ⟨ Caso de uso 1: Imprimir ajuda de criação de projeto 30 ⟩ ≡

```
if (¬inside_weaver_directory ∧ (¬have_arg ∨ ¬strcmp(argument, "--help")))
  {
  printf("␣␣␣␣.␣␣.␣␣␣␣␣You␣are␣outside␣a␣Weaver␣D\
      irectory.\n""␣␣␣.|␣␣|.␣␣␣␣The␣following␣command␣uses\
      ␣are␣available:\n""␣␣␣||␣␣||\n""␣␣␣\\\\()//␣␣weaver\
      \n""␣␣␣.={}=.␣␣␣␣␣␣Print␣this␣message␣and␣e\
      xits.\n""␣␣/␣/‘’\\␣\\\n""␣␣‘␣\\␣␣/␣’␣␣weaver\
      ␣PROJECT_NAME\n""␣␣␣␣␣‘’␣␣␣␣␣␣␣␣Creates␣a␣new␣Weaver␣Dir\
      ectory␣with␣a␣new\n""␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣project.\n");
  END();
}
```

This code is used in chunk 8.

1.4.8 Caso de uso 2: Imprimir ajuda de gerenciamento de projeto

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
.       /
```

31 ⟨ Caso de uso 2: Imprimir ajuda de gerenciamento de projeto 31 ⟩ ≡

```
if (inside_weaver_directory ∧ (¬have_arg ∨ ¬strcmp(argument, "--help"))) {
  printf("␣␣␣␣␣␣␣\\␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣You␣are␣inside\
      ␣a␣Weaver␣Directory.\n""␣␣␣␣␣␣␣\\______/␣␣\
      ␣␣␣␣␣␣The␣following␣command␣uses␣are␣ava\
      ilable:\n""␣␣␣␣␣␣␣/\\____/\\\n""␣␣␣␣␣␣␣/␣/\\__/\\␣\\
      \␣␣␣␣␣␣␣weaver\n""␣␣␣␣__/_/_/\\/\\_\\_\\___␣␣␣␣␣Prints␣th\
      is␣message␣and␣exits.\n""␣␣␣␣␣␣\\␣\\␣\\/\\/␣\
      /␣/\n""␣␣␣␣␣␣␣\\␣\\/__\\/␣/␣␣␣␣␣␣weaver␣NAME\\
      n""␣␣␣␣␣␣␣\\/____\\/␣␣␣␣␣␣␣␣␣␣Creates␣NAM\
      E.c␣and␣NAME.h,␣updating\n""␣␣␣␣␣␣␣/␣␣␣␣␣␣\\␣␣\
      ␣␣␣␣␣␣␣the␣Makefile␣and␣headers\n""␣␣␣␣␣␣␣/\n");
  END();
}
```

This code is used in chunk 8.

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`:

32 ⟨ Caso de uso 3: Mostrar versão 32 ⟩ ≡

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

This code is used in chunk 8.

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. Afinal, 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.

Assumindo que exista uma função $copy\_files(a, b)$ que copia todos os arquivos de a para b, definimos este caso de uso como:

33 ⟨ Caso de uso 4: Atualizar projeto Weaver 33 ⟩ ≡

```
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 ≡ Λ) ERROR();
        /* buffer2 passa a valer PROJECT_DIR/src/weaver/ */
    buffer2 = concatenate(argument, "/src/weaver/", "");
    if (buffer2 ≡ Λ) {
      free(buffer);
      ERROR();
    }
    if (copy_files(buffer, buffer2) ≡ 0) {
      free(buffer);
      free(buffer2);
      ERROR();
    }
    free(buffer);
    free(buffer2);
  }
  END();
```

```
}
```

This code is used in chunk 8.

¶ Resta então definirmos a função *copy_files* que usaremos para copiar arquivos. Mas antes dela iremos definir uma função usada para copiar um único arquivo, a qual chamaremos de *copy_single_file*:

34 ⟨ Funções auxiliares Weaver 27 ⟩ +≡

```
int copy_single_file(char *file, char *directory)
{
  int block_size;
  char *buffer;
  char *file_dst;
  FILE *orig, *dst;
  int bytes_read;
  ⟨ Descobre tamanho do bloco do sistema de arquivos 35 ⟩
    /* Nesta parte, block_size já foi inicializado com o tamanho do bloco
     do sistema de arquivos. Isso tornará a cópia seguinte mais eficiente. */
  buffer = (char *) malloc(block_size);
  if (buffer ≡ Λ) return 0;
  file_dst = concatenate(directory, "/", basename(file), "");
  if (file_dst ≡ Λ) return 0;
  orig = fopen(file, "r");
  if (orig ≡ Λ) {
    free(buffer);
    free(file_dst);
    return 0;
  }
  dst = fopen(file_dst, "w");
  if (dst ≡ Λ) {
    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);
  }
  fclose(orig);
  fclose(dst);
  free(file_dst);
  free(buffer);
  return 1;
}
```

¶ Para finalizar a função de cópia, basta descobrirmos agora como obter o valor do tamanho do bloco do sistema de arquivos usado. Para isso, usamos novamente a função *stat* em qualquer arquivo ou diretório do sistema de arquivos do destino. Isso funcionará em qualquer sistema POSIX. No código abaixo, tomamos também o cuidado de preencher um valor padrão para o caso de algo ter dado errado.

35 ⟨ Descobre tamanho do bloco do sistema de arquivos 35 ⟩ ≡

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

This code is used in chunks 34 and 41.

¶ Já tendo a função que copia um único arquivo para um destino, podemos escrever agora a função que percorre recursivamente um diretório de origem entrando nos diretórios e percorrendo todos os arquivos para copiá-los. Ela assume que o diretório de destino possui a mesma estrutura de diretórios que o de origem e copia os arquivos para os seus locais respectivos.

Pode-se notar que é muito mais fácil fazer a tarefa no Linux e sistemas BSD, pois a informação do tipo de arquivo (se é um diretório, por exemplo) pode ser obtida pelo próprio retorno da função *readdir*. Em outros sistemas que não adotam o mesmo padrão, é necessário chamar uma função *stat* adicional para obter a informação.

36 ⟨ Funções auxiliares Weaver 27 ⟩ +≡

```
int copy_files(char *orig, char *dst) { DIR * d = Λ;
    struct dirent *dir;
    d = opendir(orig); if (d) { while ((dir = readdir(d)) ≠ Λ) { char *file;
    file = concatenate(orig, "/", dir→d_name, "");
    if (file ≡ Λ) {
      return 0;
    }
    # if (defined (__linux__) ∨ defined (_BSD_SOURCE)) ∧ defined
        (DT_DIR)
    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      /* Aqui executamos se nesta iteração devemos copiar um
     diretório */
char *new_dst;
new_dst = concatenate(dst, "/", dir→d_name, "");
if (new_dst ≡ Λ) {
  return 0;
}
if (strcmp(dir→d_name, ".") ∧ strcmp(dir→d_name, "..")) {
  if (copy_files(file, new_dst) ≡ 0) {
    free(new_dst);
    free(file);
    closedir(d);
    return 0;
  }
}
free(new_dst); }
else {      /* Aqui executamos se nesta iteração devemos copiar um
       arquivo */
  if (copy_single_file(file, dst) ≡ 0) {
    free(file);
    closedir(d);
    return 0;
  }
}
free(file); } closedir(d); } return 1; }
```

¶

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:

- Criar arquivos `.c` e `.h` base, deixando seus nomes iguais ao nome do módulo criado.
- Adicionar em ambos um código com copyright e licenciamento com o nome do autor, do projeto e ano.
- 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.

- 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.

O código para isso, assumindo que exista a função *write_copyright* para imprimir o comentário de copyright e licenciamento é:

38 ⟨ Caso de uso 5: Criar novo módulo 38 ⟩ ≡

```
if (inside_weaver_directory ∧ have_arg) {
  if (arg_is_valid_module) {
    char *filename;
    FILE *fp;      /* Creating the .c: */
    filename = concatenate(project_path, "src/", argument, ".c", "");
    if (filename ≡ Λ) ERROR( );
    fp = fopen(filename, "w");
    if (fp ≡ Λ) {
      free(filename);
      ERROR( );
    }
    write_copyright(fp, author_name, project_name, year);
    fprintf(fp, "#include␣\"%s.h\"", argument);
    fclose(fp);
    filename[strlen(filename) − 1] = 'h';      /* Creating the .h: */
    fp = fopen(filename, "w");
    if (fp ≡ Λ) {
      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);      /* Updating src/includes.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( );
}
```

This code is used in chunk 8.

Função Auxiliar: Imprimir código de copyright e licenciamento

Para preencher o código de copyright tanto em novos módulos como no `main.c` de novos projetos, podemos usar a função abaixo:

39 ⟨ Funções auxiliares Weaver 27 ⟩ +≡

```
void write_copyright(FILE *fp, char *author_name, char *project_name, int
        year)
{
  char license[] = "/*\nCopyright (c) %s, %d\n\nThis file i\
      s part of %s.\n\n%s is freesoftware: you\
      \ can redistribute it and/or modify\nit u\
      nder the terms of the GNUGeneral Public \
      License as published by\nthe Free Softwa\
      re Foundation, either version 3 of the Li\
      cense, or\n(at your option) any later ve\
      rsion.\n\n%s isdistributed in the hope t\
      hat it will be useful,\nbut WITHOUT ANY \
      WARRANTY;without even the implied warran\
      ty of\nMERCHANTABILITY or FITNESS FOR AP\
      ARTICULAR 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);
}
```

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` (assumindo que existe a função *append_basefile* que faça isso para nos ajudar). 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` (°*755* em octal).

40 ⟨ Caso de uso 6: Criar novo projeto 40 ⟩ ≡

```
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( );
    chdir(argument);
    mkdir(".weaver", °755 );
    mkdir("conf", °755 );
    mkdir("src", °755 );
    mkdir("src/weaver", °755 );
    mkdir("image", °755 );
    mkdir("sound", °755 );
    mkdir("music", °755 );
    dir_name = concatenate(shared_dir, "project", "");
    if (dir_name ≡ Λ) 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 ≡ Λ) 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 ≡ Λ) 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( );
}
```

This code is used in chunk 8.

Função Auxiliar: Adicionando em arquivo aberto conteúdo de arquivo

A função *append_basefile* deve receber 3 argumentos. Um ponteiro para arquivo aberto, um endereço absoluto de diretório e um nome de arquivo. Ela cuidará da parte de concatenar o nomed e diretório com o de arquivo gerando um endereço absoluto de arquivo, abrirá tal arquivo e escreverá no ponteiro para arquivo todo o conteúdo que houver. Se houver algum erro, ela retorna 0. Caso contrário, retorna 1.

41 ⟨ Funções auxiliares Weaver 27 ⟩ +≡

```
int append_basefile(FILE *fp, char *dir, char *file)
{
  int block_size, bytes_read;
  char *buffer, *directory = ".";
  char *path = concatenate(dir, file, "");
  if (path ≡ Λ) return 0;
  FILE *origin;
  ⟨ Descobre tamanho do bloco do sistema de arquivos 35 ⟩
  buffer = (char *) malloc(block_size);
  if (buffer ≡ Λ) {
    free(path);
    return 0;
  }
  origin = fopen(path, "r");
  if (origin ≡ Λ) {
    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:

42 ⟨ Inclui Cabeçalho de Configuração 42 ⟩ ≡
```
#include "conf_begin.h"
#include "../../conf/conf.h"
```
This code is used in chunks 44, 49, 94, 95, 121, 122, 267, 307, and 332.

¶ 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`:

43 ⟨ `project/src/weaver/conf_begin.h` 43 ⟩ ≡
```
#define W_ELF 0
#define W_WEB 1
```
See also chunk 324.

1.6 Funções básicas Weaver

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

44 ⟨ `project/src/weaver/weaver.h` 44 ⟩ ≡

```
#ifndef _weaver_h_
#define _weaver_h_
#ifdef __cplusplus
  extern "C"
  {
#endif
    ⟨ Inclui Cabeçalho de Configuração 42 ⟩
    ⟨ Cabeçalhos Weaver 45 ⟩
#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 preocupar 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.

45 ⟨ Cabeçalhos Weaver 45 ⟩ ≡

```
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*)

See also chunks 48, 88, 90, 91, 93, 99, 106, 120, 139, 148, 165, 170, 172, 174, 180, 181, 183, 186, 188, 190, 194, 195, 217, 233, 243, 245, 256, 258, 260, 274, 308, and 334.

This code is used in chunk 44.

¶ 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*.

46 ⟨ project/src/weaver/weaver.c 46 ⟩ ≡

```
#include "weaver.h"
  ⟨ API Weaver: Definições 149 ⟩
  void _awake_the_weaver (char *filename, unsigned long line )
  {
    ⟨ API Weaver: Inicialização 81 ⟩
  }
  void _may_the_weaver_sleep(void)
  {
    ⟨ API Weaver: Finalização 82 ⟩
    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
    ⟨ API Weaver: Loop Principal 47 ⟩
#if W_TARGET ≡ W_ELF
    glXSwapBuffers(_dpy, _window);
#else
    glFlush( );
#endif
#if W_TARGET ≡ W_ELF
    nanosleep(&req, Λ);
#endif
```

```
}
```

47 ¶⟨ API Weaver: Loop Principal 47 ⟩ ≡ /* A definir... */

See also chunks 126, 158, 163, 167, 179, 202, and 288.
This code is used in chunk 46.

Capítulo 2

Gerenciamento de memória

Alocar memória dinamicamente de uma heap é uma operação cujo tempo gasto nem sempre pode ser previsto. Isso é algo que depende da quantidade de blocos contínuos de memória presentes na heap que o gerenciador organiza. Por sua vez, isso depende muito do padrão de uso das funções `malloc` e `free`, e por isso não é algo fácil de ser previsto.

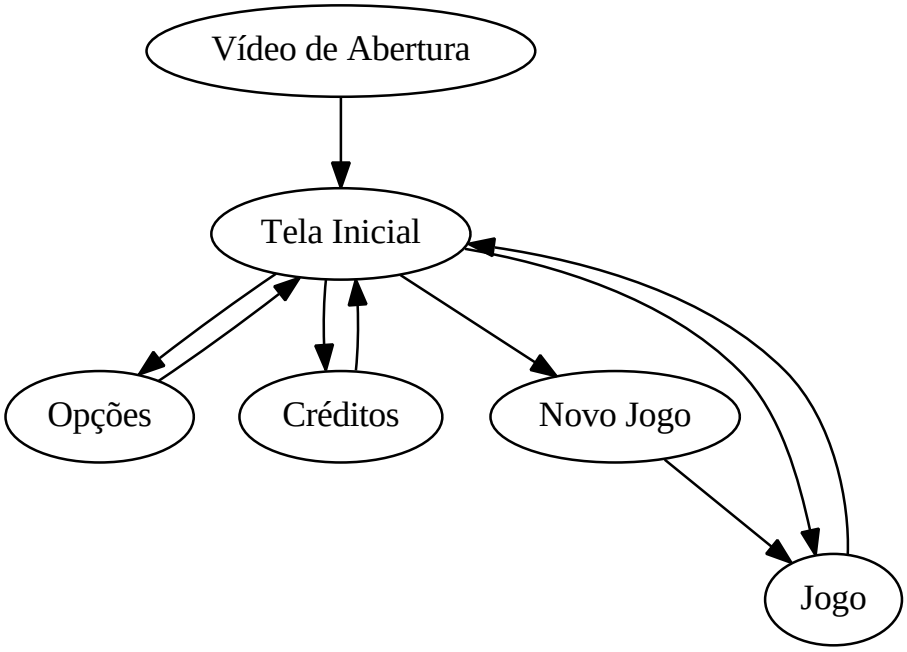
Jogos de computador tradicionalmente evitam o uso contínuo de `malloc` e `free` por causa disso. Tipicamente jogos programados para ter um alto desempenho alocam toda (ou a maior parte) da memória de que vão precisar logo no início da execução gerando um *pool* de memória e gerenciando ele ao longo da execução. De fato, esta preocupação direta com a memória é o principal motivo de linguagens sem *garbage collectors* como C++ serem tão preferidas no desenvolvimento de grandes jogos comerciais.

O gerenciador de memória do Weaver, com o objetivo de permitir que um programador tenha um controle sobre a quantidade máxima de memória que será usada, espera que a quantidade máxima sempre seja declarada previamente. E toda a memória é preparada e alocada durante a inicialização do programa. Caso tente-se alocar mais memória do que o disponível desta forma, uma mensagem de erro será impressa na saída de erro para avisar o que está acontecendo ao programador. Desta forma é difícil deixar passar vazamentos de memória e pode-se estabelecer mais facilmente se o jogo está dentro dos requisitos de sistema esperados.

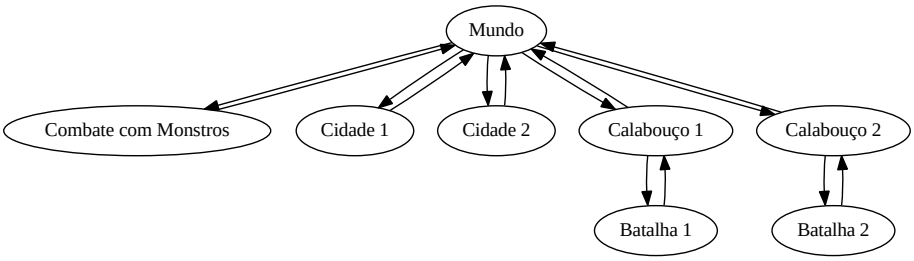
Weaver de fato aloca mais de uma região contínua de memória onde pode-se alocar coisas. Uma das regiões contínuas será alocada e usada pela própria API Weaver à medida que for necessário. A segunda região de memória contínua, cujo tamanho deve ser declarada em `conf/conf.h` é a região dedicada para que o usuário possa alocar por meio de *Walloc* (que funciona como o *malloc*). Além disso, o usuário deve poder criar novas regiões contínuas de memória. O nome que tais regiões recebem é **arena**.

Além de um *Walloc*, também existe um *Wfree*. Entretanto, o jeito recomendável de desalocar na maioria das vezes é usando uma outra função chamada *Wtrash*. Para explicar a ideia de seu funcionamento, repare que tipicamente um

jogo funciona como uma máquina de estados onde mudamos várias vezes de estado. Por exemplo, em um jogo de RPG clássico como Final Fantasy, podemos encontrar os seguintes estados:

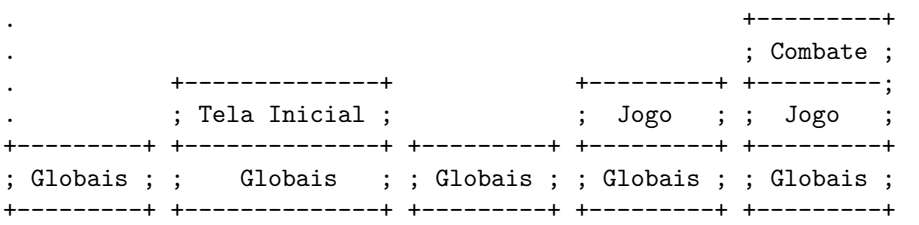


E cada um dos estados pode também ter os seus próprios sub-estados. Por exemplo, o estado "Jogo" seria formado pela seguinte máquina de estados interna:



Cada estado precisará fazer as suas próprias alocações de memória. Algumas vezes, ao passar de um estado pro outro, não precisamos lembrar do quê havia no estado anterior. Por exemplo, quando passamos da tela inicial para o jogo em si, não precisamos mais manter na memória a imagem de fundo da tela inicial. Outras vezes, podemos precisar memorizar coisas. Se estamos andando pelo mundo e somos atacados por monstros, passamos para o estado de combate. Mas uma vez que os monstros sejam derrotados, devemos voltar ao estado anterior, sem esquecer de informações como as coordenadas em que estávamos. Mas quando formos esquecer um estado, iremos querer sempre desalocar toda a memória relacionada à ele.

Por causa disso, um jogo pode ter um gerenciador de memória que funcione como uma pilha. Primeiro alocamos dados globais que serão úteis ao longo de todo o jogo. Todos estes dados só serão desalocados ao término do jogo. Em seguida, podemos criar um **breakpoint** e alocamos todos os dados referentes à tela inicial. Quando passarmos da tela inicial para o jogo em si, podemos desalocar de uma vez tudo o que foi alocado desde o último *breakpoint* e removê-lo. Ao entrar no jogo em si, criamos um novo *breakpoint* e alocamos tudo o que precisamos. Se entramos em tela de combate, criamos outro *breakpoint* (sem desalocar nada e sem remover o *breakpoint* anterior) e alocamos os dados referentes à batalha. Depois que ela termina, desalocamos tudo até o último *breakpoint* para apagarmos os dados relacionados ao combate e voltamos assim ao estado anterior de caminhar pelo mundo. Ao longo destes passos, nossa memória terá aproximadamente a seguinte estrutura:



Sendo assim, nosso gerenciador de memória torna-se capaz de evitar completamente fragmentação tratando a memória alocada na heap como uma pilha. O desenvolvedor só precisa desalocar a memória na ordem inversa da alocação (se não o fizer, então haverá fragmentação). Entretanto, a desalocação pode ser um processo totalmente automatizado. Toda vez que encerramos um estado, podemos ter uma função que desaloca tudo o que foi alocado até o último *breakpoint* na ordem correta e elimina aquele *breakpoint* (exceto o último na base da pilha que não pode ser eliminado). Fazendo isso, o gerenciamento de memória fica mais simples de ser usado, pois o próprio gerenciador poderá desalocar tudo que for necessário, sem esquecer e sem deixar vazamentos de memória. O que a função *Wtrash* faz então é desalocar na ordem certa toda a memória alocada até o último *breakpoint* e destrói o *breakpoint* (exceto o primeiro que nunca é removido). Para criar um novo *breakpoint*, usamos a função *Wbreakpoint*.

Tudo isso sempre é feito na arena padrão. Mas pode-se criar uma nova arena (*Wcreate_arena*) bem como destruir uma arena (*Wdestroy_arena*). E pode-se então alocar memória na arena personalizada criada (*Walloc_arena*) e desalocar (*Wfree_arena*). Da mesmo forma, pode-se também criar um *breakpoint* na arena personalizada (*Wbreakpoint_arena*) e descartar tudo que foi alocado nela até o último *breakpoint* (*Wtrash_arena*).

Para garantir a inclusão da definição de todas estas funções e estruturas, usamos o seguinte código:

48 ⟨ Cabeçalhos Weaver 45 ⟩ +≡
```
#include "memory.h"
```

¶ E também criamos o cabeçalho de memória. À partir de agora, cada novo módulo de Weaver terá um nome associado à ele. O deste é "Memória". E todo cabeçalho `.h` dele conterá, além das macros comuns para impedir que ele seja inserido mais de uma vez e para que ele possa ser usado em C++, uma parte na qual será inserido o cabeçalho de configuração (visto no fim do capítulo anterior) e a parte de declarações, com o nome `Declarações de NOME_DO_MODULO`.

49 ⟨ `project/src/weaver/memory.h` 49 ⟩ ≡
```
#ifndef _memory_h_
#define _memory_h_
#ifdef __cplusplus
  extern "C" {
#endif
      ⟨ Inclui Cabeçalho de Configuração 42 ⟩
      ⟨ Declarações de Memória 50 ⟩
#ifdef __cplusplus
      }
#endif
#endif
```

¶ No caso, as Declarações de Memória que usaremos aqui começam com os cabeçalhos que serão usados, e posteriormente passaraão para as declarações das funções e estruturas de dado a serem usadas nele:

50 ⟨ Declarações de Memória 50 ⟩ ≡
```
#include <sys/mman.h>     /* mmap, munmap */
#include <pthread.h>     /* pthread_mutex_init, pthread_mutex_destroy */
#include <string.h>     /* strncpy */
#include <unistd.h>     /* sysconf */
#include <stdlib.h>     /* size_t */
#include <stdio.h>     /* perror */
#include <math.h>     /* ceil */
#include <stdbool.h>
```
See also chunks 53, 56, 59, 60, 61, 68, 70, 72, 73, 76, 79, 83, and 85.
This code is used in chunk 49.

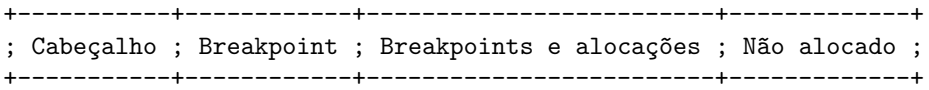
¶ Outra coisa relevante a mencionar é que à partir de agora assumiremos que as seguintes macros são definidas em `conf/conf.h`:

- `W_MAX_MEMORY`: O valor máximo em bytes de memória que iremos alocar por meio da função *Walloc* de alocação de memória na arena padrão.
- `W_WEB_MEMORY`: A quantidade de memória adicional em bytes que reservaremos para uso caso compilemos o nosso jogo para a Web ao invés de gerar um programa executável. O Emscripten precisará de memória adicional e a quantidade pode depender do quanto outras funções como *malloc* e

Walloc_arena são usadas. Este valor deve ser aumentado se forem encontrados problemas de falta de memória na web. Esta macro será consultada na verdade por um dos `Makefiles`, não por código que definiremos neste PDF.

2.1 Estruturas de Dados Usadas

Vamos considerar primeiro uma **arena**. Toda **arena** terá a seguinte estrutura:



2.1.1 Cabeçalho da Arena

O cabeçalho conterá todas as informações que precisamos para usar a arena. Chamaremos sua estrutura de dados de **struct** *arena_header*. O primeiro *breakpoint* nunca pode ser removido e ele é útil para que o comando *Wtrash* sempre funcione e seja definido, pois sempre existirá um último **breakpoint**. Em seguida, virá uma lista que talvez seja vazia (para arenas recém-criadas) de *breakpoints* e regiões de memória alocadas. E por fim, haverá uma região de memória desalocada.

O tamanho total da arena nunca muda. O cabeçalho e primeiro breakpoint também tem tamanho constante. A região de breakpoint e alocações pode crescer e diminuir, mas isso sempre implica que a região não-alocada respectivamente diminui e cresce na mesma proporção.

As informações encontradas no cabeçalho são:

1. **Total:** A quantidade total em bytes de memória que a arena possui. Como precisamos garantir que ele tenha um tamanho suficientemente grande para que alcance qualquer posição que possa ser alcançada por um endereço, ele precisa ser um **size_t**. Pelo padrão ISO isso será no mínimo 2 bytes, mas em computadores pessoais atualmente está chegando a 8 bytes.

   Esta informação será preenchida na inicialização da arena e nunca mais será mudada.

2. **Usado:** A quantidade de memória que já está em uso nesta arena. Isso nos permite verificar se temos espaço disponível ou não para cada alocação. Pelo mesmo motivo do anterior, precisa ser um **size_t**.

   Esta informação precisará ser atualizada toda vez que mais memória for alocada ou desalocada. Ou quando um **breakpoint** for criado ou destruído.

3. **Último Breakpoint:** Armazenar isso nos permite saber à partir de qual posição podemos começar a desalocar memória em caso de um *Wtrash*. Outro **size_t**.

   Eta informaççõn precisa ser atualizada toda vez que um **breakpoint** for criado ou destruído. Um último breakpoint sempre existirá, pois o primeiro breakpoint nunca pode ser removido.

4. **Último Elemento:** Endereço do último elemento que foi armazenado. É útil guardar esta informação porque quando criamos um novo elemento com *Walloc* ou *Wbreakpoint*, o novo elemento precisa apontar para o último que havia antes dele.

   Esta informação precisa ser atualizada após qualquer operação de alocação, desalocação ou **breakpoint**. Sempre existirá um último elemento na arena, pois um primeiro breakpoint sempre estará posicionado após o cabeçalho.

5. **Posição Vazia:** Um ponteiro para a próxima região contínua de memória não-alocada. É preciso saber disso para podermos criar novas estruturas e retornar um espaço ainda não-utilizado em caso de *Walloc*. Outro **size_t**.

   Novamente é algo que precisa ser atualizado após qualquer uma das operações de memória sobre a arena. É possível que não hajam mais regiões vazias caso tudo já tenha sido alocado. Neste caso, o ponteiro deverá ser Λ.

6. **Mutex:** Opcional. Só precisamos definir isso se estivermos usando mais de uma thread. Neste caso, o mutex servirá para prevenir que duas threads tentem modificar qualquer um destes valores ao mesmo tempo.

   Caso seja usado, o mutex precisa ser usado em qualquer operação de memória, pois todas elas precisam modificar elementos da arena.

7. **Uso Máximo:** Opcional. Só precisamos definir isso se estamos rodando o programa em um nível alto de depuração e por isso queremos saber ao fim do uso da arena qual a quantidade máxima de memória que alocamos nela ao longo da execução do programa. Um **size_t**.

   Se estivermos monitorando o valor, precisamos verificar se ele precisa ser atualizado após qualquer alocação ou criação de **breakpoint**.

Então, assim podemos definir o nosso cabeçalho para arenas. Toda arena terá tal estrutura em seu início.

53 ⟨ Declarações de Memória 50 ⟩ +≡

```
struct _arena_header {
    size_t total, used;
    struct _breakpoint *last_breakpoint;
    void *empty_position, *last_element;
#ifdef W_MULTITHREAD
    pthread_mutex_t mutex;
#endif
#if W_DEBUG_LEVEL ≥ 3
    size_t max_used;
#endif
};
```

¶ Pela definição, existem algumas restrições sobre os valores presentes em cabeçalhos de arena: $total \geq max\_used \geq used$, $last\_element \geq last\_breakpoint$, $(empty\_position = NULL) \vee (empty\_position > last\_element)$ e $used > 0$ (o breakpoint e o cabeçalho usam o espaço).

Quando criamos a arena e desejamos inicializar o valor de seu cabeçalho, tudo o que precisamos saber é o tamanho total que a arena tem. Os demais valores podem ser deduzidos. Portanto, podemos usar esta função interna para a tarefa:

54 ⟨ project/src/weaver/memory.c 54 ⟩ ≡

```
#include "memory.h"
  static bool _initialize_arena_header(struct _arena_header *header, size_t
          total)
  {
    header→total = total;
    header→used = sizeof(struct _arena_header) − sizeof(struct
        _breakpoint);
    header→last_breakpoint = (struct _breakpoint *)(header + 1);
    header→last_element = (void *) header→last_breakpoint;
    header→empty_position = (void *)(header→last_breakpoint + 1);
#ifdef W_MULTITHREAD
    if (pthread_mutex_init(&(header→mutex), Λ) ≠ 0) {
      return false;
    }
#endif
#if W_DEBUG_LEVEL ≥ 3
    header→max_used = header→used;
#endif
    return true;
  }
```

See also chunks 57, 58, 62, 65, 69, 71, 74, 75, 77, 80, 84, and 86.

¶ É importante notar que tal função de inicialização só pode falhar se ocorrer algum erro inicializando o mutex. Por isso podemos representar o seu sucesso ou fracasso fazendo-a retornar um valor booleano.

2.1.2 Breakpoints

A função primária de um breakpoint é interagir com as função *Wbreakpoint* e *Wtrash*. As informações que devem estar presentes nele são:

1. **Tipo:** Um número mágico que corresponde sempre à um valor que identifica o elemento como sendo um *breakpoint*, e não um fragmento alocado de memória. Se o elemento realmente for um breakpoint e não possuir um número mágico correspondente, então ocorreu um *buffer overflow* em

memória alocada e podemos acusar isso. Definiremos tal número como #11010101.

2. **Último breakpoint:** No caso do primeiro breakpoint, isso deve apontar para ele próprio (e assim o primeiro breakpoint pode ser identificado diante dos demais). nos demais casos, ele irá apontar para o breakpoint anterior. Desta forma, em caso de *Wtrash*, poderemos restaurar o cabeçalho da arena para apontar para o breakpoint anterior, já que o atual está sendo apagado.

3. **Último Elemento:** Para que a lista de elementos de uma arena possa ser percorrida, cada elemento deve ser capaz de apontar para o elemento anterior. Desta forma, se o breakpoint for removido, podemos restaurar o último elemento da arena para o elemento antes dele (assumindo que não tenha sido marcado para remoção como será visto adiante). O último elemento do primeiro breakpoint é ele próprio.

4. **Arena:** Um ponteiro para a arena à qual pertence a memória.

5. **Tamanho:** A quantidade de memória alocada até o breakpoint em questão. Quando o breakpoint for removido, a quantidade de memória usada pela arena passa a ser o valor presente aqui.

6. **Arquivo:** Opcional para depuração. O nome do arquivo onde esta região da memória foi alocada.

7. **Linha:** Opcional para depuração. O número da linha onde esta região da memória foi alocada.

Sendo assim, a nossa definição de breakpoint é:

56 ⟨ Declarações de Memória 50 ⟩ +≡
```
  struct _breakpoint { unsigned long type;
  void *last_element;
  struct _arena_header *arena;
  struct _breakpoint *last_breakpoint;
  size_t size;
#if W_DEBUG_LEVEL ≥ 1
  char file[32]; unsigned long line ;
#endif
  } ;
```

¶ As seguintes restrições sempre devem valer para tais dados: $type = 0{\times}11010101$, $last\_breakpoint \leq last\_element$.

Pode-se definir uma função para inicializar tais valores. As informações externas necessárias são todos os elementos, exceto o tipo (*type*) e o tamanho que pode ser deduzido pela arena:

```
57 ⟨ project/src/weaver/memory.c 54 ⟩ +≡
  static void _initialize_breakpoint (struct _breakpoint *self, void
      *last_element, struct _arena_header *arena, struct _breakpoint
      *last_breakpoint, char *file, unsigned long line ) {
      self→type = _BREAKPOINT_T;
  self→last_element = last_element;
  self→arena = arena;
  self→last_breakpoint = last_breakpoint;
  self→size = arena→used − sizeof(struct _breakpoint);
#if W_DEBUG_LEVEL ≥ 1
  if ((void *) self→last_breakpoint > self→last_element) { fprintf
      (stderr, "Something␣is␣very␣wrong!␣Arena␣with␣las\
      t␣breakpoint␣greater␣than␣""last␣element:␣%s:␣%lu\n", file,
      line ) ;
  exit(−1); } strncpy(self→file, file, 32); self → line = line ;
#endif
  }
```

¶ Notar que assumimos que quando vamos inicializar um breakpoint, todos os dados do cabeçalho da arena já foram atualizados como tendo o breakpoint já existente. E como consultamos tais dados, o mutex da arena precisa estar bloqueado para que coisas como o tamanho da arena não mudem.

O primeiro dos breakpoints é especial e pode ser inicializado como abaixo. Para ele não precisamos nos preocupar em armazenar o nome de arquivo e número de linha em que é definido.

```
58 ⟨ project/src/weaver/memory.c 54 ⟩ +≡
  static void _initialize_first_breakpoint(struct _breakpoint *self, struct
          _arena_header *arena)
  {
    _initialize_breakpoint(self, self, arena, self, "", 0);
  }
```

¶ Por fim, vamos à definição da memória alocada. Ela é formada basicamente por um cabeçalho, o espaço alocado em si e uma finalização. No caso do cabeçalho, precisamos dos seguintes elementos:

1. **Tipo:** Um número que identifica o elemento como um cabeçalho de dados, não um breakpoint. No caso, usaremos o número mágico $0{\times}10101010$.
2. **Tamanho Real:** Quantos bytes tem a região alocada para dados. É igual ao tamanho pedido mais alguma quantidade adicional de bytes de preenchimento para podermos manter o alinhamento da memória.
3. **Tamanho Pedido:** Quantos bytes foram pedidos na alocação, ignorando o preenchimento.

4. **Último Elemento:** A posição do elemento anterior da arena. Pode ser outro cabeçalho de dado alocado ou um breakpoint. Este ponteiro nos permite acessar os dados como uma lista encadeada.

5. **Arena:** Um ponteiro para a arena à qual pertence a memória.

6. **Flags:** Permite que coloquemos informações adicionais. o último bit é usado para definir se a memória foi marcada para ser apagada ou não.

7. **Arquivo:** Opcional para depuração. O nome do arquivo onde esta região da memória foi alocada.

8. **Linha:** Opcional para depuração. O número da linha onde esta região da memória foi alocada.

As seguintes restrições sempre são válidas para tais dados: $tipo = 0{\times}10101010$, $tamanho\_pedido \leq tamanho\_real$.

A definição de nosso cabeçalho de dados é:

59 ⟨ Declarações de Memória 50 ⟩ +≡
```
  struct _memory_header { unsigned long type;
  void *last_element;
  void *arena;
  size_t real_size, requested_size;
  unsigned long flags;
#if W_DEBUG_LEVEL ≥ 1
  char file[32]; unsigned long line ;
#endif
  } ;
```

¶ E por fim, precisamos definir os 2 números mágicos que mencionamos em nossa descrição das estruturas de memória:

60 ⟨ Declarações de Memória 50 ⟩ +≡
```
#define _BREAKPOINT_T #11010101
#define _DATA_T #10101010
```

2.2 Criando e destruindo arenas

Criar uma nova arena envolve basicamente alocar memória usando *mmap* e tomando o cuidado para alocarmos sempre um número múltiplo do tamanho de uma página (isso garante alinhamento de memória e também nos dá um tamanho ótimo para paginarmos). Em seguida preenchemos o cabeçalho da arena e colocamos o primeiro breakpoint nela.

A função que cria novas arenas deve receber como argumento o tamanho mínimo que ela deve ter em bytes. Já destruir uma arena requer um ponteiro para ela:

61 ⟨ Declarações de Memória 50 ⟩ +≡
```
void *Wcreate_arena(size_t);
int Wdestroy_arena(void *);
```

2.2.1 Criando uma arena

O processo de criar a arena funciona alocando todo o espaço de que precisamos e em seguida preenchendo o cabeçalho inicial e breakpoint:

62 ⟨ project/src/weaver/memory.c 54 ⟩ +≡
```
void *Wcreate_arena(size_t size) { void *arena;
    size_t real_size = 0;
    struct _breakpoint *breakpoint; ⟨ Aloca 'arena' com cerca de 'size'
        bytes e preenche 'real_size' 63 ⟩
    if (arena ≠ Λ) {
      if (¬_initialize_arena_header((struct _arena_header *)
          arena, real_size)) {⟨ Desaloca 'arena' 64 ⟩
      }    /*
          Preenchendo o primeiro breakpoint
          */
      breakpoint = ((struct _arena_header *) arena)→last_breakpoint;
      _initialize_first_breakpoint(breakpoint, (struct _arena_header *)
          arena);
    }
    return arena; }
```

¶ Então usar esta função nos dá como retorno Λ ou um ponteiro para uma nova arena cujo tamanho total é no mínimo o pedido como argumento, mas talvez será maior por motivos de alinhamento e paginação. Partes desta região contínua serão gastos com cabeçalhos da arena, das regiões alocadas e *breakpoints*. Então pode ser que obtenhamos como retorno uma arena onde caibam menos coisas do que caberia no tamanho especificado como argumento.

Mas qual será o tamanho real da arena se não é necessariamente o que pedimos como argumento? Resposta: será o menor tamanho que é maior ou igual ao valor pedido e que seja múltiplo do tamanho de uma página do sistema.

Usamos a chamada de sistema *mmap* para obter a memória. Outra opção seria o *brk*, mas usar tal chamada de sistema criaria conflito caso o usuário tentasse usar o *malloc* da biblioteca padrão ou usasse uma função de biblioteca que usa internamento o *malloc*. Como até um simples *sprintf* usa *malloc*, não podemos usar o *brk*, pois isso criaria muitos conflitos com outras bibliotecas:

63 ⟨ Aloca 'arena' com cerca de 'size' bytes e preenche 'real_size' 63 ⟩ ≡

```
{
  long page_size = sysconf(_SC_PAGESIZE);
  real_size = ((int) ceil((double) size/(double) page_size)) * page_size;
  arena = mmap(0, real_size, PROT_READ | PROT_WRITE,
      MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
  if (arena ≡ MAP_FAILED) {
    arena = Λ;
  }
}
```

This code is used in chunk 62.

¶ E para desalocar uma arena, fazemos simplesmente:

64 ⟨ Desaloca 'arena' 64 ⟩ ≡

```
{
  if (munmap(arena, ((struct _arena_header *) arena)->total) ≡ -1) {
    arena = Λ;
  }
}
```

This code is used in chunks 62 and 65.

2.2.2 Destruindo uma arena

Destruir uma arena é uma simples questão de finalizar o seu mutex caso estejamos criando um programa com muitas threads e usar um *munmap*. Entretanto, se estamos rodando uma versão em desenvolvimento do jogo, com depuração, este será o momento no qual informaremos a existência de vazamentos de memória. E dependendo do nível da depuração, podemos imprimir também a quantidade máxima de memória usada:

65 ⟨ project/src/weaver/memory.c 54 ⟩ +≡

```
  int Wdestroy_arena(void *arena) {
#if W_DEBUG_LEVEL ≥ 1
      struct _arena_header *header = (struct _arena_header *) arena;
          ⟨ Checa vazamento de memória em 'arena' dado seu 'header' 66 ⟩
#endif
#if W_DEBUG_LEVEL ≥ 3
      fprintf (stderr,
          "WARNING (3): Max memory used in arena %s:%lu: %lu/%lu\n",
```

```
        header→file, header → line , (unsigned long) header→max_used,
        (unsigned long) header→total ) ;
#endif
#ifdef W_MULTITHREAD
    {
      struct _arena_header *header = (struct _arena_header *) arena;
      if (pthread_mutex_destroy(&(header→mutex)) ≠ 0) {
        return 0;
      }
    }
#endif
    ⟨ Desaloca 'arena' 64 ⟩
    if (arena ≡ Λ) {
      return 0;
    }
    return 1; }
```

¶ Agora resta apenas definir como checamos a existência de vazamentos de memória. Cada arena tem em seu cabeçalho um ponteiro para seu último elemento. E cada elemento tem um ponteiro para um elemento anterior. Sendo assim, basta percorrermos a lista encadeada e verificarmos se encontramos um cabeçalho de memória alocada que não foi desalocado. Tais cabeçalhos são identificados como tendo o último bit de sua variável *flags* como sendo 1. E devemos percorrer a lista até chegarmos ao primeiro breakpoint.

66 ⟨ Checa vazamento de memória em 'arena' dado seu 'header' 66 ⟩ ≡

```
{ struct _memory_header *p = (struct _memory_header *)
    header→last_element; while (p→type ≠ _BREAKPOINT_T ∨ ((struct
    _breakpoint *) p)→last_breakpoint ≠ (struct _breakpoint *) p) { if
    (p→type ≡ _DATA_T ∧ p→flags % 2) { fprintf (stderr,
    "WARNING␣(1):␣Memory␣leak␣in␣data␣allocated␣in␣%s:%lu\n",
    p→file, p → line ) ; } p = (struct _memory_header *) p→last_element;
    } }
```

This code is used in chunk 65.

¶ A única coisa que não temos como checar é se toda arena criada é depois destruída. Caso um programador decida manipular manualmente suas arenas, ele deverá assumir responsabilidade por isso.

2.3 Alocação e desalocação de memória

68 ⟨ Declarações de Memória 50 ⟩ +≡
```
void * _alloc (void *arena, size_t size, char *filename, unsigned long
    line ) ; void _free (void *mem, char *filename, unsigned long line
    ) ;
```

¶ Alocar memória significa basicamente atualizar informações no cabeçalho de sua arena indicando quanto de memória estamos pegando e atualizando o ponteiro para o último elemento e para o próximo espaço disponível para alocação. Podemos também ter que atualizar qual a quantidade máxima de memória usada por tal arena. E podemos precisar usar um mutex para isso.

Além do cabeçalho da arena, temos também que colocar o cabeçalho da região alocada e o seu rodapé. Mas nesta parte não precisaremos mais segurar o mutex.

Podemos ter que alocar uma quantidade ligeiramente maior que a pedida para preservarmos o alinhamento dos dados na memória. A memória sempre se manterá alinhada com um **long**. O verdadeiro tamanho alocado será armazenado em *real_size*.

O que pode dar errado é que podemos não ter espaço na arena para fazer a alocação. Neste caso, teremos que retornar Λ e se estivermos em fase de depuração, imprimiremos uma mensagem avisando isso:

69 ⟨ `project/src/weaver/memory.c` 54 ⟩ +≡
```
void * _alloc (void *arena, size_t size, char *filename, unsigned long
    line ) { struct _arena_header *header = arena;
  struct _memory_header *mem_header;
  void *mem = Λ, *old_last_element;
#ifdef W_MULTITHREAD
  pthread_mutex_lock(&(header→mutex));
#endif
  mem_header = header→empty_position;
  old_last_element = header→last_element;
    /* Parte 1: Calcular o verdadeiro tamanho a se alocar:
     */
  size_t real_size = (size_t)(ceil((float) size/(float)
    sizeof(long)) * sizeof(long));
    /* Parte 2: Atualizar o cabeçalho da arena:
     */
  if (header→used + real_size + sizeof(struct
    _memory_header) > header→total) {
#if W_DEBUG_LEVEL ≥ 1
  fprintf (stderr,
    "WARNING (1): No memory enough to allocate in %s:%lu.\n",
    filename, line ) ;
```

```
#endif
#ifdef W_MULTITHREAD
  pthread_mutex_unlock(&(header→mutex));
#endif
  return Λ; } header→used += real_size + sizeof(struct _memory_header);
  mem = (void *)((char *) header→empty_position + sizeof(struct
      _memory_header));
  header→last_element = header→empty_position;
  header→empty_position = (void *)((char *) mem + real_size);
#if W_DEBUG_LEVEL ≥ 3
  if (header→used > header→max_used) {
    header→max_used = header→used;
  }
#endif      /* Parte 3: Preencher o cabeçalho do dado a ser alocado:
      */
  mem_header→type = _DATA_T;
  mem_header→last_element = old_last_element;
  mem_header→real_size = real_size;
  mem_header→requested_size = size;
  mem_header→flags = #1;
  mem_header→arena = arena;
#if W_DEBUG_LEVEL ≥ 1
  strncpy(mem_header→file, filename, 32); mem_header → line = line ;
#endif
#ifdef W_MULTITHREAD
  pthread_mutex_unlock(&(header→mutex));
#endif
  return mem; }
```

¶ E agora precisamos só de uma função de macro para cuidar automaticamente da tarefa de coletar o nome de arquivo e número de linha para mensagens de depuração:

70 ⟨ Declarações de Memória 50 ⟩ +≡
```
#define Walloc_arena(a, b)_alloc (a, b, __FILE__, __LINE__)
```

¶ Para desalocar a memória, existem duas possibilidades. Podemos estar desalocando a última memória alocada ou não. No primeiro caso, tudo é uma questão de atualizar o cabeçalho da arena modificando o valor do último elemento armazenado e também um ponteiro pra o próximo espaço vazio. No segundo caso, tudo o que fazemos é marcar o elemento para ser desalocado no futuro.

Caso o elemento realmente seja desalocado (seja o último elemento alocado), temos que percorrer os elementos anteriores desalocando todos aqueles que fo-

ram marcados para desalocar e parar no primeiro elemento que ainda estiver em uso.

```
71 ⟨ project/src/weaver/memory.c 54 ⟩ +≡
  void _free (void *mem, char *filename, unsigned long line ) { struct
      _memory_header *mem_header = ((struct _memory_header *)
      mem) − 1;
  struct _arena_header *arena = mem_header→arena;
  void *last_freed_element;
  size_t memory_freed = 0;
#ifdef W_MULTITHREAD

  pthread_mutex_lock(&(arena→mutex));
#endif    /* Primeiro checamos se não estamos desalocando a última
      memória. Se */    /* é a última memória, precisamos manter o mutex
      ativo para impedir */
     /* que hajam novas escritas na memória depois dela no momento: */
  if ((struct _memory_header *) arena→last_element ≠ mem_header) {
#ifdef W_MULTITHREAD
  pthread_mutex_unlock(&(arena→mutex));
#endif
  mem_header→flags = #0;
#if W_DEBUG_LEVEL ≥ 2
  fprintf (stderr, "WARNING␣(2):␣%s:%lu:␣Memory␣allocated␣i\
      n␣%s:%lu␣should␣be""␣freed␣first.\n", filename, line ,
      ((struct _memory_header *)(arena→last_element))→file, ((struct
      _memory_header *)(arena→last_element)) → line ) ;
#endif
  return; } memory_freed = mem_header→real_size + sizeof(struct
      _memory_header);
  last_freed_element = mem_header;
  mem_header = mem_header→last_element;
  while (mem_header→type ≠ _BREAKPOINT_T ∧ mem_header→flags ≡ #0) {
    memory_freed += mem_header→real_size + sizeof(struct
        _memory_header);
    last_freed_element = mem_header;
    mem_header = mem_header→last_element;
  }
  arena→last_element = mem_header;
  arena→empty_position = last_freed_element;
  arena→used −= memory_freed;
#ifdef W_MULTITHREAD
  pthread_mutex_unlock(&(arena→mutex));
#endif
  }
```

¶ E agora a macro que automatiza a obtenção do nome de arquivo e número de linha:

72 ⟨ Declarações de Memória 50 ⟩ +≡
#define *Wfree*(*a*)*_free* (*a*, `__FILE__`, `__LINE__`)

2.4 Usando a heap descartável

Graças ao conceito de *breakpoints*, pode-se desalocar ao mesmo tempo todos os elementos alocados desde o último *breakpoint* por meio do *Wtrash*. A criação de um *breakpoit* e descarte de memória até ele se dá por meio das funções declaradas abaixo:

73 ⟨ Declarações de Memória 50 ⟩ +≡
```
int _new_breakpoint (void *arena, char *filename, unsigned long line ) ;
void _trash(void *arena);
```

¶ As funções precisam receber como argumento apenas um ponteiro para a arena na qual realizar a operação. Além disso, elas recebem também o nome de arquivo e número de linha como nos casos anteriores para que isso ajude na depuração:

74 ⟨ project/src/weaver/memory.c 54 ⟩ +≡
```
int _new_breakpoint (void *arena, char *filename, unsigned long line ) {
    struct _arena_header *header = (struct _arena_header *) arena;
  struct _breakpoint *breakpoint, *old_breakpoint;
  void *old_last_element;
  size_t old_size;
#ifdef W_MULTITHREAD
  pthread_mutex_lock(&(header->mutex));
#endif
  if (header->used + sizeof(struct _breakpoint) > header->total) {
#if W_DEBUG_LEVEL ≥ 1
  fprintf (stderr,
      "WARNING (1): No memory enough to allocate in %s:%lu.\n",
      filename, line ) ;
#endif
#ifdef W_MULTITHREAD
  pthread_mutex_unlock(&(header->mutex));
#endif
  return 0; } old_breakpoint = header->last_breakpoint;
  old_last_element = header->last_element;
  old_size = header->used;
  header->used += sizeof(struct _breakpoint);
  breakpoint = (struct _breakpoint *) header->empty_position;
  header->last_breakpoint = breakpoint;
  header->empty_position = ((struct _breakpoint *)
      header->empty_position) + 1;
  header->last_element = header->last_breakpoint;
#if W_DEBUG_LEVEL ≥ 3
  if (header->used > header->max_used) {
    header->max_used = header->used;
```

```
  }
#endif
#ifdef W_MULTITHREAD
  pthread_mutex_unlock(&(header→mutex));
#endif
  breakpoint→type = _BREAKPOINT_T;
  breakpoint→last_element = old_last_element;
  breakpoint→arena = arena;
  breakpoint→last_breakpoint = (void *) old_breakpoint;
  breakpoint→size = old_size;
#if W_DEBUG_LEVEL ≥ 1
  strncpy(breakpoint→file, filename, 32); breakpoint → line = line ;
#endif
  return 1; }
```

¶ E a função para descartar toda a memória presente na heap até o último breakpoint:

75 ⟨ project/src/weaver/memory.c 54 ⟩ +≡

```
  void _trash(void *arena)
  {
    struct _arena_header *header = (struct _arena_header *) arena;
    struct _breakpoint *previous_breakpoint = ((struct _breakpoint *)
        header→last_breakpoint)→last_breakpoint;
#ifdef W_MULTITHREAD
    pthread_mutex_lock(&(header→mutex));
#endif
    if (header→last_breakpoint ≡ previous_breakpoint) {
      header→last_element = previous_breakpoint;
      header→empty_position = (void *)(previous_breakpoint + 1);
      header→used = previous_breakpoint→size + sizeof(struct _breakpoint);
    }
    else {
      struct _breakpoint *last = (struct _breakpoint *)
          header→last_breakpoint;
      header→used = last→size;
      header→empty_position = last;
      header→last_element = last→last_element;
      header→last_breakpoint = previous_breakpoint;
      ;
    }
#ifdef W_MULTITHREAD
    pthread_mutex_unlock(&(header→mutex));
#endif
  }
```

¶ E para finalizar, as macros necessárias para usarmos as funções sem nos preocuparmos com o nome do arquivo e número de linha:

76 ⟨ Declarações de Memória 50 ⟩ +≡
#**define** *Wbreakpoint_arena*(*a*)*_new_breakpoint* (*a*, `__FILE__`, `__LINE__`)
#**define** *Wtrash_arena*(*a*)*_trash* (*a*)

2.5 Usando as arenas de memória padrão

Ter que se preocupar com arenas muitas vezes é desnecessário. O usuário pode querer simplesmente usar uma função *Walloc* sem ter que se preocupar com qual arena usar. Usando simplesmente a arena padrão. E associada à ela deve haver as funções *Wfree*, *Wbreakpoint* e *Wtrash*.

Primeiro precisaremos declarar duas variáveis globais. Uma delas será uma arena padrão do usuário, a outra deverá ser uma arena usada pelas funções internas da própria API. Ambas as variáveis devem ficar restritas ao módulo de memória, então serão declaradas como estáticas:

77 ⟨ `project/src/weaver/memory.c` 54 ⟩ +≡
```
static void *_user_arena, *_internal_arena;
```

¶

¶ Vamos precisar inicializar e finalizar estas arenas com as seguinte funções:

79 ⟨ Declarações de Memória 50 ⟩ +≡
```
void _initialize_memory( );
void _finalize_memory( );
```

¶ Note que são funções que sabem o nome do arquivo e número de linha em que estão para propósito de depuração. Elas são definidas como sendo:

80 ⟨ `project/src/weaver/memory.c` 54 ⟩ +≡
```
void _initialize_memory(void)
{
  _user_arena = Wcreate_arena(W_MAX_MEMORY);
  _internal_arena = Wcreate_arena(4000);
}
void _finalize_memory( )
{
  Wdestroy_arena(_user_arena);
  Wtrash_arena(_internal_arena);
  Wdestroy_arena(_internal_arena);
}
```

¶ Passamos adiante o número de linha e nome do arquivo para a função de criar as arenas. Isso ocorre porque um usuário nunca invocará diretamente estas funções. Quem vai chamar tal função é a função de inicialização da API. Se uma mensagem de erro for escrita, ela deve conter o nome de arquivo e número de linha onde está a própria função de inicialização da API. Não onde tais funções estão definidas.

A invocação destas funções se dá na inicialização da API, a qual é mencionada na Introdução. Da mesma forma, na finalização da API, chamamos a função de finalização:

81 ⟨ API Weaver: Inicialização 81 ⟩ ≡
_*initialize_memory* (*filename*, **line**) ;

See also chunks 96, 123, 150, 157, 162, 166, 171, 196, 204, 210, 212, 214, 215, 218, 220, 222, 223, 228, 235, 244, 250, 257, 276, 303, 314, 326, 327, 329, and 351.

This code is used in chunk 46.

82 ¶⟨ API Weaver: Finalização 82 ⟩ ≡
/∗ Em "desalocações" desalocamos memória alocada com *Walloc*: ∗/
⟨ API Weaver: Desalocações 281 ⟩
_*finalize_memory*( );

See also chunks 97, 124, 213, 219, and 221.

This code is used in chunk 46.

¶ Agora para podermos alocar e desalocar memória da arena padrão e da arena interna, criaremos a seguinte funções:

83 ⟨ Declarações de Memória 50 ⟩ +≡
void ∗ _*Walloc* (**size_t** *size*, **char** ∗*filename*, **unsigned int line**) ;
#**define** *Walloc*(*n*)_*Walloc* (*n*, `__FILE__`, `__LINE__`)
void ∗ _*Winternal_alloc* (**size_t** *size*, **char** ∗*filename*, **unsigned int line**) ;
#**define** _*iWalloc*(*n*)_*Winternal_alloc* (*n*, `__FILE__`, `__LINE__`)

84 ¶⟨ `project/src/weaver/memory.c` 54 ⟩ +≡
void ∗ _*Walloc* (**size_t** *size*, **char** ∗*filename*, **unsigned int line**) { **return** _*alloc* (_*user_arena*, *size*, *filename*, **line**) ; } **void** ∗ _*Winternal_alloc* (**size_t** *size*, **char** ∗*filename*, **unsigned int line**) { **return** _*alloc* (_*internal_arena*, *size*, *filename*, **line**) ; }

¶ O *Wfree* já foi definido e irá funcionar sem problemas, independente da arena à qual pertence o trecho de memória alocado. Sendo assim, resta definir apenas o *Wbreakpoint* e *Wtrash*:

85 ⟨ Declarações de Memória 50 ⟩ +≡
int _*Wbreakpoint* (**char** ∗*filename*, **unsigned long line**) ;
void _*Wtrash*( );
#**define** *Wbreakpoint*( )_*Wbreakpoint* (`__FILE__`, `__LINE__`)
#**define** *Wtrash*( )_*Wtrash* ( )

¶ E a definição das funções segue abaixo:

86 ⟨ `project/src/weaver/memory.c` 54 ⟩ +≡

```
int _Wbreakpoint (char *filename, unsigned long line ) { return
        _new_breakpoint (_user_arena, filename, line ) ; } void _Wtrash( )
{
  _trash(_user_arena);
}
```

¶

2.6 Medindo o desempenho

Existem duas macros que são úteis de serem definidas que podem ser usadas para avaliar o desempenho do gerenciador de memória definido aqui. Elas são:

88 ⟨ Cabeçalhos Weaver 45 ⟩ +≡

```
#include <stdio.h>
#include <sys/time.h>
#define W_TIMER_BEGIN( ) { struct timeval _begin, _end;
        gettimeofday(&_begin, Λ);
#define W_TIMER_END( )gettimeofday(&_end, Λ);
        printf("%ld us\n",
            (1000000*(_end.tv_sec−_begin.tv_sec)+_end.tv_usec−_begin.tv_usec));
            }
```

¶ Como a primeira macro inicia um bloco e a segunda termina, ambas devem ser sempre usadas dentro de um mesmo bloco de código, ou um erro ocorrerá. O que elas fazem nada mais é do que usar *gettimeofday* e usar a estrutura retornada para calcular quantos microssegundos se passaram entre uma invocação e outra. Em seguida, escreve-se na saída padrão quantos microssegundos se passaram.

Como exemplo de uso das macros, podemos usar a seguinte função *main* para obtermos uma medida de performance das funções *Walloc* e *Wfree*:

```
int main(int argc, char **argv){
unsigned long i;
void *m[1000000];
Winit();
W_TIMER_BEGIN();
for(i = 0; i < 1000000; i ++){
m[i] = Walloc(1);
}
for(i = 0; i < 1000000; i ++){
Wfree(m[i]);
}
Wtrash();
W_TIMER_END();
Wexit();
return 0;
}
```

Rodando este código em um Pentium B980 2.40GHz Dual Core, obtemos os seguintes resultados para o *Walloc*/*Wfree* (em vermelho) comparado com o *malloc*/*free* (em azul) da biblioteca padrão (Glibc 2.20) comparado ainda com a substituição do segundo loop por uma única chamada para *Wtrash* (em verde):

O alto desempenho do uso de *Walloc*/*Wtrash* é compreensível pelo fato da função *Wtrash* desalocar todo o espaço ocupado pelo último milhão de alocações

no mesmo tempo que *Wfree* levaria para desalocar uma só alocação. Isso explica o fato de termos reduzido pela metade o tempo de execução do exemplo.

Entretanto, tais resultados positivos só são obtidos caso usemos a macro `W_DEBUG_LEVEL` ajustada para zero, como é recomendado fazer ao compilar um jogo pela última vez antes de distribuir. Caso o jogo ainda esteja em desenvolvimento e tal macro tenha um valor maior do que zero, o desempenho de *Walloc* e *Wfree* pode tornar-se de duas à vinte vezes pior devido à estruturas adicionais estarem sendo usadas para depuração e devido à mensagens poderem ser escritas na saída padrão.

Os bons resultados são ainda mais visíveis caso compilemos nosso programa para a Web (ajustando a macro `W_TARGET` para `W_WEB`). Neste caso, o desempenho do *malloc* tem uma queda brutal. Ele passa a executar 20 vezes mais lentamente no exemplo acima, enquanto as funções que desenvolvemos ficam só 1,8 vezes mais lentas. É até difícil mostrar isso em gráfico devido à diferença de escala entre as medidas. Nos testes, usou-se o Emscripten versão 1.34.

Mas e se usarmos várias threads para realizarmos este milhão de alocações nesta máquina com 2 processadores? Supondo que exista a função *test* que realiza todas as alocações e desalocações de um milhão de posições de memória divididas pelo número de threads e supondo que executemos o seguinte código:

```
int main(int argc, char **argv){
pthread_t threads[NUM_THREADS];
int i;
Winit();
for(i = 0; i < NUM_THREADS; i ++)
pthread_create(&threads[i], NULL, test, (void *) NULL);
W_TIMER_BEGIN();
for (i = 0; i < NUM_THREADS; i++)
pthread_join(threads[i], NULL);
W_TIMER_END();
Wexit();
pthread_exit(NULL);
return 0;
}
```

O resultado é este:

O desempenho de *Walloc* e *Wfree* (em vermelho) passa a deixar muito à desejar comparado com o uso de *malloc* e *free* (em azul). Isso ocorre porque na nossa função de alocação, para alocarmos e desalocarmos, precisamos bloquear um mutex. Desta forma, neste exemplo, como tudo o que as threads fazem é alocar e desalocar, na maior parte do tempo elas ficam bloqueadas. As funções *malloc* e *free* da biblioteca padrão não sofrem com este problema, pois cada thread sempre possui a sua própria arena para alocação. Nós não podemos fazer isso automaticamente porque no nosso gerenciador de memória, para que possamos realizar otimizações, precisamos saber com antecedência qual a quantidade máxima de memória que iremos alocar. Não temos como deduzir este valor para cada thread.

Mas nós podemos criar manualmente arenas ara as nossas threads por meio de *Wcreate_arena* e depois podemos usar *Wdestroy_arena* pouco antes da thread encerrar. Desta forma podemos usar *Walloc_arena* para alocar a memória em uma arena particular da thread. Com isso, conseguimos desempenho equivalente ao *malloc* para uma ou duas threads. Para mais threads, conseguimos um desempenho ainda melhor em relação ao *malloc*, já que nosso desempenho não sofre tanta degradação se usamos mais threads que o número de processadores. Podemos analisar o desempenho no gráfico mais abaixo por meio da cor verde.

Mas se reservamos manualmente uma arena para cada thread, então somos capazes de desalocar toda a memória da arena por meio da *Wtrash_arena*. Sendo assim, economizamos o tempo que seria gasto desalocando memória. O desempenho desta forma de uso do nosso alocador pode ser visto no gráfico em amarelo.

O uso de threads na web por meio de Emscripten no momento da escrita deste texto ainda está experimental. Somente o Firefox Nightly suporta o recurso no momento. Por este motivo, testes de desempenho envolvendo threads em programas web ficarão pendentes.

Capítulo 3

Criando uma janela ou espaço de desenho

Para que tenhamos um jogo, precisamos de gráficos. E também precisamos de um local onde desenharmos os gráficos. Em um jogo compilado para Desktop, tipicamente criaremos uma janela na qual invocaremos funções OpenGL. Em um jogo compilado para a Web, tudo será mais fácil, pois não precisaremos de uma janela especial. Por padrão já teremos um "*canvas*" para manipular com WebGL. Portanto, o código para estes dois cenários irá diferir bastante neste capítulo. De qualquer forma, ambos usarão OpenGL:

90 ⟨ Cabeçalhos Weaver 45 ⟩ +≡
```
#include <GL/glew.h>
```

¶ Para criar uma janela, usaremos o Xlib ao invés de bibliotecas de mais alto nível. Primeiro porque muitas bibliotecas de alto nível como SDL parecem não funcionar bem em ambientes gráficos mais excêntricos como o *ratpoison*, o qual eu uso. Em especial quando tentam usar a tela-cheia. Durante um tempo usei também o Xmonad, no qual algumas bibliotecas não conseguiam deixar suas janelas em tela-cheia. Além disso, o Xlib é uma biblioteca bastante universal. Geralmente se um sistema não tem o X, é porque ele não tem interface gráfica e não iria rodar um jogo mesmo.

O nosso arquivo `conf/conf.h` precisará de duas macros novas para estabelecermos o tamanho de nossa janela (ou do "canvas" para a Web):

- `W_DEFAULT_COLOR`: A cor padrão da janela, a ser exibida na ausência de qualquer outra coisa para desenhar. Representada como três números em ponto flutuante separados por vírgulas.
- `W_HEIGHT`: A altura da janela ou do "canvas". Se for definido como zero, será o maior tamanho possível.

- `W_WIDTH`: A largura da janela ou do "canvas". Se for definido como zero, será o maior tamanho possível.

Por padrão, ambos serão definidos como zero, o que tem o efeito de deixar o programa em tela-cheia.

Vamos precisar definir também duas variáveis globais que armazenarão o tamanho da janela em que estamos e duas outras para saber em que posição da tela está nossa janela. Se estivermos rodando o jogo em um navegador, seus valores nunca mudarão, e serão os que forem indicados por tais macros. Mas se o jogo estiver rodando em uma janela, um usuário pode querer modificar seu tamanho. Ou alternativamente, o próprio jogo pode pedir para ter o tamanho de sua janela modificado.

Saber a altura e largura da janela em que estamos tem importância central para podermos desenhar na tela uma interface. Saber a posição da janela é muito menos útil. Entretanto, podemos pensar em conceitos experimentais de jogos que podem levar em conta tal informação. Talvez possa-se criar uma janela que tente evitar ser fechada movendo-se caso o mouse aproxime-se dela para fechá-la. Ou um jogo que crie uma janela que ao ser movida pela Área de trabalho possa revelar imagens diferentes, como se funcionasse como um raio-x da tela.

91 ⟨ Cabeçalhos Weaver 45 ⟩ +≡
```
extern int W_width, W_height, W_x, W_y;
```

¶ Estas variáveis precisarão ser atualizadas caso o tamanho da janela mude e caso a janela seja movida.

3.1 Criar janelas

O código de criar janelas só será usado se estivermos compilando um programa nativo. Por isso, só iremos definir e declarar suas funções se a macro `W_TARGET` for igual à `W_ELF`. Por isso, realizamos a importação do cabeçalho condicionalmente:

93 ⟨ Cabeçalhos Weaver 45 ⟩ +≡
```
#if W_TARGET ≡ W_ELF
#include "window.h"
#endif
```

¶ E o cabeçalho em si terá a forma:

94 ⟨ project/src/weaver/window.h 94 ⟩ ≡
```
#ifndef _window_H_
#define _window_h_
#ifdef __cplusplus
  extern "C"
  {
#endif
    ⟨ Inclui Cabeçalho de Configuração 42 ⟩
#include "weaver.h"
#include <signal.h>
#include <stdio.h>     /* fprintf */
#include <stdlib.h>     /* exit */
#include <X11/Xlib.h>
        /* XOpenDisplay, XCloseDisplay, DefaultScreen, */
        /* DisplayPlanes, XFree, XCreateSimpleWindow, */
        /* XDestroyWindow, XChangeWindowAttributes, */
        /* XSelectInput, XMapWindow, XNextEvent, */
        /* XSetInputFocus, XStoreName, */
#include <GL/gl.h>
#include <GL/glx.h>
        /* glXChooseVisual, glXCreateContext, glXMakeCurrent */
#include <X11/extensions/Xrandr.h>     /* XRRSizes, XRRRates,
         XRRGetScreenInfo, */     /* XRRConfigCurrentRate, */
        /* XRRConfigCurrentConfiguration, */
        /* XRRFreeScreenConfigInfo, */
        /* XRRSetScreenConfigAndRate */
#include <X11/XKBlib.h>     /* XkbKeycodeToKeysym */
         void _initialize_window(void);
         void _finalize_window(void);
    ⟨ Janela: Declaração 115 ⟩
#ifdef __cplusplus
  }
#endif
```

```
#endif
```

¶ Enquanto o próprio arquivo de definição de funções as definirá apenas condicionalmente:

95 ⟨ project/src/weaver/window.c 95 ⟩ ≡
```
⟨ Inclui Cabeçalho de Configuração 42 ⟩
    extern int make_iso_compilers_happy;
#if W_TARGET ≡ W_ELF
#include "window.h"
    int W_width, W_height, W_x, W_y; ⟨ Variáveis de Janela 98 ⟩
        void _initialize_window(void) {⟨ Janela: Inicialização 100 ⟩
            } void _finalize_window(void) {⟨ Janela: Pré-Finalização 145 ⟩
                ⟨ Janela: Finalização 118 ⟩
                }⟨ Janela: Definição 129 ⟩
#endif
```

¶ Desta forma, nada disso será incluído desnecessariamente quando compilarmos para a Web. Mas caso seja incluso, precisamos invocar uma função de inicialização e finalização na inicialização e finalização da API:

96 ⟨ API Weaver: Inicialização 81 ⟩ +≡
```
#if W_TARGET ≡ W_ELF
  _initialize_window( );
#endif
```

97 ¶⟨ API Weaver: Finalização 82 ⟩ +≡
```
#if W_TARGET ≡ W_ELF
  _finalize_window( ); ⟨ Restaura os Sinais do Programa (SIGINT, SIGTERM,
      etc) 151 ⟩
#endif
```

¶ Para que possamos criar uma janela, como o Xlib funciona segundo um modelo cliente-servidor, precisaremos de uma conexão com tal servidor. Tipicamente, tal conexão é chamada de "Display". Na verdade, além de ser uma conexão, um Display também armazena informações sobre o servidor com o qual nos conectamos. Como ter acesso à conexão é necessário para fazer muitas coisas diferentes, tais como obter entrada e saída, teremos que definir o nosso display como variável global para que esteja acessível para outros módulos.

98 ⟨ Variáveis de Janela 98 ⟩ ≡
```
  Display * _dpy;
```
See also chunks 101, 103, 105, 108, 112, 113, 137, and 141.
This code is used in chunk 95.

99 ¶⟨ Cabeçalhos Weaver 45 ⟩ +≡
```
#if W_TARGET ≡ W_ELF
#include <X11/Xlib.h>
  extern Display*_dpy;
#endif
```

¶ Ao inicializar uma conexão, o que pode dar errado é que podemos fracassar, talvez por o servidor não estar ativo. Como iremos abrir uma conexão com o servidor na própria máquina em que estamos executando, então não é necessário passar qualquer argumento para a função *XOpenDisplay*:

100 ⟨ Janela: Inicialização 100 ⟩ ≡
```
_dpy = XOpenDisplay(Λ);
if (_dpy ≡ Λ) {
  fprintf(stderr, "ERROR: Couldn't connect with the X Serv\
      er. Are you running a ""graphical interface?\n");
  exit(1);
}
```
See also chunks 102, 104, 107, 109, 110, 111, 114, 116, and 142.

This code is used in chunk 95.

¶ Nosso próximo passo será obter o número da tela na qual a janela estará. Teoricamente um dispositivo pode ter várias telas diferentes. Na prática provavelmente só encontraremos uma. Caso uma pessoa tenha duas, ela provavelmente ativa a extensão Xinerama, que faz com que suas duas telas sejam tratadas como uma só (tipicamente com uma largura bem grande). De qualquer forma, obter o ID desta tela será importante para obtermos alguns dados como a resolução máxima e quantidade de bits usado em cores.

101 ⟨ Variáveis de Janela 98 ⟩ +≡
```
static int screen;
```

¶ Para inicializar o valor, usamos a seguinte macro, a qual nunca falhará:

102 ⟨ Janela: Inicialização 100 ⟩ +≡
```
screen = DefaultScreen(_dpy);
```

¶ Como a tela é um inteiro, não há nada que precisemos desalocar depois. E de posse do ID da tela, podemos obter algumas informações à mais como a profundidade dela. Ou seja, quantos bits são usados para representar as cores.

103 ⟨ Variáveis de Janela 98 ⟩ +≡
```
static int depth;
```

¶ No momento da escrita deste texto, o valor típico da profundidade de bits é de 24. Assim, as cores vermelho, verde e azul ficam cada uma com 8 bits (totalizando 24) e 8 bits restantes ficam representando um valor alpha que armazena informação de transparência.

104 ⟨ Janela: Inicialização 100 ⟩ +≡
```
depth = DisplayPlanes(_dpy, screen); # if W_DEBUG_LEVEL ≥
    3printf("WARNING (3): Color depth: %d\n", depth); # endif
```

¶ De posse destas informaões, já podemos criar a nossa janela. Ela é declarada assim:

105 ⟨ Variáveis de Janela 98 ⟩ +≡
```
Window _window;
```

106 ¶⟨ Cabeçalhos Weaver 45 ⟩ +≡
```
#if W_TARGET ≡ W_ELF
#include <X11/Xlib.h>
  extern Window _window;
#endif
```

¶ E é inicializada com os seguintes dados:

107 ⟨ Janela: Inicialização 100 ⟩ +≡
```
  W_x = 0;
  W_y = 0;
#if W_WIDTH > 0
  W_width = W_WIDTH;
#else
  W_width = DisplayWidth(_dpy, screen);
#endif
#if W_HEIGHT > 0
  W_height = W_HEIGHT;
#else
  W_height = DisplayHeight(_dpy, screen);
#endif
  _window = XCreateSimpleWindow(_dpy,
      /* Conexão com o servidor X */
  DefaultRootWindow(_dpy),      /* A janeela-mãe */
  W_x, W_y,      /* Coordenadas da janela */
  W_width,      /* Largura da janela */
  W_height,      /* Altura da janela */
  0, 0,      /* Borda (espessura e cor) */
  0);      /* Cor padrão */
```

¶ Agora já podemos criar uma janela. Mas isso não quer dizer que a janela será exibida depois de criada. Ainda temos que fazer algumas coisas como mudar alguns atributos de configuração da janela. E só depois disso poderemos pedir para que o servidor mostre a janela visualmente.

Vamos nos concentrar agora nos atributos da janela. Primeiro nós queremos que nossas escolhas de configuração sejam as mais soberanas possíveis. Devemos pedir que o gerenciador de janelas faça todo o possível para cumpri-las. Por isso, começamos ajustando a flag "Override Redirect", o que propagandeia nossa janela como uma janela de"pop-up". Isso faz com que nossos pedidos de entrar em tela cheia sejam atendidos, mesmo quando estamos em ambientes como o XMonad.

A próxima coisa que fazemos é informar quais eventos devem ser notificados para nossa janela. No caso, queremos ser avisados quando um botão é pressionado, liberado, bem como botões do mouse e quando a janela é revelada ou tem o seu tamanho mudado.

E por fim, mudamos tais atributos na janela e fazemos o pedido para começarmos a ser notificados de quando houverem eventos de entrada:

108 ⟨ Variáveis de Janela 98 ⟩ +≡
```
static XSetWindowAttributes at;
```

109 ¶⟨ Janela: Inicialização 100 ⟩ +≡
```
{
    at.override_redirect = True;
    at.event_mask = ButtonPressMask | ButtonReleaseMask | KeyPressMask |
        KeyReleaseMask | PointerMotionMask | ExposureMask |
        StructureNotifyMask;
    XChangeWindowAttributes(_dpy, _window, CWOverrideRedirect, &at);
    XSelectInput(_dpy, _window, StructureNotifyMask | KeyPressMask |
        KeyReleaseMask | ButtonPressMask | ButtonReleaseMask |
        PointerMotionMask | ExposureMask | StructureNotifyMask);
}
```

¶ Agora o que enfim podemos fazer é pedir para que a janela seja desenhada na tela. Primeiro pedimos sua criação e depois aguardamos o evento de sua criação. Quando formos notificados do evento, pedimos para que a janela receba foco, mas que devolva o foco para a janela-mãe quando terminar de executar. Ajustamos o nome que aparecerá na barra de título do programa. E se nosso programa tiver várias threads, avisamos o Xlib disso:

110 ⟨ Janela: Inicialização 100 ⟩ +≡
```
XMapWindow(_dpy, _window);
{
    XEvente;
    XNextEvent(_dpy, &e);
```

```
    while (e.type ≠ MapNotify) {
      XNextEvent(_dpy, &e);
    }
  }
  XSetInputFocus(_dpy, _window, RevertToParent, CurrentTime);
#ifdef W_PROGRAM_NAME
  XStoreName(_dpy, _window, W_PROGRAM_NAME);
#else
  XStoreName(_dpy, _window, W_PROG);
#endif
#ifdef W_MULTITHREAD
  XInitThreads( );
#endif
```

¶ Antes de inicializarmos o código para OpenGL, precisamos garantir que tenhamos uma versão do GLX de pelo menos 1.3. Antes disso, não poderíamos ajustar as configurações do contexto OpenGL como queremos. Sendo assim, primeiro precisamos checar se estamos com uma versão compatível:

111 ⟨ Janela: Inicialização 100 ⟩ +≡

```
  {
    int glx_major, glx_minor;
    Bool ret;
    ret = glXQueryVersion(_dpy, &glx_major, &glx_minor);
    if (¬ret ∨ ((glx_major ≡ 1) ∧ (glx_minor < 3)) ∨ glx_major < 1) {
      fprintf(stderr, "ERROR: GLX is version %d.%d, but should\
          be at least 1.3.\n", glx_major, glx_minor);
      exit(1);
    }
  }
```

¶ A última coisa que precisamos fazer agora na inicialização é criar um contexto OpenGL e associá-lo à nossa recém-criada janela para que possamos usar OpenGL nela:

112 ⟨ Variáveis de Janela 98 ⟩ +≡

```
  static GLXContext context;
```

¶ Também vamos precisar de configurações válidas para o nosso contexto:

113 ⟨ Variáveis de Janela 98 ⟩ +≡

```
  static GLXFBConfig *fbConfigs;
```

¶ Estas são as configurações que queremos para termos uma janela colorida que pode ser desenhada e com buffer duplo.

114 ⟨ Janela: Inicialização 100 ⟩ +≡

```
{
  int return_value;
  int doubleBufferAttributes[] = {GLX_DRAWABLE_TYPE, GLX_WINDOW_BIT,
      GLX_RENDER_TYPE, GLX_RGBA_BIT, GLX_DOUBLEBUFFER, True,
      GLX_RED_SIZE, 1, GLX_GREEN_SIZE, 1, GLX_BLUE_SIZE, 1,
      GLX_DEPTH_SIZE, 1, None};
  fbConfigs = glXChooseFBConfig(_dpy, screen, doubleBufferAttributes,
      &return_value);
  if (fbConfigs ≡ Λ) {
    fprintf(stderr, "ERROR: Not possible to create a double-\
        buffered window.\n");
    exit(1);
  }
}
```

¶ Agora iremos precisar usar uma função chamada *glXCreateContextAttribsARB* para criar um contexto OpenGL 3.0. O problema é que nem todas as placas de vídeo possuem ela. Algumas podem não ter suporte à versões mais novas do openGL. Por causa disso, esta função não está delarada em nenhum cabeçalho. Nós mesmos precisamos declará-la e obter o seu valor dinamicamente se ela existir:

115 ⟨ Janela: Declaração 115 ⟩ ≡

```
typedef GLXContext(*glXCreateContextAttribsARBProc) ( Display * ,
    GLXFBConfig, GLXContext, Bool , const int * ) ;
```

See also chunks 128, 132, 143, 146, and 152.

This code is used in chunk 94.

¶ Tendo declarado o novo tipo, tentamos obter a função e usá-la para criar o contexto:.

116 ⟨ Janela: Inicialização 100 ⟩ +≡

```
{
  int context_attribs[] = {GLX_CONTEXT_MAJOR_VERSION_ARB, 3,
      GLX_CONTEXT_MINOR_VERSION_ARB, 3, None};
  glXCreateContextAttribsARBProc glXCreateContextAttribsARB = 0;
  ⟨ Checa suporte à glXGetProcAddressARB 117 ⟩
  glXCreateContextAttribsARB =
      (glXCreateContextAttribsARBProc)glXGetProcAddressARB((const
      GLubyte*) "glXCreateContextAttribsARB");
  context = glXCreateContextAttribsARB(_dpy, *fbConfigs, Λ, GL_TRUE,
      context_attribs);
```

```
    glXMakeCurrent(_dpy, _window, context);
  }
```

¶ Isso criará o contexto se tivermos suporte à função. Mas e se não tivermos? Neste caso, um ponteiro inválido será passado para *glXCreateContextAttribsARB* e obteremos uma falha de segmentação quando tentarmos executá-lo. A API não tem como sabercom certeza se a função existe ou não durante a invocação de *glXGetProcAddressARB*. Neste caso, a função não pode nos avisar se algo der errado fazendo algo como retornar Λ. Portanto, para evitarmos a mensagem antipática de falha de segmentação, teremos que checar antes se temos suporte à esta função ou não. Se não tivermos, temos que cancelar o programa:

117 ⟨ Checa suporte à glXGetProcAddressARB 117 ⟩ ≡

```
  {     /* Primeiro obtemos lista de extensões OpenGL: */
    const char *glxExts = glXQueryExtensionsString(_dpy, screen);
    if (strstr(glxExts, "GLX_ARB_create_context") ≡ Λ) {
      fprintf(stderr, "ERROR: Can't create an OpenGL 3.0 context.\n");
      exit(1);
    }
  }
```

This code is used in chunk 116.

¶ À partir de agora, se tudo deu certo e suportamos todos os pré-requisitos, já criamos a nossa janela e ela está pronta para receber comandos OpenGL. Agora é só na finalização destruirmos o contexto que criamos. Colocamos logo em seguida o código para destruir a janela e encerrar a conexão, já que estas coisas precisam ser feitas nesta ordem:

118 ⟨ Janela: Finalização 118 ⟩ ≡

```
  glXMakeCurrent(_dpy, None, Λ);
  glXDestroyContext(_dpy, context);
  XDestroyWindow(_dpy, _window);
  XCloseDisplay(_dpy);
```

This code is used in chunk 95.

¶

3.2 Definir tamanho do canvas

Agora é hora de definirmos também o espaço na qual poderemos desenhar na tela quando compilamos o programa para a Web. Felizmente, isso é mais fácil que criar uma janela no Xlib. Basta usarmos o suporte que Emscripten tem para as funções SDL. Então adicionamos como cabeçalho da API:

120 ⟨ Cabeçalhos Weaver 45 ⟩ +≡
```
#if W_TARGET ≡ W_WEB
#include "canvas.h"
#endif
```

¶ Agora definimos o nosso cabeçalho do módulo de "canvas":

121 ⟨ project/src/weaver/canvas.h 121 ⟩ ≡
```
#ifndef _canvas_H_
#define _canvas_h_
#ifdef __cplusplus
  extern "C"
  {
#endif
    ⟨ Inclui Cabeçalho de Configuração 42 ⟩
#include "weaver.h"
#include <stdio.h>     /* fprintf */
#include <stdlib.h>     /* exit */
#include <SDL/SDL.h>
            /* SDL_Init, SDL_CreateWindow, SDL_DestroyWindow, */
            /* SDL_Quit */
        void _initialize_canvas(void);
        void _finalize_canvas(void);
    ⟨ Canvas: Declaração 130 ⟩
#ifdef __cplusplus
  }
#endif
#endif
```

¶ E por fim, o nosso `canvas.c` que definirá as funções que criarão nosso espaço de desenho pode ser definido. Como ele é bem mais simples, será inteiramente definido abaixo:

122 ⟨ project/src/weaver/canvas.c 122 ⟩ ≡
```
  ⟨ Inclui Cabeçalho de Configuração 42 ⟩
      extern int make_iso_compilers_happy;
#if W_TARGET ≡ W_WEB
#include "canvas.h"
      static SDL_Surface*window;
```

```
int W_width, W_height, W_x = 0, W_y = 0; ⟨ Canvas: Variáveis 138 ⟩
    void _initialize_canvas(void)
    {
      SDL_Init(SDL_INIT_VIDEO);
      SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 3);
      SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 3);
      window = SDL_SetVideoMode(
#if W_WIDTH > 0
      W_width = W_WIDTH,      /* Largura da janela */
#else
      W_width = 800,      /* Largura da janela */
#endif
#if W_HEIGHT > 0
      W_height = W_HEIGHT,      /* Altura da janela */
#else
      W_height = 600,      /* Altura da janela */
#endif
      0,      /* Bits por pixel, usar o padrão */
      SDL_OPENGL      /* Inicializar o contexto OpenGL */
#if W_WIDTH ≡ 0 ∧ W_HEIGHT ≡ 0
      | SDL_WINDOW_FULLSCREEN
#endif
      );
      if (window ≡ Λ) {
        fprintf(stderr, "ERROR: Could not create window: %s\n",
            SDL_GetError());
        exit(1);
      }
      ⟨ Canvas: Inicialização 140 ⟩
    }
    void _finalize_canvas(void)
    {
      SDL_FreeSurface(window);
    }
    ⟨ Canvas: Definição 131 ⟩
#endif
```

¶ Note que o que estamos chamando de "janela"na verdade é uma superfície SDL. E que não é necessário chamar *SDL_Quit*, tal função seria ignorada se usada.

Por fim, basta agora apenas invocarmos tais funções na inicialização e finalização da API:

123 ⟨ API Weaver: Inicialização 81 ⟩ +≡

```
#if W_TARGET ≡ W_WEB
```

```
    _initialize_canvas( );
#endif
```

124 ¶⟨ API Weaver: Finalização 82 ⟩ +≡

```
#if W_TARGET ≡ W_WEB
    _finalize_canvas( );
#endif
```

¶

3.3 Mudanças no Tamanho e Posição da Janela

Em Xlib, quando uma janela tem o seu tamanho mudado, ela recebe um evento do tipo *ConfigureNotify*. Além dele, também existirão novos eventos se o usuário apertar uma tecla, mover o mouse e assim por diante. Por isso, precisamos adicionar código para tratarmos de eventos no loop principal:

126 ⟨ API Weaver: Loop Principal 47 ⟩ +≡
```
  ⟨ API Weaver: Imediatamente antes de tratar eventos 178 ⟩
#if W_TARGET ≡ W_ELF
  {
    XEvent event;
    while (XPending(_dpy)) {
      XNextEvent(_dpy, &event);
          /* A variável 'event' terá todas as informações do evento */
      ⟨ API Weaver: Trata Evento Xlib 127 ⟩
    }
  }
#endif
#if W_TARGET ≡ W_WEB
  {
    SDL_Event event;
    while (SDL_PollEvent(&event)) {
       /* A variável 'event' terá todas as informações do evento */
    ⟨ API Weaver: Trata Evento SDL 184 ⟩
    }
  }
#endif
```

¶ Por hora definiremos só o tratamento do evento de mudança de tamanho e posição da janela em Xlib. Outros eventos terão seus tratamentos definidos mais tarde, assim como os eventos SDL caso estejamos rodando em um navegador web.

Tudo o que temos que fazer no caso deste evento é atualizar as variáveis globais *W_width*, *W_height*, *W_x* e *W_y*. Nem sempre o evento *ConfigureNotify* significa que a janela mudou de tamanho ou foi movida. Mas mesmo assim, não custa quase nada atualizarmos tais dados. Se eles não mudaram, de qualquer forma este código será inócuo:

127 ⟨ API Weaver: Trata Evento Xlib 127 ⟩ ≡
```
  if (event.type ≡ ConfigureNotify) {
    XConfigureEvent config = event.xconfigure;
    W_x = config.x;
    W_y = config.y;
    W_width = config.width;
    W_height = config.height;
```

```
    continue;
  }
```

See also chunks 176, 177, 198, 199, and 206.
This code is used in chunk 126.

¶ Mas e se nós quisermos mudar o tamanho ou a posição de uma janela diretamente? Para mudar o tamanho, definiremos separadamente o código tanto para o caso de termos uma janela como para o caso de termos um "canvas" web para o jogo. No caso da janela, usamos uma função XLib para isso:

128 ⟨ Janela: Declaração 115 ⟩ +≡

```
  void Wresize_window(int width, int height);
```

129 ¶⟨ Janela: Definição 129 ⟩ ≡

```
  void Wresize_window(int width, int height)
  {
    XResizeWindow(_dpy, _window, width, height);
    W_width = width;
    W_height = height;
    ⟨ Ações após Redimencionar Janela 237 ⟩
  }
```

See also chunks 133, 144, 147, and 153.
This code is used in chunk 95.

¶ No caso de termos um "canvas" web, então usamos SDL para obtermos o mesmo efeito:

130 ⟨ Canvas: Declaração 130 ⟩ ≡

```
  void Wresize_window(int width, int height);
```

See also chunks 134 and 154.
This code is used in chunk 121.

131 ¶⟨ Canvas: Definição 131 ⟩ ≡

```
  void Wresize_window(int width, int height)
  {
    window = SDL_SetVideoMode(width, height, 0,
        /* Bits por pixel, usar o padrão */
    SDL_OPENGL      /* Inicializar o contexto OpenGL */
    );
    W_width = width;
    W_height = height;
    ⟨ Ações após Redimencionar Janela 237 ⟩
  }
```

See also chunks 135 and 155.
This code is used in chunk 122.

¶ Mudar a posição da janela é algo diferente. Isso só faz sentido se realmente tivermos uma janela Xlib, e não um "canvas" web. De qualquer forma, precisaremos definir esta função em ambos os casos. Mas caso estejamos diante de um "canvas", a função de mudar posição deve ser simplesmente ignorada.

132 ⟨ Janela: Declaração 115 ⟩ +≡

```
void Wmove_window(int x, int y);
```

133 ¶⟨ Janela: Definição 129 ⟩ +≡

```
void Wmove_window(int x, int y)
{
  XMoveWindow(_dpy, _window, x, y);
  W_x = x;
  W_y = y;
}
```

134 ¶⟨ Canvas: Declaração 130 ⟩ +≡

```
void Wmove_window(int x, int y);
```

135 ¶⟨ Canvas: Definição 131 ⟩ +≡

```
void Wmove_window(int width, int height)
{
  return;
}
```

¶

3.4 Mudando a resolução da tela

Inicialmente o Servidor X não possuía qualquer recurso para que fosse possível mudar a sua resolução enquanto ele executa, ou coisas como rotacionar a janela raiz. A única forma de obter isso era encerrando o servidor e iniciando-o novamente com nova configuração. Mas programas como jogos podem ter a necessidade de rodar em resolução menor para melhorar o desempenho, mas ao mesmo tempo podem precisar ocupar a tela toda para obter imersão.

Note que isso só faz sentido quando lidamos com uma janela rodando em um gerenciador de janelas. Não no "canvas" de um navegador.

O primeiro problema que temos é que não dá pra mudar a resolução arbitrariamente. Existe apenas um conjunto limitado de resoluções que são realmente possíveis em um dado monitor. Então a primeira coisa que precisamos fazer é descobrir quantos modos são realmente possíveis na tela em que estamos.

Cada modo de funcionamento suportado por uma tela possui três valores distintos: a resolução horizontal, vertical, e a frequência de atualização da tela. A ideia é que nós usemos uma variável *Wnumber_of_modes* para armazenar quantos modos diferentes temos, *Wcurrent_mode* para sabermos qual o modo atual e aloquemos uma estrutura formada por um *array* de triplas de números contendo os dados de cada modo, a qual pode ser acessada por meio de *Wmodes*. Cada um dos modos possíveis terá um número sequencial. E se quisermos passar para outro modo, usaremos uma função que recebe como argumento tal número e facilmente pode checar se está diante de um valor inválido.

137 ⟨ Variáveis de Janela 98 ⟩ +≡
```
unsigned Wnumber_of_modes, Wcurrent_mode;
struct _wmodes {
  int width, height, rate, id;
} *Wmodes;
```

138 ¶⟨ Canvas: Variáveis 138 ⟩ ≡
```
unsigned Wnumber_of_modes, Wcurrent_mode;
struct _wmodes {
  int width, height, rate, id;
} *Wmodes;
```
This code is used in chunk 122.

139 ¶⟨ Cabeçalhos Weaver 45 ⟩ +≡
```
extern unsigned Wnumber_of_modes, Wcurrent_mode;
extern struct _wmodes *Wmodes;
```

¶ Agora cabe à nós inicializarmos isso tudo. Se estamos programando para a Web, nós não podemos mesmo mudar a resolução. Então, o número de modos que temos é sempre um só. E a informação de resolução de tela pode ser

obtida armazenando o retorno de *SDL_GetVideoInfo* em uma estrutura de informação. A taxa de atualização de tela é setada como zero, significando um valor indefinido.

140 ⟨ Canvas: Inicialização 140 ⟩ ≡

```
{
  const SDL_VideoInfo*info = SDL_GetVideoInfo( );
  Wnumber_of_modes = 1;
  Wcurrent_mode = 0;
  Wmodes = (struct _wmodes *) _iWalloc(sizeof(struct _wmodes));
  Wmodes[0].width = info→current_w;
  Wmodes[0].height = info→current_h;
  Wmodes[0].rate = 0;
  Wmodes[0].id = 0;
}
#if W_DEBUG_LEVEL ≥ 3
  fprintf(stderr, "WARNING␣(3):␣Screen␣resolution:␣%dx%d.\n",
      Wmodes[0].width, Wmodes[0].height);
#endif
```

This code is used in chunk 122.

¶ Se não estamos programando para a Web, inicializar tais dados é mais complicado. Nós vamos precisar usar a extensão XRandr. E além disso, como podemos mudar a resolução da nossa tela, é importante memorizarmos os valores iniciais. Usaremos duas variáveis abaixo para fazer isso. A primeira é um ID que representa a resolução e a segunta é a taxa de atualização a tela. Mais abaixo, uma terceira variável armazena a rotação atual da tela. Weaver não permite que rotacionemos a tela, mas mesmo assim tal informação deve ser obtida para quando depois tivermos que restaurar as configurações iniciais.

Por fim, a quarta variável que definimos é uma que irá armazenar as informações das configurações relacionadas à resolução e taxa de atualização da tela.

141 ⟨ Variáveis de Janela 98 ⟩ +≡

```
static int _orig_size_id, _orig_rate;
static Rotation _orig_rotation;
static XRRScreenConfiguration*conf;
```

142 ¶⟨ Janela: Inicialização 100 ⟩ +≡

```
{
  Window root = RootWindow(_dpy, 0);      /* Janela raíz da tela padrão */
  int num_modes, num_rates, i, j, k;
  XRRScreenSize * modes = XRRSizes(_dpy, 0, &num_modes);
  short *rates;      /* Obtendo o número de modos */
```

```
    Wnumber_of_modes = 0;
    for (i = 0; i < num_modes; i++) {
      rates = XRRRates(_dpy, 0, i, &num_rates);
      Wnumber_of_modes += num_rates;
    }
    Wmodes = (struct _wmodes *) _iWalloc(sizeof(struct
        _wmodes) * Wnumber_of_modes);
      /* obtendo o valor original de resolução e taxa de atualização: */
    conf = XRRGetScreenInfo(_dpy, root);
    _orig_rate = XRRConfigCurrentRate(conf);
    _orig_size_id = XRRConfigCurrentConfiguration(conf, &_orig_rotation);
      /* Preenchendo as informações dos modos e descobrindo o ID do atual
       */
    k = 0;
    for (i = 0; i < num_modes; i++) {
      rates = XRRRates(_dpy, 0, i, &num_rates);
      for (j = 0; j < num_rates; j++) {
        Wmodes[k].width = modes[i].width;
        Wmodes[k].height = modes[i].height;
        Wmodes[k].rate = rates[j];
        Wmodes[k].id = i;
        if (i ≡ _orig_size_id ∧ rates[j] ≡ _orig_rate) Wcurrent_mode = k;
        k++;
      }
    }
#if W_DEBUG_LEVEL ≥ 3
    fprintf(stderr, "WARNING␣(3):␣Screen␣resolution:␣%dx%d␣(%dHz).\n",
        Wmodes[Wcurrent_mode].width, Wmodes[Wcurrent_mode].height,
        Wmodes[Wcurrent_mode].rate);
#endif
  }
```

¶ Caso modifiquemos a resolução da tela, antes de fechar o programa, precisamos fazer tudo voltar ao que era antes. E o mesmo se o programa for encerrado devido à uma falha de segmentação, divisão por zero, ou algo assim. Independente do que causar o fim do programa, precisamos chamar a função que definiremos:

143 ⟨ Janela: Declaração 115 ⟩ +≡

```
  void _restore_resolution(void);
```

144 ¶⟨ Janela: Definição 129 ⟩ +≡

```
  void _restore_resolution(void)
  {
    Window root = RootWindow(_dpy, 0);
```

```
    XRRSetScreenConfigAndRate(_dpy, conf, root, _orig_size_id,
        _orig_rotation, _orig_rate, CurrentTime);
    XRRFreeScreenConfigInfo(conf);
  }
```

¶ O primeiro caso no qual chamamos esta função é quando encerramos o programa normalmente. Mas precisamos chamar ela antes de termos fechado a conexão com o servidor X. Por isso colocamos este código de finalização imediatamente antes:

145 ⟨ Janela: Pré-Finalização 145 ⟩ ≡
```
  _restore_resolution( );
```
This code is used in chunk 95.

¶ Criamos também uma função *restore_and_quit*, que será a que será chamada caso recebamos um sinal fatal que encerre abruptamente nosso programa:

146 ⟨ Janela: Declaração 115 ⟩ +≡
```
  void _restore_and_quit(int signal, siginfo_t * si, void *arg);
```

147 ¶⟨ Janela: Definição 129 ⟩ +≡
```
  void _restore_and_quit(int signal, siginfo_t * si, void *arg)
  {
    fprintf(stderr, "ERROR: Received signal %d (%p %p).\n", signal, (void
        *) si, (void *) arg);
    Wexit( );
    exit(1);
  }
```

¶ E por fim, trataremos agora dos sinais. Primeiro precisamos salvar as funções responsáveis por tratar os sinais fatais ao mesmo tempo em que as redefinimos. Para isso, precisaremos de um vetor:

148 ⟨ Cabeçalhos Weaver 45 ⟩ +≡
```
#if W_TARGET ≡ W_ELF
#include <signal.h>
#endif
```

149 ¶⟨ API Weaver: Definições 149 ⟩ ≡
```
#if W_TARGET ≡ W_ELF
  static struct sigaction *actions[12];
#endif
```
See also chunks 161, 164, 169, 173, 175, 182, 189, 191, 193, 211, 216, 227, 234, 242, 246, 249, 255, 259, 261, 275, 302, and 325.
This code is used in chunk 46.

150 ¶⟨ API Weaver: Inicialização 81 ⟩ +≡
```
#if W_TARGET ≡ W_ELF
  {
    struct sigaction sa;
    memset(&sa, 0, sizeof(struct sigaction));
    sigemptyset(&sa.sa_mask);
    sa.sa_sigaction = _restore_and_quit;
    sa.sa_flags = SA_SIGINFO;
    sigaction(SIGHUP, &sa, actions[0]);
    sigaction(SIGINT, &sa, actions[1]);
    sigaction(SIGQUIT, &sa, actions[2]);
    sigaction(SIGILL, &sa, actions[3]);
    sigaction(SIGABRT, &sa, actions[4]);
    sigaction(SIGFPE, &sa, actions[5]);
    sigaction(SIGSEGV, &sa, actions[6]);
    sigaction(SIGPIPE, &sa, actions[7]);
    sigaction(SIGALRM, &sa, actions[8]);
    sigaction(SIGTERM, &sa, actions[9]);
    sigaction(SIGUSR1, &sa, actions[10]);
    sigaction(SIGUSR2, &sa, actions[11]);
  }
#endif
```

¶ O único caso no qual não seremos capazes de restaurar a resolução é quando recebermos um SIGKILL. Não há muito a fazer com relação à isso. Entretanto, um sinal desta magnitude só pode ser gerado por um usuário, nunca será a reação do Sistema Operacional à uma ação do programa. Então, teremos que assumir que caso isso aconteça, o usuário sabe o que está fazendo e saberá retornar a resolução ao seu estado atual.

Precisamos agora durante a finalização da API restaurar os tratamentos originais dos sinais. Para isso, usamos o seguinte código:

151 ⟨ Restaura os Sinais do Programa (SIGINT, SIGTERM, etc) 151 ⟩ ≡
```
  sigaction(SIGHUP, actions[0], Λ);
  sigaction(SIGINT, actions[1], Λ);
  sigaction(SIGQUIT, actions[2], Λ);
  sigaction(SIGILL, actions[3], Λ);
  sigaction(SIGABRT, actions[4], Λ);
  sigaction(SIGFPE, actions[5], Λ);
  sigaction(SIGSEGV, actions[6], Λ);
  sigaction(SIGPIPE, actions[7], Λ);
  sigaction(SIGALRM, actions[8], Λ);
  sigaction(SIGTERM, actions[9], Λ);
  sigaction(SIGUSR1, actions[10], Λ);
  sigaction(SIGUSR2, actions[11], Λ);
```
This code is used in chunk 97.

¶ Uma vez que tenhamos garantido que a resolução voltará ao normal após o programa se encerrar, podemos fornecer então uma função responsável por mudar a resolução e modo da tela. Esta função deverá receber como argumento um número inteiro. Se este número for menor que zero ou maior ou igual ao número total de modos que temos em nossa tela, a função não fará nada e retornará zero. Caso contrário, ela mudará o modo da tela para o representado pelo índice passado como argumento em *Wmodes*. Além disso, ela mudará o tamanho da janela para o da nova resolução, deixando o jogo em tela cheia, e retornará 1:

152 ⟨ Janela: Declaração 115 ⟩ +≡
```
int Wfullscreen_mode(unsigned int mode);
```

153 ¶⟨ Janela: Definição 129 ⟩ +≡
```
int Wfullscreen_mode(unsigned int mode)
{
  if (mode ≥ Wnumber_of_modes) return 0;
  else {
    Window root = RootWindow(_dpy, 0);
    Wmove_window(0, 0);
    Wresize_window(Wmodes[mode].width, Wmodes[mode].height);
    XRRSetScreenConfigAndRate(_dpy, conf, root, Wmodes[mode].id,
        _orig_rotation, Wmodes[mode].rate, CurrentTime);
    return 1;
  }
}
```

¶ Também teremos que definir a mesma função caso estejamos fazendo um jogo para a Web. Mas neste caso, a função não fará sentido e sempre retornará 0:

154 ⟨ Canvas: Declaração 130 ⟩ +≡
```
int Wfullscreen_mode(int mode);
```

155 ¶⟨ Canvas: Definição 131 ⟩ +≡
```
int Wfullscreen_mode(int mode)
{
  return 0;
}
```

¶

3.5 Configurações Básicas OpenGL

A única configuração que temos no momento é a cor de fundo de nossa janela, a qual será exibida na ausência de qualquer coisa a ser mostrada:

157 ⟨ API Weaver: Inicialização 81 ⟩ +≡ /∗ COm que cor limpamos a tela: ∗/
glClearColor(`W_DEFAULT_COLOR`, 1.0 F);
/∗ Ativamos o buffer de profundidade: ∗/
glEnable(`GL_DEPTH_TEST`);

158 ¶⟨ API Weaver: Loop Principal 47 ⟩ +≡
glClear(`GL_COLOR_BUFFER_BIT`);

¶

Capítulo 4

Teclado e Mouse

Uma vez que tenhamos uma janela, podemos começar a acompanhar os eventos associados à ela. Um usuário pode apertar qualquer botão no seu teclado ou mouse e isso gerará um evento. Devemos tratar tais eventos no mesmo local em que já estamos tratando coisas como o mover e o mudar tamanho da janela (algo que também é um evento). Mas devemos criar uma interface mais simples para que um usuário possa acompanhar quando certas teclas são pressionadas, e por quanto tempo elas estão sendo pressionadas.

Nossa proposta é que exista um vetor de inteiros chamado *Wkeyboard*, por exemplo, e que cada posição dele represente uma tecla diferente. Se o valor dentro de uma posição do vetor é 0, então tal tecla não está sendo pressionada. Caso o seu valor seja um número positivo, então a tecla está sendo pressionada e o número representa por quantos milissegundos a tecla vem sendo pressionada. Caso o valor seja um número negativo, significa que a tecla acabou de ser solta e o inverso deste número representa por quantos milissegundos a tecla ficou pressionada.

Acompanhar o tempo no qual uma tecla é pressionada é tão importante quanto saber se ela está sendo pressionada ou não. Por meio do tempo, podemos ser capazes de programar personagens que pulam mais alto ou mais baixo, dependendo do quanto um jogador apertou uma tecla, ou fazer com que jogadores possam escolher entre dar um soco rápido, mas fraco ou devagar, mas forte em outros tipos de jogo. Tudo depende da intensidade com a qual eles pressionam os botões.

Entretanto, tanto o Xlib como SDL funcionam reportando apenas o momento no qual uma tecla é pressionada e o momento na qual ela é solta. Então, em cada iteração, precisamos memorizar quais teclas estão sendo pressionadas. Se duas pessoas estiverem compartilhando um mesmo teclado, teoricamente, o número máximo de teclas que podem ser pressionadas é 20 (se cada dedo da mão de cada uma delas estiver sobre uma tecla). Então, vamos usar um vetor de 20 posições para armazenar o número de cada tecla sendo pressionada. Isso é apenas para podermos atualizar em cada iteração do loop principal o tempo em que cada tecla é pressionada. Se hipoteticamente mais de 20 teclas forem pressionadas, o

fato de perdermos uma delas não é algo muito grave e não deve causar qualquer problema.

Até agora estamos falando do teclado, mas o mesmo pode ser implementado nos botões do mouse. Mas no caso do mouse, além dos botões, temos o seu movimento. Então será importante armazenarmos a sua posição (x, y), mas também um vetor representando o seu deslocamento. Tal vetor deve considerar como se a posição atual do ponteiro do mouse fosse a $(0, 0)$ e deve conter qual a sua posição no próximo segundo caso o seu deslocamento continue constante na mesma direção e sentido em que vem sendo desde a última iteração. Desta forma, tal vetor também será útil para verificar se o mouse está em movimento ou não. E saber a intensidade e direção do movimento do mouse pode permitir interações mais ricas com o usuário.

4.1 Preparando o Loop Principal: Medindo a Passagem de Tempo

Conforme exposto na introdução, toda vez que estivermos em um loop principal do jogo, a função *weaver_rest* deve ser invocada uma vez a cada iteração. Devemos então manter algumas variáveis controlando a passagem do tempo, e tais variáveis devem ser atualizadas sempre dentro destas funções.

No caso, vamos precisar inicialmente de uma variável para armazenar o tempo da iteração atual e a da iteração anterior, em escala de microssegundos:

161 ⟨ API Weaver: Definições 149 ⟩ +≡
```
static struct timeval _last_time, _current_time;
```

¶ É importante que ambos os valores sejam inicializados como zero, caso contrário, valores estranhos podem ser derivados caso usemos os valores antes de serem corretamente inicializados na primeira iteração de um loop principal:

162 ⟨ API Weaver: Inicialização 81 ⟩ +≡
```
_last_time.tv_sec = 0;
_last_time.tv_sec = 0;
_current_time.tv_sec = 0;
_current_time.tv_usec = 0;
```

¶ No loop principal em si, o valor que temos como o do tempo atual deve ser passado para o tempo anterior, e em seguida deve ser sobrescrito por um novo tempo atual:

163 ⟨ API Weaver: Loop Principal 47 ⟩ +≡
```
{
  _last_time.tv_sec = _current_time.tv_sec;
  _last_time.tv_usec = _current_time.tv_usec;
  gettimeofday(&_current_time, Λ);
}
```

¶ Estas medidas de tempo serão realmente usadas para atualizar duas variáveis a cada iteração. A primeira será uma variável interna e armazenará quantos milissegundos se passaram entre uma iteração e outra. A segunda será uma variável global que poderá ser consultada por usuários e conterá à quantos frames por segundo o jogo está rodando:

164 ⟨ API Weaver: Definições 149 ⟩ +≡
```
static int _elapsed_milisseconds;
int Wfps;
```

165 ¶⟨ Cabeçalhos Weaver 45 ⟩ +≡
```
extern int Wfps;
```

¶ Naturalmente, tais valores também precisam ser inicializados para prevenir que contenham números absurdos na primeira iteração:

166 ⟨ API Weaver: Inicialização 81 ⟩ +≡
```
{
  _elapsed_milisseconds = 0;
  Wfps = 0;
}
```

¶ E em cada iteração do loop principal, atualizamos os valores. Lembrando que realizar a subtração de dois **struct** *timeval* pode ser um pouco chato, mas o próprio manual da biblioteca C GNU demonstra como fazer:

167 ⟨ API Weaver: Loop Principal 47 ⟩ +≡
```
{
  _elapsed_milisseconds = (_current_time.tv_sec − _last_time.tv_sec) ∗ 1000;
  _elapsed_milisseconds += (_current_time.tv_usec − _last_time.tv_usec)/1000;
  if (_elapsed_milisseconds > 0) Wfps = 1000/_elapsed_milisseconds;
  else Wfps = 0;
}
```

¶

4.2 O Teclado

Como mencionado, para o teclado, precisaremos de uma variável local ao arquivo que armazenará as teclas que já estão sendo pressionadas neste momento e uma variável global que será um vetor de números representando a quanto tempo cada tecla é pressionada. Adicionalmente, também precisamos tomar nota das teclas que acabaram de ser soltas para que na iteração seguinte possamos zerar os seus valores no vetor do teclado.

Mas a primeira questão que temos a responder é que tamanho deve ter tal vetor? E como associar cada posição à uma tecla?

Um teclado típico tem entre 80 e 100 teclas diferentes. Entretanto, diferentes teclados representam em cada uma destas teclas diferentes símbolos e caracteres. Alguns teclados possuem "Ç", outros possuem o símbolo do Euro, e outros podem possuir símbolos bem mais exóticos. Há também teclas modificadoras que transformam determinadas teclas em outras. O Xlib reconhece diferentes teclas associando à elas um número chamado de **KeySym**, que são inteiros de 29 bits.

Entretanto, não podemos criar um vetor de 2^{29} números para representar se uma das diferentes teclas possíveis está pressionada. Se cada inteiro tiver 4 bytes, vamos precisar de 2GB de memória para conter tal vetor. Por isso, precisamos nos ater à uma quantidade menor de símbolos.

A vasta maioria das teclas possíveis é representada por números entre 0 e 0xffff. Isso inclui até mesmo caracteres em japonês, "Ç", todas as teclas do tipo Shift, Esc, Caps Lock, Ctrl e o "N" com um til do espanhol. Mas algumas coisas ficam de fora, como cirílico, símbolos árabes, vietnamitas e símbolos matemáticos especiais. Contudo, isso não será algo grave, pois podemos fornecer uma função capaz de redefinir alguns destes símbolos para valores dentro de tal intervalo. O que significa que vamos precisar também de espaço em memória para armazenar tais traduções. Um número de 100 delas pode ser estabelecido como máximo, pois a maioria dos teclados tem menos teclas que isso.

Note que este é um problema do XLib. O SDL de qualquer forma já se atém somente à 16 bytes para representar suas teclas. Então, podemos ignorar com segurança tais traduções quando estivermos programando para a Web.

Sabendo disso, o nosso vetor de teclas e vetor de traduções pode ser declarado, bem como o vetor de teclas pressionadas. Vamos também já deixar declarado um vetor idêntico aos de teclas pressionadas e soltas, mas para os botões do teclado:

169 ⟨ API Weaver: Definições 149 ⟩ +≡

```
  int Wkeyboard[#ffff];
#if W_TARGET ≡ W_ELF
  static struct _k_translate {
    unsigned original_symbol, new_symbol;
  } _key_translate[100];
#endif
  static unsigned _pressed_keys[20];
  static unsigned _released_keys[20];
```

```
static unsigned _pressed_buttons[5];
static unsigned _released_buttons[5];
```

170 ¶⟨ Cabeçalhos Weaver 45 ⟩ +≡

```
extern int Wkeyboard[^ffff];
```

¶ A inicialização de tais valores consiste em deixar todos contendo zero como valor:

171 ⟨ API Weaver: Inicialização 81 ⟩ +≡

```
{
  int i;
  for (i = 0; i < ^ffff; i++) Wkeyboard[i] = 0;
#if W_TARGET ≡ W_ELF
  for (i = 0; i < 100; i++) {
    _key_translate[i].original_symbol = 0;
    _key_translate[i].new_symbol = 0;
  }
#endif
  for (i = 0; i < 20; i++) {
    _pressed_keys[i] = 0;
    _released_keys[i] = 0;
  }
}
```

¶ Inicializar tais vetores para o valor zero funciona porque nem o SDL e nem o XLib associa qualquer tecla ao número zero. De fato, o XLib ignora os primeiros 31 valores e o SDL ignora os primeiros 7. Desta forma, podemos usar tais espaços com segurança para representar conjuntos de teclas ao invés de uma tecla individual. Por exemplo, podemos associar a posição 1 como sendo o de todas as teclas. Qualquer tecla pressionada faz com que ativemos o seu valor. Outra posição pode ser associada ao Shift, que faria com que fosse ativada toda vez que o Shift esquerdo ou direito fosse pressionado. O mesmo para o Ctrl e Alt. Já o valor zero deve continuar sem uso para que possamos reservá-lo para valores inicializados, mas vazios ou indefinidos.

172 ⟨ Cabeçalhos Weaver 45 ⟩ +≡

```
#define W_SHIFT 2
#define W_CTRL 3
#define W_ALT 4
#define W_ANY 6
```

¶ A consulta de um vetor de traduções consiste em percorrermos ele verificando se um determinado símbolo existe nele. Se o encontrarmos, retornamos a sua tradução. Caso contrário, retornamos seu valor inicial:

173 ⟨ API Weaver: Definições 149 ⟩ +≡

```
#if W_TARGET ≡ W_ELF
  static unsigned _translate_key(unsigned symbol)
  {
    int i;
    for (i = 0; i < 100; i++) {
      if (_key_translate[i].original_symbol ≡ 0) return symbol % ^ffff;
      if (_key_translate[i].original_symbol ≡ symbol)
        return _key_translate[i].new_symbol % ^ffff;
    }
    return symbol % ^ffff;
  }
#endif
```

¶ Agora respectivamente a tarefa de adicionar uma nova tradução de tecla e a tarefa de limpar todas as traduções existentes. O que pode dar errado aí é que pode não haver espaço para novas traduções quando vamos adicionar mais uma. Neste caso, a função sinaliza isso retornando 0 ao invés de 1.

174 ⟨ Cabeçalhos Weaver 45 ⟩ +≡

```
  int Wkey_translate(unsigned old_value, unsigned new_value);
  void Werase_key_translations(void);
```

175 ¶⟨ API Weaver: Definições 149 ⟩ +≡

```
  int Wkey_translate(unsigned old_value, unsigned new_value)
  {
#if W_TARGET ≡ W_ELF
    int i;
    for (i = 0; i < 100; i++) {
      if (_key_translate[i].original_symbol ≡
            0 ∨ _key_translate[i].original_symbol ≡ old_value)
        {
        _key_translate[i].original_symbol = old_value;
        _key_translate[i].new_symbol = new_value;
        return 1;
      }
    }
#endif
    return 0;
  }
```

```
  void Werase_key_translations(void)
  {
#if W_TARGET ≡ W_ELF
    int i;
    for (i = 0; i < 100; i++) {
      _key_translate[i].original_symbol = 0;
      _key_translate[i].new_symbol = 0;
    }
#endif
  }
```

¶ Uma vez que tenhamos preparado as traduções, podemos enfim ir até o loop principal e acompanhar o surgimento de eventos para saber quando o usuário pressiona ou solta uma tecla. No caso de estarmos usando XLib e uma tecla é pressionada, o código abaixo é executado. A coisa mais críptica abaixo é o suo da função *XkbKeycodeToKeysym*. Mas basicamente o que esta função faz é traduzir o valor da variável *event.xkey.keycode* de uma representação inicial, que representa a posição da tecla em um teclado para o símbolo específico associado àquela tecla, algo que muda em diferentes teclados.

176 ⟨ API Weaver: Trata Evento Xlib 127 ⟩ +≡

```
  if (event.type ≡ KeyPress) {
    unsigned int code = _translate_key(XkbKeycodeToKeysym(_dpy,
        event.xkey.keycode, 0, 0));
    int i;      /* Adiciona na lista de teclas pressionadas */
    for (i = 0; i < 20; i++) {
      if (_pressed_keys[i] ≡ 0 ∨ _pressed_keys[i] ≡ code) {
        _pressed_keys[i] = code;
        break;
      }
    }     /* Atualiza vetor de teclado se a tecla não estava sendo pressionada.
             Algumas vezes este evento é gerado repetidas vezes quando
             apertamos uma tecla por muito tempo. Então só devemos atribuir 1
             à posição do vetor se realmente a tecla não estava sendo pressionada
             antes: */
    if (Wkeyboard[code] ≡ 0) Wkeyboard[code] = 1;
    else if (Wkeyboard[code] < 0) Wkeyboard[code] *= −1;
    continue;
  }
```

¶ Já se uma tecla é solta, precisamos removê-la da lista de teclas pressionadas e adicioná-la na lista de teclas que acabaram de ser soltas:

177 ⟨ API Weaver: Trata Evento Xlib 127 ⟩ +≡

```
if (event.type ≡ KeyRelease) {
  unsigned int code = _translate_key(XkbKeycodeToKeysym(_dpy,
      event.xkey.keycode, 0, 0));
  int i;      /* Remove da lista de teclas pressionadas */
  for (i = 0; i < 20; i++) {
    if (_pressed_keys[i] ≡ code) {
      _pressed_keys[i] = 0;
      break;
    }
  }
  for ( ; i < 19; i++) {
    _pressed_keys[i] = _pressed_keys[i + 1];
  }
  _pressed_keys[19] = 0;      /* Adiciona na lista de teclas soltas: */
  for (i = 0; i < 20; i++) {
    if (_released_keys[i] ≡ 0 ∨ _released_keys[i] ≡ code) {
      _released_keys[i] = code;
      break;
    }
  }     /* Atualiza vetor de teclado */
  Wkeyboard[code] *= −1;
  continue;
}
```

¶ Mas e quando esvaziamos o vetor de teclas soltas? E quando incrementamos o valor de cada posição em *Wkeyboard* caso uma tecla esteja sendo pressionada? Isso precisa ser feito antes de checarmos os eventos de entrada para que desta forma consigamos manter em 1 o valor de uma tecla que acabou de ser pressionada neste última iteração, e só depois seu valor vá sendo atualizada para outros números. Por isso o seguinte código deve ser posicionado antes do tratamento de eventos:

178 ⟨ API Weaver: Imediatamente antes de tratar eventos 178 ⟩ ≡

```
{
  int i, key;      /* Limpar o vetor de teclas soltas e zerar seus valores no
      vetor de teclado: */
  for (i = 0; i < 20; i++) {
    key = _released_keys[i];      /* Se a tecla está com um valor positivo,
        isso significa que os eventos de soltar a tecla e apertar ela de
        novo foram gerados juntos. Isso geralmente acontece quando um
        usuário pressiona uma tecla por muito tempo. Depois de algum
        tempo, o servidor passa a interpretar isso como se o usuário
        estivesse apertando e soltando a tecla sem parar. Isso é útil em
        editores de texto quando você segura uma tecla e a letra que ela
```

```
            representa começa a ser inserida sem parar após um tempo. Mas
            aqui isso deixa o ato de medir o tempo cheio de detalhes incômodos.
            Aqui temos que remover da lista de teclas soltas esta tecla, que
            provavelmente não foi solta de verdade: */
        while (Wkeyboard[key] > 0) {
          int j;
          for (j = i; j < 19; j++) {
            _released_keys[j] = _released_keys[j + 1];
          }
          _released_keys[19] = 0;
          key = _released_keys[i];
        }
        if (key ≡ 0) break;
        if (key ≡ W_LEFT_CTRL ∨ key ≡ W_RIGHT_CTRL)
          Wkeyboard[W_CTRL] = 0;
        else if (key ≡ W_LEFT_SHIFT ∨ key ≡ W_RIGHT_SHIFT)
          Wkeyboard[W_SHIFT] = 0;
        else if (key ≡ W_LEFT_ALT ∨ key ≡ W_RIGHT_ALT)
          Wkeyboard[W_ALT] = 0;
        Wkeyboard[key] = 0;
        _released_keys[i] = 0;
      }     /* Para teclas pressionadas, incrementar o tempo em que elas estão
            pressionadas: */
      for (i = 0; i < 20; i++) {
        key = _pressed_keys[i];
        if (key ≡ 0) break;
        if (key ≡ W_LEFT_CTRL ∨ key ≡ W_RIGHT_CTRL)
          Wkeyboard[W_CTRL] += _elapsed_milisseconds;
        else if (key ≡ W_LEFT_SHIFT ∨ key ≡ W_RIGHT_SHIFT)
          Wkeyboard[W_SHIFT] += _elapsed_milisseconds;
        else if (key ≡ W_LEFT_ALT ∨ key ≡ W_RIGHT_ALT)
          Wkeyboard[W_ALT] += _elapsed_milisseconds;
        Wkeyboard[key] += _elapsed_milisseconds;
      }
    }
```

See also chunks 197 and 205.

This code is used in chunk 126.

¶ Por fim, preenchemos a posição *Wkeyboard*[W_ANY] depois de tratarmos todos os eventos:

179 ⟨ API Weaver: Loop Principal 47 ⟩ +≡

```
Wkeyboard[W_ANY] = (_pressed_keys[0] ≠ 0);
```

¶ Isso conclui o código que precisamos para o teclado no Xlib. Mas ainda não acabou. Precisamos de macros para representar as diferentes teclas de modo que um usuário possa consultar se uma tecla está pressionada sem saber o código da tecla no Xlib:0

180 ⟨ Cabeçalhos Weaver 45 ⟩ +≡

```
#if W_TARGET ≡ W_ELF
#define W_UP XK_Up
#define W_RIGHT XK_Right
#define W_DOWN XK_Down
#define W_LEFT XK_Left
#define W_PLUS XK_KP_Add
#define W_MINUS XK_KP_Subtract
#define W_ESC XK_Escape
#define W_A XK_a
#define W_S XK_s
#define W_D XK_d
#define W_W XK_w
#define W_ENTER XK_Return
#define W_LEFT_CTRL XK_Control_L
#define W_RIGHT_CTRL XK_Control_R
#define W_F1 XK_F1
#define W_F2 XK_F2
#define W_F3 XK_F3
#define W_F4 XK_F4
#define W_F5 XK_F5
#define W_F6 XK_F6
#define W_F7 XK_F7
#define W_F8 XK_F8
#define W_F9 XK_F9
#define W_F10 XK_F10
#define W_F11 XK_F11
#define W_F12 XK_F12
#define W_BACKSPACE XK_BackSpace
#define W_TAB XK_Tab
#define W_PAUSE XK_Pause
#define W_DELETE XK_Delete
#define W_SCROLL_LOCK XK_Scroll_Lock
#define W_HOME XK_Home
#define W_PAGE_UP XK_Page_Up
#define W_PAGE_DOWN XK_Page_Down
#define W_END XK_End
#define W_INSERT XK_Insert
#define W_NUM_LOCK XK_Num_Lock
#define W_ZERO XK_KP_0
#define W_ONE XK_KP_1
```

```
#define W_TWO  XK_KP_2
#define W_THREE  XK_KP_3
#define W_FOUR  XK_KP_4
#define W_FIVE  XK_KP_5
#define W_SIX  XK_KP_6
#define W_SEVEN  XK_KP_7
#define W_EIGHT  XK_KP_8
#define W_NINE  XK_KP_9
#define W_LEFT_SHIFT  XK_Shift_L
#define W_RIGHT_SHIFT  XK_Shift_R
#define W_CAPS_LOCK  XK_Caps_Lock
#define W_LEFT_ALT  XK_Alt_L
#define W_RIGHT_ALT  XK_Alt_R
#define W_Q  XK_q
#define W_E  XK_e
#define W_R  XK_r
#define W_T  XK_t
#define W_Y  XK_y
#define W_U  XK_u
#define W_I  XK_i
#define W_O  XK_o
#define W_P  XK_p
#define W_F  XK_f
#define W_G  XK_g
#define W_H  XK_h
#define W_J  XK_j
#define W_K  XK_k
#define W_L  XK_l
#define W_Z  XK_z
#define W_X  XK_x
#define W_C  XK_c
#define W_V  XK_v
#define W_B  XK_b
#define W_N  XK_n
#define W_M  XK_m
#endif
```

¶ A última coisa que resta para termos uma API funcional para lidar com teclados é uma função para limpar o vetor de teclados e a lista de teclas soltas e pressionadas. Desta forma, podemos nos livrar de teclas pendentes quando saímos de um loop principal para outro, além de termos uma forma de fazer com que o programa possa descartar teclas pressionadas em momentos dos quais não era interessante levá-las em conta.

Mas não vamos querer fazer isso só com o teclado, mas com todas as formas de entrada possíveis. Portanto, vamos deixar este trecho de código com uma

marcação para inserirmos mais coisas depois:

181 ⟨ Cabeçalhos Weaver 45 ⟩ +≡
```
void Wflush_input(void);
```

182 ¶⟨ API Weaver: Definições 149 ⟩ +≡
```
void Wflush_input(void)
{
  {     /* Limpa informação do teclado */
    int i, key;
    for (i = 0; i < 20; i++) {
      key = _pressed_keys[i];
      _pressed_keys[i] = 0;
      Wkeyboard[key] = 0;
      key = _released_keys[i];
      _released_keys[i] = 0;
      Wkeyboard[key] = 0;
    }
  }
  ⟨ Limpar Entrada 208 ⟩
}
```

¶ Quase tudo o que foi definido aqui aplica-se tanto para o Xlib rodando em um programa nativo para Linux como em um programa SDL compilado para a Web. A única exceção é o tratamento de eventos, que é feita usando funções diferentes nas duas bibliotecas.

É preciso inserir o cabeçalho SDL neste caso:

183 ⟨ Cabeçalhos Weaver 45 ⟩ +≡
```
#if W_TARGET ≡ W_WEB
#include <SDL/SDL.h>
#endif
```

¶ E tratamos o evento de uma tecla ser pressionada exatamente da mesma forma, mas respeitando as diferenças das bibliotecas em como acessar cada informação:

184 ⟨ API Weaver: Trata Evento SDL 184 ⟩ ≡
```
if (event.type ≡ SDL_KEYDOWN) {
  unsigned int code = event.key.keysym.sym;
  int i;      /* Adiciona na lista de teclas pressionadas */
```

```
  for (i = 0; i < 20; i++) {
    if (_pressed_keys[i] ≡ 0 ∨ _pressed_keys[i] ≡ code) {
      _pressed_keys[i] = code;
      break;
    }
  }     /* Atualiza vetor de teclado se a tecla não estava sendo pressionada.
           Algumas vezes este evento é gerado repetidas vezes quando
           apertamos uma tecla por muito tempo. Então só devemos atribuir 1
           à posição do vetor se realmente a tecla não estava sendo pressionada
           antes. */
  if (Wkeyboard[code] ≡ 0) Wkeyboard[code] = 1;
  else if (Wkeyboard[code] < 0) Wkeyboard[code] *= −1;
  continue;
}
```

See also chunks 185, 200, 201, and 207.

This code is used in chunk 126.

¶ Por fim, o evento da tecla sendo solta:

185 ⟨ API Weaver: Trata Evento SDL 184 ⟩ +≡

```
if (event.type ≡ SDL_KEYUP) {
  unsigned int code = event.key.keysym.sym;
  int i;      /* Remove da lista de teclas pressionadas */
  for (i = 0; i < 20; i++) {
    if (_pressed_keys[i] ≡ code) {
      _pressed_keys[i] = 0;
      break;
    }
  }
  for ( ; i < 19; i++) {
    _pressed_keys[i] = _pressed_keys[i + 1];
  }
  _pressed_keys[19] = 0;      /* Adiciona na lista de teclas soltas: */
  for (i = 0; i < 20; i++) {
    if (_released_keys[i] ≡ 0 ∨ _released_keys[i] ≡ code) {
      _released_keys[i] = code;
      break;
    }
  }     /* Atualiza vetor de teclado */
  Wkeyboard[code] *= −1;
  continue;
}
```

¶ E por fim, a posição das teclas para quando usamos SDL no vetor de teclado será diferente e correspondente aos valores usados pelo SDL:

186 ⟨ Cabeçalhos Weaver 45 ⟩ +≡
```
#if W_TARGET ≡ W_WEB
#define W_UP SDLK_UP
#define W_RIGHT SDLK_RIGHT
#define W_DOWN SDLK_DOWN
#define W_LEFT SDLK_LEFT
#define W_PLUS SDLK_PLUS
#define W_MINUS SDLK_MINUS
#define W_ESC SDLK_ESCAPE
#define W_A SDLK_a
#define W_S SDLK_s
#define W_D SDLK_d
#define W_W SDLK_w
#define W_ENTER SDLK_RETURN
#define W_LEFT_CTRL SDLK_LCTRL
#define W_RIGHT_CTRL SDLK_RCTRL
#define W_F1 SDLK_F1
#define W_F2 SDLK_F2
#define W_F3 SDLK_F3
#define W_F4 SDLK_F4
#define W_F5 SDLK_F5
#define W_F6 SDLK_F6
#define W_F7 SDLK_F7
#define W_F8 SDLK_F8
#define W_F9 SDLK_F9
#define W_F10 SDLK_F10
#define W_F11 SDLK_F11
#define W_F12 SDLK_F12
#define W_BACKSPACE SDLK_BACKSPACE
#define W_TAB SDLK_TAB
#define W_PAUSE SDLK_PAUSE
#define W_DELETE SDLK_DELETE
#define W_SCROLL_LOCK SDLK_SCROLLOCK
#define W_HOME SDLK_HOME
#define W_PAGE_UP SDLK_PAGEUP
#define W_PAGE_DOWN SDLK_PAGEDOWN
#define W_END SDLK_END
#define W_INSERT SDLK_INSERT
#define W_NUM_LOCK SDLK_NUMLOCK
#define W_ZERO SDLK_0
#define W_ONE SDLK_1
#define W_TWO SDLK_2
#define W_THREE SDLK_3
#define W_FOUR SDLK_4
#define W_FIVE SDLK_5
#define W_SIX SDLK_6
```

```
#define W_SEVEN  SDLK_7
#define W_EIGHT  SDLK_8
#define W_NINE  SDLK_9
#define W_LEFT_SHIFT  SDLK_LSHIFT
#define W_RIGHT_SHIFT  SDLK_RSHIFT
#define W_CAPS_LOCK  SDLK_CAPSLOCK
#define W_LEFT_ALT  SDLK_LALT
#define W_RIGHT_ALT  SDLK_RALT
#define W_Q  SDLK_q
#define W_E  SDLK_e
#define W_R  SDLK_r
#define W_T  SDLK_t
#define W_Y  SDLK_y
#define W_U  SDLK_u
#define W_I  SDLK_i
#define W_O  SDLK_o
#define W_P  SDLK_p
#define W_F  SDLK_f
#define W_G  SDLK_g
#define W_H  SDLK_h
#define W_J  SDLK_j
#define W_K  SDLK_k
#define W_L  SDLK_l
#define W_Z  SDLK_z
#define W_X  SDLK_x
#define W_C  SDLK_c
#define W_V  SDLK_v
#define W_B  SDLK_b
#define W_N  SDLK_n
#define W_M  SDLK_m
#endif
```

¶

4.3 Invocando o loop principal

Um jogo ode ter vários loops principais. Um para a animação de abertura. Outro para a tela de título onde escolhe-se o modo do jogo. Um para cada fase ou cenário que pode-se visitar. Pode haver outro para cada "fase especial" ou mesmo para cada batalha em um jogo de RPG.

Em cada um dos loops principais, precisamos rodar possivelmente milhares de iterações. E em cada uma delas precisamos fazer algumas coisas em comum. Imediatamente antes do loop precisamos limpar todos os valores prévios armazenados no vetor de teclado. E depois em cada iteração precisamos rodar *weaver_rest* para obtermos os eventos de entrada, atualizarmos várias variáveis e poder desenhar na tela.

O problema é que este tipo de coisa depende do ambiente de execução em que estamos. Por exemplo, se estamos executando um programa Linux, o seguinte loop principal seria válido:

```
while(1){
handle_input();
handle_objects();
weaver_rest(10);
}
```

Além disso poderíamos criar uma condição explícita para sairmos do loop e entrarmos em outra logo em seguida. Mas infelizmente se estamos executando em um navegador de Internet após termos o código compilado para Javascript, isso não é possível. Um loop infinito geraria um loop no código Javascript e isso faria com que a função Javascript nunca termine. Isso faria com que o navegador congelasse dentro do loop e se oferecesse para matar o script problemático, sem poder fazer coisas como desenhar na tela. Talvez o navegador não conseguisse nem mesmo detectar teclas pressionadas pelo jogador.

Portanto, não podemos deixar que o loop principal seja um loop neste caso. Ele precisa ser uma função que executa de tempos em tempos. Infelizmente, a API Emscripten requer que tal função não retorne nada e nem receba argumentos. Sendo assim, toda informação necessária para o loop principal deve estar em variáveis globais. É algo ruim, mas podemos minimizar os danos disso usando a palavra-chave **static** para limitar o escopo de nossas variáveis em cada módulo.

O que queremos então é que um programa Weaver possa ter então a seguinte forma:

```
void main_loop(void){
// ...
weaver_rest(10);
}

int main(int argc, char **argv){
awake_the_weaver();
```

```
// Executa \PB{\\{main\_loop}} como o loop principal
Wloop(main_loop);

weaver_rest();
}
```

A função *Wloop* então executa a função que recebe como argumento em um loop infinito. E esta função deve ser definida de modo diferente dependendo de qual é o nosso ambiente de execução. A declaração dela, de qualquer forma, será a mesma:

188 ⟨ Cabeçalhos Weaver 45 ⟩ +≡
```
void Wloop(void(*f)(void));
```

¶ No caso do nosso ambiente de execução ser o de um programa Linux normal, a definição da função é:

189 ⟨ API Weaver: Definições 149 ⟩ +≡
```
#if W_TARGET ≡ W_ELF
  void Wloop(void(*f)(void))
  {
    Wflush_input( );
    for ( ; ; ) {
      f( );
    }
  }
#endif
```

¶ Já se estamos no ambiente de execução de um navegador de Internet, temos preocupações adicionais. Precisamos registrar uma função como um loop principal. Mas se já existe um loop principal anteriormente registrado, precisamos cancelar ele primeiro.

190 ⟨ Cabeçalhos Weaver 45 ⟩ +≡
```
#if W_TARGET ≡ W_WEB
#include <emscripten.h>
#endif
```

191 ¶⟨ API Weaver: Definições 149 ⟩ +≡
```
#if W_TARGET ≡ W_WEB
  void Wloop(void(*f)(void))
  {
    emscripten_cancel_main_loop( );
    Wflush_input( );
      /* O segundo argumento é o número de frames por segundo: */
```

```
    emscripten_set_main_loop(f, 0, 1);
  }
#endif
```

¶ Tudo isso significa que um loop principal nunca chega ao fim. Podemos apenas invocar outro loop principal recursivamente dentro do atual. Não há como evitar esta limitação com a atual API Emscripten que precisa usar *emscripten_set_main_loop* para ativar o loop sem interferir na usabilidade do navegador de Internet. Isso também com que todo loop principal seja uma função que não retorna nada e nem recebe argumentos.

A única possibilidade de evitar isso seria se fosse possível usar clausuras (*closures*). Neste caso, poderíamos definir *Wloop* como uma macro que expandiria para a definição de uma clausura que poderia ter acesso à todas as variáveis da função atual ao mesmo tempo em que ela poderia ser passada para a função de invocaçã do loop. O único compilador compatível com Emscripten é o Clang, que até implementa clausuras por meio de uma extensão não-portável chamada de "blocos". O problema é que um bloco não é intercambiável e nem pode ser convertido para uma função. Então não seria possível passá-lo para a atual função da API Emscripten que espera uma função. O GCC suporta clausuras na forma de funções aninhadas por meio de extensão não-portável, mas o GCC não é compatível com Emscripten. Então simplesmente não temos como evitar este efeito colateral.

4.4 O Mouse

Um mouse do nosso ponto de vista é como se fosse um teclado, mas com menos teclas. O Xlib reconhece que mouses podem ter até 5 botões (*Button1*, *Button2*, *Button3*, *Button4* e *Button5*). O SDL, tentando manter portabilidade, em sua versão 1.2 reconhece 3 botões (`SDL_BUTTON_LEFT`, `SDL_BUTTON_MIDDLE`, `SDL_BUTTON_RIGHT`). Convenientemente, ambas as bibliotecas numeram cada um dos botões sequencialmente à partir do número 1. Nós iremos suportar 5 botões, mas um jogo deve assumir que apenas dois botões são realmente garantidos: o botão direito e esquerdo.

Além dos botões, um mouse possui também uma posição (x, y) na janela em que o jogo está. Mas às vezes mais importante do que sabermos a posição é sabermos se o mouse está se movendo ou não. E caso esteja se movendo, para onde ele está indo e em qual velocidade. Ambas as informações podem ser captadas por valores (dx, dy) que capturam em qual posição estará no mouse em 1 segundo se ele manter o mesmo deslocamento observado entre estre frame e o anterior.

Em suma, podemos representar o mouse como a seguinte estrutura:

193 ⟨ API Weaver: Definições 149 ⟩ +≡
```
struct _mouse Wmouse;
```

194 ¶⟨ Cabeçalhos Weaver 45 ⟩ +≡
```
extern struct _mouse {      /* Posições de 1 a 5 representarão cada um
        dos botões e o 6 é reservado para qualquer tecla. */
  int buttons[7];
  int x, y, dx, dy;
} Wmouse;
```

¶ E a tradução dos botões, dependendo do ambiente de execução será dada por:

195 ⟨ Cabeçalhos Weaver 45 ⟩ +≡
```
#if W_TARGET ≡ W_ELF
#define W_MOUSE_LEFT  Button1
#define W_MOUSE_MIDDLE  Button2
#define W_MOUSE_RIGHT  Button3
#define W_MOUSE_B1  Button4
#define W_MOUSE_B2  Button5
#endif
#if W_TARGET ≡ W_WEB
#define W_MOUSE_LEFT  SDL_BUTTON_LEFT
#define W_MOUSE_MIDDLE  SDL_BUTTON_MIDDLE
#define W_MOUSE_RIGHT  SDL_BUTTON_RIGHT
#define W_MOUSE_B1  4
#define W_MOUSE_B2  5
```

```
#endif
```

¶ Agora podemos inicializar os vetores de botões soltos e pressionados:

196 ⟨ API Weaver: Inicialização 81 ⟩ +≡

```
{
  int i;
  for (i = 0; i < 5; i++) Wmouse.buttons[i] = 0;
  for (i = 0; i < 5; i++) {
    _pressed_buttons[i] = 0;
    _released_buttons[i] = 0;
  }
}
```

¶ Imediatamente antes de tratarmos eventos, precisamos percorrer a lista de botões pressionados para atualizar seus valores e a lista de botões recém-soltos para removê-los da lista:

197 ⟨ API Weaver: Imediatamente antes de tratar eventos 178 ⟩ +≡

```
{
  int i, button;      /* Limpar o vetor de botõoes soltos e zerar seus valores
      no vetor de mouse: */
  for (i = 0; i < 5; i++) {
    button = _released_buttons[i];
    while (Wmouse.buttons[button] > 0) {
      int j;
      for (j = i; j < 4; j++) {
        _released_buttons[j] = _released_buttons[j + 1];
      }
      _released_buttons[4] = 0;
      button = _released_buttons[i];
    }
    if (button ≡ 0) break;
    Wmouse.buttons[button] = 0;
    _released_buttons[i] = 0;
  }     /* Para botões pressionados, incrementar o tempo em que eles estão
       pressionadas: */
  for (i = 0; i < 5; i++) {
    button = _pressed_buttons[i];
    if (button ≡ 0) break;
    Wmouse.buttons[button] += _elapsed_milisseconds;
  }
}
```

¶ Tendo esta estrutura pronta, iremos então tratar a chegada de eventos de botões do mouse sendo pressionados caso estejamos em um ambiente de execução baseado em Xlib:

198 ⟨ API Weaver: Trata Evento Xlib 127 ⟩ +≡

```
if (event.type ≡ ButtonPress) {
  unsigned int code = event.xbutton.button;
  int i;      /* Adiciona na lista de botões pressionados: */
  for (i = 0; i < 5; i++) {
    if (_pressed_buttons[i] ≡ 0 ∨ _pressed_buttons[i] ≡ code) {
      _pressed_buttons[i] = code;
      break;
    }
  }      /* Atualiza vetor de mouse se a tecla não estava sendo pressionada.
            Ignoramos se o evento está sendo gerado mais de uma vez sem que o
            botão seja solto ou caso o evento seja gerado imediatamente depois
            de um evento de soltar o mesmo botão: */
  if (Wmouse.buttons[code] ≡ 0)  Wmouse.buttons[code] = 1;
  else if (Wmouse.buttons[code] < 0)  Wmouse.buttons[code] *= −1;
  continue;
}
```

¶ E caso um botão seja solto, também tratamos tal evento:

199 ⟨ API Weaver: Trata Evento Xlib 127 ⟩ +≡

```
if (event.type ≡ ButtonRelease) {
  unsigned int code = event.xbutton.button;
  int i;      /* Remove da lista de botões pressionados */
  for (i = 0; i < 5; i++) {
    if (_pressed_buttons[i] ≡ code) {
      _pressed_buttons[i] = 0;
      break;
    }
  }
  for ( ; i < 4; i++) {
    _pressed_buttons[i] = _pressed_buttons[i + 1];
  }
  _pressed_buttons[4] = 0;      /* Adiciona na lista de botões soltos: */
  for (i = 0; i < 5; i++) {
    if (_released_buttons[i] ≡ 0 ∨ _released_buttons[i] ≡ code) {
      _released_buttons[i] = code;
      break;
    }
  }      /* Atualiza vetor de mouse */
  Wmouse.buttons[code] *= −1;
  continue;
```

```
}
```

¶ No ambiente de execução com SDL também precisamos checar quando um botão é pressionado:

200 ⟨ API Weaver: Trata Evento SDL 184 ⟩ +≡

```
if (event.type ≡ SDL_MOUSEBUTTONDOWN) {
  unsigned int code = event.button.button;
  int i;      /* Adiciona na lista de botões pressionados */
  for (i = 0; i < 5; i++) {
    if (_pressed_buttons[i] ≡ 0 ∨ _pressed_buttons[i] ≡ code) {
      _pressed_buttons[i] = code;
      break;
    }
  }      /* Atualiza vetor de mouse se o botão já não estava sendo
            pressionado antes. */
  if (Wmouse.buttons[code] ≡ 0) Wmouse.buttons[code] = 1;
  else if (Wmouse.buttons[code] < 0) Wmouse.buttons[code] *= −1;
  continue;
}
```

¶ E quando um botão é solto:

201 ⟨ API Weaver: Trata Evento SDL 184 ⟩ +≡

```
if (event.type ≡ SDL_MOUSEBUTTONUP) {
  unsigned int code = event.button.button;
  int i;      /* Remove da lista de botões pressionados */
  for (i = 0; i < 5; i++) {
    if (_pressed_buttons[i] ≡ code) {
      _pressed_buttons[i] = 0;
      break;
    }
  }
  for ( ; i < 4; i++) {
    _pressed_buttons[i] = _pressed_buttons[i + 1];
  }
  _pressed_buttons[4] = 0;      /* Adiciona na lista de botões soltos: */
  for (i = 0; i < 5; i++) {
    if (_released_buttons[i] ≡ 0 ∨ _released_buttons[i] ≡ code) {
      _released_buttons[i] = code;
      break;
    }
  }      /* Atualiza vetor de teclado */
  Wmouse.buttons[code] *= −1;
  continue;
}
```

¶ E finalmente, o caso especial para verificar se qualquer botão foi pressionado:

202 ⟨ API Weaver: Loop Principal 47 ⟩ +≡
```
Wmouse.buttons[W_ANY] = (_pressed_buttons[0] ≠ 0);
```

¶

4.4.1 Obtendo o movimento

Agora iremos calcular o movimento do mouse. Primeiramente, no início do programa devemos zerar tais valores para evitarmos valores absurdos na primeira iteração:

204 ⟨ API Weaver: Inicialização 81 ⟩ +≡
```
{
  Wmouse.x = Wmouse.y = Wmouse.dx = Wmouse.dy = 0;
}
```

¶ É importante que no início de cada iteração, antes de tratarmos os eventos, nós zeremos os valores (dx, dy) do mouse. Caso o mouse não receba nenhum evento de movimento, tais valores estarão corretos. Já se ele receber, aí de qualquer forma teremos a chance de atualizar os valores no tratamento do evento:

205 ⟨ API Weaver: Imediatamente antes de tratar eventos 178 ⟩ +≡
```
{
  Wmouse.dx = Wmouse.dy = 0;
}
```

¶ continue; Em seguida, cuidamos do caso no qual temos um evento Xlib de movimento do mouse:

206 ⟨ API Weaver: Trata Evento Xlib 127 ⟩ +≡
```
if (event.type ≡ MotionNotify) {
  int x, y, dx, dy;
  x = event.xmotion.x;
  y = event.xmotion.y;
  dx = x − Wmouse.x;
  dy = y − Wmouse.y;
  Wmouse.dx = ((float) dx/_elapsed_milisseconds) ∗ 1000;
  Wmouse.dy = ((float) dy/_elapsed_milisseconds) ∗ 1000;
  Wmouse.x = x;
  Wmouse.y = y;
  continue;
}
```

¶ Agora é só usarmos a mesma lógica para tratarmos o evento SDL:

207 ⟨ API Weaver: Trata Evento SDL 184 ⟩ +≡
```
if (event.type ≡ SDL_MOUSEMOTION) {
  int x, y, dx, dy;
  x = event.motion.x;
  y = event.motion.y;
  dx = x − Wmouse.x;
  dy = y − Wmouse.y;
  Wmouse.dx = ((float) dx/_elapsed_milisseconds) ∗ 1000;
  Wmouse.dy = ((float) dy/_elapsed_milisseconds) ∗ 1000;
  Wmouse.x = x;
  Wmouse.y = y;
  continue;
}
```

¶ E a última coisa que precisamos fazer é zerar e limpar todos os vetores de botões e variáveis de movimento toda vez que for requisitado limpar todos os buffers de entrada. Como ocorre antes de entrarmos em um loop principal:

208 ⟨ Limpar Entrada 208 ⟩ ≡
```
{
  int i;
  for (i = 0; i < 5; i++) {
    _released_buttons[i] = 0;
    _pressed_buttons[i] = 0;
  }
  for (i = 0; i < 7; i++) Wmouse.buttons[i] = 0;
  Wmouse.dx = 0;
  Wmouse.dy = 0;
}
```
This code is used in chunk 182.

¶

Capítulo 5

Shaders

Aqui apresentamos todo o código que é executado na GPU ao invés da CPU. Como usaremos shaders, precisaremos usar e inicializar também a biblioteca GLEW:

210 ⟨ API Weaver: Inicialização 81 ⟩ +≡

```
{
  GLenum dummy;
  glewExperimental = GL_TRUE;
  GLenum err = glewInit( );
  if (err ≠ GLEW_OK) {
    fprintf(stderr, "ERROR: GLW not supported.\n");
    exit(1);
  }     /* Dependendo da versão, glewInit gera um erro completamente
           inócuo acusando valor inválido passado para alguma função. A
           linha seguinte serve apenas para ignorarmos o erro, impedindo-o de
           se propagar. */
  dummy = glGetError( );
  glewExperimental += dummy;
  glewExperimental −= dummy;
}
```

¶ Para isso, primeiro precisamos declarar na inicialização que iremos usá-los. As versões mais novas de OpenGL permitem 4 Shaders diferentes. Um para processar os vértices, outro para processar cada pixel e mais dois para adicionar vértices e informações aos modelos dentro da GPU. Mas quando programamos para WebGL, só podemos contar com o padrão OpenGL ES 1.0. Por isso, só podemos usar os dois primeiros tipos de shaders. Iremos declará-los abaixo:

211 ⟨ API Weaver: Definições 149 ⟩ +≡

```
static GLuint _vertex_shader, _fragment_shader;
```

¶ Primeiro precisamos avisar o servidor OpenGL que iremos usá-los. Isso dará à eles um ID para poderem ser referenciados:

212 ⟨ API Weaver: Inicialização 81 ⟩ +≡
```
{
  _vertex_shader = glCreateShader(GL_VERTEX_SHADER);
  _fragment_shader = glCreateShader(GL_FRAGMENT_SHADER);
}
```

¶ E quando o programa terminar, nós destruímos os shaders criados:

213 ⟨ API Weaver: Finalização 82 ⟩ +≡
```
{
  glDeleteShader(_vertex_shader);
  glDeleteShader(_fragment_shader);
}
```

¶ Mas como acrescentar o código para os shaders? O seu código é escrito em GLSL e é compilado durante a execução do programa que os invoca. O seu código deve então estar na memória do programa como uma string.

O problema é que não queremos definir o código GLSL desta forma. Idealmente, queremos que o código GLSL seja em parte definido por programação literária, já que ele é suficientemente próximo do código C. E se formos faze isso, será chato definirmos ele como string, pois teremos que ficar inserindo quebras de linha, temos que tomar cuidado para escapar abertura de aspas e coisas assim. Sem falar que perderemos a identação.

A solução? Iremos definir o código GLSL de cada shader para um arquivo. O Makefile será responsável por converter o arquivo de código GLSL para um outro arquivo onde cada caractere é traduzido para a representação em C de seu valor hexadecimal. E então, nós inserimos tais valores abaixo:

214 ⟨ API Weaver: Inicialização 81 ⟩ +≡
```
{ char vertex_source[] = {
#include "vertex.data"
, 00 } ; char fragment_source[] = {
#include "fragment.data"
, 00 } ;
const char *ptr1 = (char *) &vertex_source, *ptr2 = (char *)
    &fragment_source;
glShaderSource(_vertex_shader, 1, &ptr1, Λ);
glShaderSource(_fragment_shader, 1, &ptr2, Λ); }
```

¶ Agora compilamos os Shaders, imprimindo uma mensagem de erro e abortando o programa se algo der errado:

215 ⟨ API Weaver: Inicialização 81 ⟩ +≡
```
{ char
error [200] ;
GLint result;
glCompileShader(_vertex_shader);
glGetShaderiv(_vertex_shader, GL_COMPILE_STATUS, &result); if
    (result ≠ GL_TRUE) { glGetShaderInfoLog (_vertex_shader, 200, Λ, error
    ) ; fprintf (stderr, "ERROR: While compiling vertex shader: %s\n",
    error ) ;
exit(1); } glCompileShader(_fragment_shader);
glGetShaderiv(_fragment_shader, GL_COMPILE_STATUS, &result); if
    (result ≠ GL_TRUE) { glGetShaderInfoLog (_fragment_shader, 200, Λ,
    error ) ; fprintf (stderr,
    "ERROR: While compiling fragment shader: %s\n", error ) ;
exit(1); } }
```

¶ Uma vez que tenhamos compilado os shaders, precisamos criar um programa que irá contê-los:

216 ⟨ API Weaver: Definições 149 ⟩ +≡
```
GLuint _program;
```

217 ¶⟨ Cabeçalhos Weaver 45 ⟩ +≡
```
extern GLuint _program;
```

218 ¶⟨ API Weaver: Inicialização 81 ⟩ +≡
```
{
  _program = glCreateProgram( );
}
```

219 ¶⟨ API Weaver: Finalização 82 ⟩ +≡
```
{
  glDeleteProgram(_program);
}
```

¶ Depois de criado um programa, precisamos associá-lo aos shaders compilados. E quando terminarmos, vamos desassociá-los:

220 ⟨ API Weaver: Inicialização 81 ⟩ +≡
```
{
  glAttachShader(_program, _vertex_shader);
  glAttachShader(_program, _fragment_shader);
}
```

221 ¶⟨ API Weaver: Finalização 82 ⟩ +≡
```
{
  glDetachShader(_program, _vertex_shader);
  glDetachShader(_program, _fragment_shader);
}
```

¶ Tendo colocado todos os shaders juntos no programa, precisamos ligá-los entre si, verificando se um erro não ocorreu nesta etapa:

222 ⟨ API Weaver: Inicialização 81 ⟩ +≡
```
{ GLint result;
glLinkProgram(_program);
glGetProgramiv(_program, GL_LINK_STATUS, &result); if (result ≠ GL_TRUE)
    { char
error [200] ;
glGetProgramInfoLog (_program, 200, Λ, error ) ; fprintf
    (stderr, "ERROR: While linking shaders: %s\n", error ) ;
exit(1); } }
```

¶ Por fim, se nenhum erro aconteceu, podemos usar o programa:

223 ⟨ API Weaver: Inicialização 81 ⟩ +≡
```
glUseProgram(_program);
```

¶

5.1 Shader de Vértice

Este é o shader de vértice inicial com a computação feita pela GPU para cada vértice. Inicialmente ele será apenas um shader que passará adiante o que recebe de entrada:

A primeira coisa que recebemos de entrada é a posição do vértice:

225 ⟨ `project/src/weaver/vertex.glsl` 225 ⟩ ≡
```
# version 100
attribute vec3 vPosition; ⟨ Shader de Vértice: Declarações 232 ⟩
```
See also chunk 226.

¶ E no programa principal, passamos para a saída o que recebemos de entrada:

226 ⟨ `project/src/weaver/vertex.glsl` 225 ⟩ +≡
```
void main( )
{
  gl_Position = vec4(vPosition, 1.0);
  ⟨ Shader de Vértice: Aplicar Matriz de Modelo 305 ⟩
  ⟨ Shader de Vértice: Ajuste de Resolução 236 ⟩
  ⟨ Shader de Vértice: Câmera (Perspectiva) 330 ⟩
  ⟨ Shader de Vértice: Cálculo do Vetor Normal 252 ⟩
}
```

¶ Isso significa que no programa principal em C, nós precisamos obter e armazenar a localização da variável *vPosition* dentro do programa de shader para que possamos passar tal variável:

227 ⟨ API Weaver: Definições 149 ⟩ +≡
```
GLint _shader_vPosition;
```

¶ E se nosso shader foi compilado sem problemas, não teremos dificuldades em obter a sua localização:

228 ⟨ API Weaver: Inicialização 81 ⟩ +≡
```
_shader_vPosition = glGetAttribLocation(_program, "vPosition");
if (_shader_vPosition ≡ −1) {
  fprintf(stderr, "ERROR: Couldn't get shader attribute index.\n");
  exit(1);
}
```

¶

5.2 Shader de Fragmento

Agora o shader de fragmento, a ser processado para cada pixel que aparecer na tela.

230 ⟨ `project/src/weaver/fragment.glsl` 230 ⟩ ≡

```
# version 100
  ⟨ Shader de Fragmento: Declarações 240 ⟩
      void main( )
      {
        ⟨ Shader de Fragmento: Variáveis Locais 262 ⟩
        gl_FragColor = vec4 (0.5, 0.5, 0.5, 1.0);
        ⟨ Shader de Fragmento: Modelo Clássico de Iluminação 241 ⟩
        gl_FragColor = min(gl_FragColor, vec4 (1.0));
      }
```

¶

5.3 Corrigindo Diferença de Resolução Horizontal e Vertical

Por padrão aparecerá na tela qualquer primitiva geométrica que esteja na posição x no intervalo $[-1.0, +1.0]$ e na posição y no mesmo intervalo. Entretanto, nossa resolução pode variar horizontalmente ou verticalmente. Se a resolução horizontal for maior (como ocorre tipicamente), as figuras geométricas serão esticadas na horizontal. Se a resolução vertical for maior, as figuras serão esticadas verticalmente.

Precisamos fazer então com que a menor resolução (seja ela horizontal ou vertical) tenha o intervalo $[-1.0, +1.0]$, mas que a outra resolução represente um intervalo proporcionalmente maior. Isso faz com que telas horizontais maiores, ou mesmo o xinerama dê o benefício de uma visão horizontal maior.

Do ponto de vista do shader, teremos um multiplicador horizontal e vertical que aplicaremos sobre cada vértice antes de qualquer outra transformação:

232 ⟨ Shader de Vértice: Declarações 232 ⟩ ≡
```
uniform
    float Whorizontal_multiplier, Wvertical_multiplier;
```
See also chunks 248, 251, 301, and 328.
This code is used in chunk 225.

¶ Na inicialização do Weaver, devemos obter a localização destas variáveis no shader. A localização será armazenada nas variáveis abaixo:

233 ⟨ Cabeçalhos Weaver 45 ⟩ +≡
```
extern GLfloat _horizontal_multiplier, _vertical_multiplier;
```

234 ¶⟨ API Weaver: Definições 149 ⟩ +≡
```
GLfloat _horizontal_multiplier, _vertical_multiplier;
```

¶ O código de obtenção da localização junto com a inicialização dos multiplicadores:

235 ⟨ API Weaver: Inicialização 81 ⟩ +≡
```
{
  _horizontal_multiplier = glGetUniformLocation(_program,
      "Whorizontal_multiplier");
  if (W_width > W_height) glUniform1f(_horizontal_multiplier,((float)
        W_height/(float) W_width));
  else glUniform1f(_horizontal_multiplier,1.0);
  _vertical_multiplier = glGetUniformLocation(_program,
      "Wvertical_multiplier");
  if (W_height > W_width)
    glUniform1f(_vertical_multiplier,((float) W_width/(float) W_height));
```

```
    else glUniform1f(_vertical_multiplier, 1.0);
  }
```

¶ O uso dos multiplicadores para corrigir a posição do vértice deve sempre ocorrer depois da rotação do objeto. Mas antes da translação:

236 ⟨ Shader de Vértice: Ajuste de Resolução 236 ⟩ ≡

```
gl_Position *= vec4(Whorizontal_multiplier, Wvertical_multiplier, 1.0, 1.0);
```

This code is used in chunk 226.

¶ Lembrando que o código para realizar tal correção não termina por aí. Uma janela pode ter o seu tamanho modificado, e assim teremos uma resolução com valores diferentes. Por isso temos que atualizar as variáveis do shader toda vez que a janela ou canvas tem o seu tamanho mudado:

237 ⟨ Ações após Redimencionar Janela 237 ⟩ ≡

```
{
  if (W_width > W_height) glUniform1f(_horizontal_multiplier, ((float)
        W_height/(float) W_width));
  else glUniform1f(_horizontal_multiplier, 1.0);
  if (W_height > W_width)
    glUniform1f(_vertical_multiplier, ((float) W_width/(float) W_height));
  else glUniform1f(_vertical_multiplier, 1.0);
}
```

This code is used in chunks 129 and 131.

¶

5.4 O Modelo Clássico de Iluminação

Uma das principais utilidades do Shader de Fragmento é calcular efeitos de luz e sombra. Vamos começar com a luz. O ponto de partida para os efeitos de iluminação é o uso do Modelo Clássico de Iluminação. Ele costuma dividir a luz em três tipos diferentes: a luz ambiente (que representa a luz espalhada por um ambiente devido à ser refletida pelo conjunto de objetos que faz parte da cena), a luz difusa (luz emitida à partir de um ponto distante e que incide mais sobre superfícies voltadas diretamente para ele) e a luz especular (luz refletida por superfícies brilhantes).

Cada uma destas luzes pode possuir diferentes cores e intensidades.

5.4.1 A Luz Ambiente

Este é o tipo de luz mais simples que existe. Ela não muda de um objeto que está sendo renderizado para outro, não depende da posição dos objetos e nem da direção de cada uma de suas faces. Por causa disso, seus valores podem ser passados como sendo uma variável uniforme para o shader:

240 ⟨ Shader de Fragmento: Declarações 240 ⟩ ≡
uniform mediump vec3 Wambient_light;

See also chunks 253 and 254.

This code is used in chunk 230.

¶ A luz nada mais é do que um valor RGB. E iluminar usando esta luz significa simplesmente multiplicar o seu valor com o valor da cor do pixel que estamos para desenhar na tela:

241 ⟨ Shader de Fragmento: Modelo Clássico de Iluminação 241 ⟩ ≡
gl_FragColor ∗= *vec4* (*Wambient_light*, 1.0);

See also chunks 263 and 264.

This code is used in chunk 230.

¶ Dentro do shader é só isso. Agora só precisamos criar uma estrutura para armazenar a cor da luz (e sua intensidade):

242 ⟨ API Weaver: Definições 149 ⟩ +≡
struct *_ambient_light Wambient_light*;

243 ¶⟨ Cabeçalhos Weaver 45 ⟩ +≡
```
extern struct _ambient_light {
  float r, g, b;
  GLuint _shader_variable;
} Wambient_light;
```

¶ Durante a inicialização do programa precisamos inicializar os valores. Vamos começar deixando eles como sendo uma luz branca de intensidade máxima.

244 ⟨ API Weaver: Inicialização 81 ⟩ +≡
```
{
  Wambient_light.r = 0.5;
  Wambient_light.g = 0.5;
  Wambient_light.b = 0.5;
  Wambient_light._shader_variable = glGetUniformLocation(_program,
      "Wambient_light");
  glUniform3f(Wambient_light._shader_variable, Wambient_light.r,
      Wambient_light.g, Wambient_light.b);
}
```

¶ E toda vez que quisermos atualizar o valor da luz ambiente, podemos usar a seguinte função:

245 ⟨ Cabeçalhos Weaver 45 ⟩ +≡
```
void Wset_ambient_light_color(float r, float g, float b);
```

246 ¶⟨ API Weaver: Definições 149 ⟩ +≡
```
void Wset_ambient_light_color(float r, float g, float b)
{
  Wambient_light.r = r;
  Wambient_light.g = g;
  Wambient_light.b = b;
  glUniform3f(Wambient_light._shader_variable, Wambient_light.r,
      Wambient_light.g, Wambient_light.b);
}
```

¶

5.4.2 A Luz Direcional

A Luz Direcional é formada por raios paralelos de luz que percorrem sempre a mesma direção em uma cena. Ela representa luz emitida por pontos luminosos distantes. Por isso, a sua intensidade não depende da posição de um objeto, apenas da orientação de suas faces. Se uma face está voltada para o lado oposto da luz, ela não recebe iluminação. Se estiver voltado para a luz, recebe a maior quantidade possível de raios. É uma boa forma de simular a luz do sol em uma boa parte das cenas.

Para calcularmos melhor a orientação de um polígono em relação à fonte de luz, nós precisamos saber o valor da normal de cada vértice do polígono. Ou seja, precisamos saber o valor de um vetor unitário que tenha a mesma

direção e sentido do vértice. Quando geramos o valor de cada pixel no shader de fragmento, obteremos assim uma interpolação deste valor e saberemos aproximadamente qual é a normal para cada pixel renderizado da imagem. Então, no shader de vértice nós devemos receber como atributo também a normal de cada vértice junto com suas coordenadas:

248 ⟨ Shader de Vértice: Declarações 232 ⟩ +≡
```
attribute vec3 VertexNormal;
```

¶ A localização deste atributo no Shader precisa ser obtida pelo programa em C, e por isso definimos e inicializamos a variável:

249 ⟨ API Weaver: Definições 149 ⟩ +≡
```
GLint _shader_VertexNormal;
```

250 ¶⟨ API Weaver: Inicialização 81 ⟩ +≡
```
_shader_VertexNormal = glGetAttribLocation(_program, "VertexNormal");
if (_shader_vPosition ≡ −1) {
  fprintf(stderr, "ERROR: Couldn't get shader attribute index.\n");
  exit(1);
}
```

¶ Ao longo do shader de vértice nós provavelmente podemos querer modificar o vetor normal do vértice recebido. Muito provavelmente para levar em conta eventuais rotações e transformações do modelo. E no fim vamos querer passar o valor adiante para o shader de fragmento, onde o valor da iluminação de cada pixel será computado. Para passar adiante o valor da normal, usaremos:

251 ⟨ Shader de Vértice: Declarações 232 ⟩ +≡
```
varying vec3 Wnormal;
```

¶ E para modificarmos o valor conforme necessário, usamos:

252 ⟨ Shader de Vértice: Cálculo do Vetor Normal 252 ⟩ ≡
```
Wnormal = VertexNormal;
```
This code is used in chunk 226.

¶ No shader de fragmento nós precisaremos receber do de vértice um vetor normal interpolado para cada pixel dentro do polígono que se está desenhando:

253 ⟨ Shader de Fragmento: Declarações 240 ⟩ +≡
```
varying mediump vec3 Wnormal;
```

¶ Duas outras coisas que precisamos receber no shader de fragmento: a direção da luz e a sua cor:

254 ⟨ Shader de Fragmento: Declarações 240 ⟩ +≡
```
uniform mediump vec3 Wlight_direction;
uniform mediump vec3 Wdirectional_light;
```

¶ Assim como no caso da luz ambiente, criamos uma estrutura para que o programa em C possa acessar os valores da luz direcional:

255 ⟨ API Weaver: Definições 149 ⟩ +≡
```
struct _directional_light Wdirectional_light;
```

256 ¶⟨ Cabeçalhos Weaver 45 ⟩ +≡
```
extern struct _directional_light {      /* A cor: */
  float r, g, b;      /* A direção: */
  float x, y, z;
  GLuint _shader_variable, _direction_variable;
} Wdirectional_light;
```

¶ Na inicialização fazemos com que a luz torne-se branca e aponte para uma direção padrão:

257 ⟨ API Weaver: Inicialização 81 ⟩ +≡
```
{
  Wdirectional_light.r = 1.0;
  Wdirectional_light.g = 1.0;
  Wdirectional_light.b = 1.0;
  Wdirectional_light.x = 0.5;
  Wdirectional_light.y = 0.5;
  Wdirectional_light.z = -1.0;
  Wdirectional_light._shader_variable = glGetUniformLocation(_program,
      "Wdirectional_light");
  glUniform3f(Wdirectional_light._shader_variable, Wdirectional_light.r,
      Wdirectional_light.g, Wdirectional_light.b);
  Wdirectional_light._direction_variable = glGetUniformLocation(_program,
      "Wlight_direction");
  glUniform3f(Wdirectional_light._direction_variable, Wdirectional_light.x,
      Wdirectional_light.y, Wdirectional_light.z);
}
```

¶ Tal como na luz ambiente, precisamos de uma função para ajustar a sua cor:

258 ⟨ Cabeçalhos Weaver 45 ⟩ +≡
```
void Wset_directional_light_color(float r, float g, float b);
```

259 ¶⟨ API Weaver: Definições 149 ⟩ +≡
```
void Wset_directional_light_color(float r, float g, float b)
{
  Wdirectional_light.r = r;
  Wdirectional_light.g = g;
  Wdirectional_light.b = b;
  glUniform3f(Wdirectional_light._shader_variable, Wdirectional_light.r,
      Wdirectional_light.g, Wdirectional_light.b);
}
```

¶ E além disso, para este tipo de luz precisamos também de uma função para modificarmos a sua direção:

260 ⟨ Cabeçalhos Weaver 45 ⟩ +≡
```
void Wset_directional_light_direction(float x, float y, float z);
```

261 ¶⟨ API Weaver: Definições 149 ⟩ +≡
```
void Wset_directional_light_direction(float x, float y, float z)
{
  Wdirectional_light.x = x;
  Wdirectional_light.y = y;
  Wdirectional_light.z = z;
  glUniform3f(Wdirectional_light._direction_variable, Wdirectional_light.x,
      Wdirectional_light.y, Wdirectional_light.z);
}
```

¶ Agora que todos os valores para a luz direcional já foram passados, o que precisamos é fazer o shader de fragmento usar tais valores no cálculo da cor de cada pixel. Primeiro precisamos de uma variável local para calcularmos a intensidade da luz, que irá variar de acordo com a direção da luz e a normal do ponto em que estamos:

262 ⟨ Shader de Fragmento: Variáveis Locais 262 ⟩ ≡
```
mediump
    float directional_light;
```
This code is used in chunk 230.

263 ¶⟨ Shader de Fragmento: Modelo Clássico de Iluminação 241 ⟩ +≡
```
directional_light = max(0.0, dot(Wnormal, Wdirectional_light));
```

¶ Em seguida, multiplicamos a intensidade obtida pela própria cor da luz e somamos ao valor já obtido da cor do pixel modificado pela luz ambiente:

264 ⟨ Shader de Fragmento: Modelo Clássico de Iluminação 241 ⟩ +≡
```
gl_FragColor += vec4(directional_light * Wdirectional_light, 0.0);
```

¶

Capítulo 6

A Câmera

Capítulo 7

Objetos Básicos

Vamos começar agora a definir os objetos reais dos mundos que podemos construir com Weaver. Poderão haver vários tipos de objetos. A ideia é que uma nuvem de partículas seja um objeto, uma mesa seja outro objeto, a água de um cenário um terceiro e o golfinho que nada nela seja mais um.

Mas vários objetos diferentes podem ter características diferentes. Um objeto pode ser apenas uma seta ou um ícone que aparece, sem que ele seja algo sólido capaz de colidir com os outros. Da mesma forma, a nuvem de partículas também não colide, mas a sua forma muda. A água tanto colide como muda de forma. A mesa colide, mas não muda de forma. E o golfinho clide, se move, mas muda de forma de maneira mais bem-definida (segundo um esqueleto).

Por causa disso, definiremos os objetos Weaver como uma união de vários tipos diferentes. Todos eles terão uma variável inteira de tipo para indicar que tipo de objeto eles são, outra para indicar quantos vértices eles tem e uma terceira para indicar a posição inicial de cada vértice, onde o centro do objeto é a coordenada $(0, 0, 0)$. Ou, $(0, 0)$ se estivermos em um universo bidimensional.

Além disso, é necessário diferenciar entre uma definição de objeto e representantes do objeto em si. Em Orientação à Objetos, seria o conceito de classe e instância. Todas as cadeiras poderão ser definidas como tendo os mesmos vértices exatamente nas mesmas coordenadas. Seria desperdício de memória fazer com que todas as cadeiras memorizem cada um de seus vértices. Cada cadeira precisa memorizar apenas uma matriz que representa a sua posição e outra que representa como ela está rotacionada (nem todas podem estar de pé e voltadas para a mesma direção). E precisa também de um ponteiro para a definição geral de todas as cadeiras onde informações mais gerais podem ser obtidas.

Então, o que chamamos de definição de um objeto (ou classe) deverá ter também um vetor com informações específicas de cada exemplo de objeto (instâncias). A quantidade de memória que cada instância usa é baixa em relação à memória da classe (que vai ter a lista de vértices, texturas e essas coisas). Sendo assim, podemos usar um vetor estático para armazenar cada instância. A questão é: qual o tamanho deste vetor? Ou qual o número máximo de instâncias que uma

classe pode ter? Esta questão é relevante pelo fato de querermos armazenar o máximo possível de coisas em vetores sequenciais ao invés de coisas que usam muitos ponteiros como referência (listas encadeadas). Além disso, nosso gerenciador de memória não suporta algo como *realloc*. Então, contamos que haja no `conf/conf.h` uma macro que informe isso:

- `W_MAX_CLASSES`: O número máximo de classes que pode ser definida.
- `W_MAX_INSTANCES`: O número máximo de instâncias que um objeto Weaver pode ter. Se você definir uma cadeira, o número máximo de cadeiras simultâneas que podem existir é este.

E a nossa definição de Objeto Weaver é:

267 ⟨ `project/src/weaver/wobject.h` 267 ⟩ ≡
```
#ifndef _wobject_h_
#define _wobject_h_
#ifdef __cplusplus
  extern "C" {
#endif
      ⟨ Inclui Cabeçalho de Configuração 42 ⟩
      ⟨ Wobject: Cabeçalho 268 ⟩
      ⟨ Wobject: Declaração 277 ⟩
#ifdef __cplusplus
      }
#endif
#endif
```

268 ¶⟨ Wobject: Cabeçalho 268 ⟩ ≡
```
  union Wobject {
    ⟨ Wobject: Tipo de Objeto 273 ⟩
  };
  union Wclass {
    ⟨ Wobject: Tipo de Classe 272 ⟩
  };
```
See also chunks 271, 348, 349, and 350.
This code is used in chunk 267.

269 ¶⟨ `project/src/weaver/wobject.c` 269 ⟩ ≡
```
#include "weaver.h"
  ⟨ Wobject: Definição 278 ⟩
```

¶

7.1 Definindo a Classe de Objetos Básicos

O primeiro tipo de objeto que definiremos são objetos básicos. Ou, *basic*, como definiremos no código. Tudo o que pode ser feito com um objeto básico é exibir seus vértices, movê-los e rotacioná-los. Em suma, qualquer coisa que pode ser feita com qualquer tipo de objeto. Objetos Básicos não são úteis por si só. Mas o código inicial que criarmos para ele poderá ser reaproveitado em todos os outros objetos, então nós os criamos mais para usá-los internamente para definir outros objetos que para usá-los externamente.

Apesar deles serem relativamente simples, já precisamos de vários dados diferentes para conseguirmos defini-los:

271 ⟨ Wobject: Cabeçalho 268 ⟩ +≡ /* Tipo de Wobject: */
```
#define W_NONE  0
#define W_BASIC  1
```

272 ¶⟨ Wobject: Tipo de Classe 272 ⟩ ≡
```
struct {
  int type;
  int number_of_objects;
  int number_of_vertices;
  int essential;
  float *vertices;
  GLuint _vertex_object, _buffer_object;
  float width, height, depth;
  union Wobject instances[W_MAX_INSTANCES];
} basic;
```
See also chunk 352.

This code is used in chunk 268.

¶ Destas coisas que usamos na definição, a única coisa que ainda não discutimos é a variável *essential*. O propósito desta variável tem à ver com o gerenciamento de novas instâncias. Vamos supor que `W_MAX_INSTANCES` é igual à 5. Isso significa que cada classe de objeto só pode ter 5 instâncias. Mas o que deve acontecer se já existirem 5 objetos e então pedirmos para criar mais um? Se definimos este tipo de objeto como não-essencial (a variável for 0), então iremos apagar o objeto mais antigo e colocamos o novo objeto em seu lugar. Já se o objeto for essencial, não podemos apagá-lo somente para que ceda lugar à um novo. Neste caso, a criação do novo objeto irá falhar e a função de criação retornará Λ. Por padrão, assumiremos que todo objeto será não-essencial, à menos que diga-se o contrário.

A instância de um objeto básico terá a seguinte forma:

273 ⟨ Wobject: Tipo de Objeto 273 ⟩ ≡
```
struct {
  int type;
  int number;
  int visible;
  float x, y, z;
  float scale_x, scale_y, scale_z;
  float translation[4][4];
  float angle_x, angle_y, angle_z;
  float rotation_x[4][4], rotation_y[4][4], rotation_z[4][4];
  float rotation_total[4][4];
  float scale_matrix[4][4];
  float model_matrix[4][4];
  float model_view_matrix[4][4];
  float normal_matrix[4][4];
  union Wclass *wclass;
} basic;
```
See also chunk 353.
This code is used in chunk 268.

¶ Cada objeto terá um número entre 0 e `W_MAX_INSTANCES`. Objetos mais antigos terão números menores. Esta variável será usada para identificarmos quais são os objetos mais antigos. Estes serão os desalocados se for necessário e se a sua classe for marcada como não-essencial.

Outra coisa que devemos lembrar. A própria API Weaver deve estar ciente de todas as classes já definidas. Isso precisa ser feito para que durante o loop principal ela possa fazer coisas como desenhá-las na tela ou calcular interações físicas dependendo da forma. Por causa disso, vamos definir um vetor de classes de objetos a ser usado durante a execução do programa:

274 ⟨ Cabeçalhos Weaver 45 ⟩ +≡
```
#include "wobject.h"
extern union Wclass _wclasses[W_MAX_CLASSES];
```

275 ¶⟨ API Weaver: Definições 149 ⟩ +≡
```
union Wclass _wclasses[W_MAX_CLASSES];
```

¶ Inicializamos esta lista de classes no início do programa:

276 ⟨ API Weaver: Inicialização 81 ⟩ +≡
```
{
  int i;
  for (i = 0; i < W_MAX_CLASSES; i++) {
    _wclasses[i].basic.type = W_NONE;
  }
}
```

¶ Como objetos básicos não foram feitos para serem usados diretamente, a sua função de definição começará com um "underline":

277 ⟨ Wobject: Declaração 277 ⟩ ≡
```
union Wclass *_define_basic_object(int number_of_vertices, float *vertices);
```
See also chunks 279, 283, 285, 290, 293, 296, 299, 321, 355, 357, and 360.
This code is used in chunk 267.

278 ¶⟨ Wobject: Definição 278 ⟩ ≡
```
union Wclass *_define_basic_object(int number_of_vertices, float *vertices)
{
  int i, j, total;      /* Variáveis usadas para deixar a coordenada (0,0,0)
      no centro da imagem: */
  float min_x, max_x, min_y, max_y, min_z, max_z;
  float x_offset, y_offset, z_offset;
  min_x = min_y = min_z = INFINITY;
  max_x = max_y = max_z = -INFINITY;
    /* Primeiro tentamos alocar uma classe no vetor de classes: */
  for (i = 0; i < W_MAX_CLASSES; i++) {
    if (_wclasses[i].basic.type ≡ W_NONE) break;
  }
  if (i ≥ W_MAX_CLASSES) return Λ;
       /* Se conseguimos, preenchemos os dados da classe: */
  _wclasses[i].basic.type = W_BASIC;
  _wclasses[i].basic.number_of_objects = 0;
  _wclasses[i].basic.number_of_vertices = number_of_vertices;
  _wclasses[i].basic.essential = 0;      /* O vetor de vértices deve ser grande
      o bastante para armazenar as coordenadas do vértice (3 floats) e o
      vetor normal de cada vértice para o cálculo de iluminação (3 floats) */
  _wclasses[i].basic.vertices = (float *)
      Walloc(sizeof(float) * (number_of_vertices + 1) * 6);
  if (_wclasses[i].basic.vertices ≡ Λ) return Λ;
  total = (number_of_vertices + 1) * 6;
    /* Vértices armazenados no vetor à partir da posição 1. A posição 0 é
      ignorada. Isso somente em _wclasses[i].basic.vertices, não em vertices,
      que é de onde lemos o vértice. Ver abaixo o motivo. */
  for (j = 6; j < total; j += 6) {
    _wclasses[i].basic.vertices[j] = vertices[(j - 6)/2];
    if (min_x > vertices[j]) min_x = vertices[(j - 6)/2];
    if (max_x < vertices[j]) max_x = vertices[(j - 6)/2];
    _wclasses[i].basic.vertices[j + 1] = vertices[(j - 4)/2];
    if (min_y > vertices[j + 1]) min_y = vertices[(j - 4)/2];
    if (max_y < vertices[j + 1]) max_y = vertices[(j - 4)/2];
    _wclasses[i].basic.vertices[j + 2] = vertices[(j - 2)/2];
    if (min_z > vertices[j + 2]) min_z = vertices[(j - 2)/2];
```

```
    if (max_z < vertices[j + 2]) max_z = vertices[(j − 2)/2];
  }     /* Corrigindo a posição dos vértices para que (0,0,0) fique no meio:
           */
  x_offset = −(min_x + max_x)/2;
  y_offset = −(min_y + max_y)/2;
  z_offset = −(min_z + max_z)/2;
  for (j = 6; j < total; j += 6) {
    _wclasses[i].basic.vertices[j] += x_offset;
    _wclasses[i].basic.vertices[j + 1] += y_offset;
    _wclasses[i].basic.vertices[j + 2] += z_offset;
  }     /* Preenchendo altura, largura e comprimento: */
  _wclasses[i].basic.width = max_x − min_x;
  _wclasses[i].basic.height = max_y − min_y;
  _wclasses[i].basic.depth = max_z − min_z;
     /* Inicializando os vértices e buffers OpenGL */
  glGenVertexArrays(1, &_wclasses[i].basic._vertex_object);
  glBindVertexArray(_wclasses[i].basic._vertex_object);
  glGenBuffers(1, &_wclasses[i].basic._buffer_object);
  glBindBuffer(GL_ARRAY_BUFFER, _wclasses[i].basic._buffer_object);
  glBufferData(GL_ARRAY_BUFFER, sizeof(float)*6*(number_of_vertices+1),
       _wclasses[i].basic.vertices, GL_STATIC_DRAW);
     /* Inicializando as instâncias */
  for (j = 0; j < W_MAX_INSTANCES; j++) {
    int k, l;

    _wclasses[i].basic.instances[j].basic.type = W_NONE;
    _wclasses[i].basic.instances[j].basic.wclass = &(_wclasses[i]);
    _wclasses[i].basic.instances[j].basic.number = −1;
    _wclasses[i].basic.instances[j].basic.x = 0;
    _wclasses[i].basic.instances[j].basic.y = 0;
    _wclasses[i].basic.instances[j].basic.z = 0;
    _wclasses[i].basic.instances[j].basic.scale_x = 1.0;
    _wclasses[i].basic.instances[j].basic.scale_y = 1.0;
    _wclasses[i].basic.instances[j].basic.scale_z = 1.0;
    _wclasses[i].basic.instances[j].basic.angle_x = 0;
    _wclasses[i].basic.instances[j].basic.angle_y = 0;
    _wclasses[i].basic.instances[j].basic.angle_z = 0;
    _wclasses[i].basic.instances[j].basic.visible = 1;
       /* inicializando as matrizes como matrizes identidade: */
    for (k = 0; k < 4; k++)
      for (l = 0; l < 4; l++) {
        if (k ≡ l) {
          _wclasses[i].basic.instances[j].basic.rotation_x[k][l] = 1;
          _wclasses[i].basic.instances[j].basic.rotation_y[k][l] = 1;
          _wclasses[i].basic.instances[j].basic.rotation_z[k][l] = 1;
          _wclasses[i].basic.instances[j].basic.rotation_total[k][l] = 1;
```

```
            _wclasses[i].basic.instances[j].basic.translation[k][l] = 1;
            _wclasses[i].basic.instances[j].basic.scale_matrix[k][l] = 1;
            _wclasses[i].basic.instances[j].basic.model_matrix[k][l] = 1;
            _wclasses[i].basic.instances[j].basic.model_view_matrix[k][l] = 1;
            _wclasses[i].basic.instances[j].basic.normal_matrix[k][l] = 1;
          }
          else {
            _wclasses[i].basic.instances[j].basic.rotation_x[k][l] = 0;
            _wclasses[i].basic.instances[j].basic.rotation_y[k][l] = 0;
            _wclasses[i].basic.instances[j].basic.rotation_z[k][l] = 0;
            _wclasses[i].basic.instances[j].basic.rotation_total[k][l] = 0;
            _wclasses[i].basic.instances[j].basic.translation[k][l] = 0;
            _wclasses[i].basic.instances[j].basic.scale_matrix[k][l] = 0;
            _wclasses[i].basic.instances[j].basic.model_matrix[k][l] = 0;
            _wclasses[i].basic.instances[j].basic.model_view_matrix[k][l] = 0;
            _wclasses[i].basic.instances[j].basic.normal_matrix[k][l] = 0;
          }
        }
      }
      return &(_wclasses[i]);
    }
```

See also chunks 280, 284, 286, 291, 294, 297, 300, 322, 356, 358, and 361.
This code is used in chunk 269.

¶ Talvez seja estranho no código acima que aloquemos espaço para $n + 1$ vértices, quando precisamos de n vértices e que ignoremos a primeira posição (a posição 0). Isso acontece que no OpenGL ES, o qual é usado em navegadores de Internet na forma de WebGL, podemos desenhar figuras na tela passando índices dos vértices que usaremos na ordem correta (*glDrawElements*). Entretanto, o valor de zero é reservado para interromper a continuidade do desenho atual e por isso não é um índice válido. Por causa disso, lidamos com tal escolha de projeto questionável fazendo com que internamente nunca precisemos referenciar um vértice na posição zero. Futuramente, no próximo capítulo, esconderemos do usuário esta bagunça fazendo com que ele possa usar o zero para se referir à primeira posição e possa usar alguma macro para interromper o desenho. A API fará a tradução conforme necessário.

Caso não precisemos mais de uma classe de objeto básico, podemos querer removê-la. Alguns "buracos" podem se formar entre as classes por causa disso. Não podemos removê-los movendo as próximas classes porque cada classe é identificada pelo seu endereço na memória. Isso também nos impede de ordená-las. Entretanto, como o usuário tem controle sobre o número de classes suportadas (o tamanho do vetor de classes) e como o acesso à uma região contínua de memória é muito rápida, estima-se que isso não será um problema.

279 ⟨ Wobject: Declaração 277 ⟩ +≡

```
void _undefine_basic_object(union Wclass *wclass);
```

280 ¶⟨ Wobject: Definição 278 ⟩ +≡

```
void _undefine_basic_object(union Wclass *wclass)
{
  int i;      /* Localiza a classe: */
  for (i = 0; i < W_MAX_CLASSES; i++)
    if (&(_wclasses[i]) ≡ wclass) break;
  if (i ≥ W_MAX_CLASSES) return;
        /* Marca o espaço da classe como vazio: */
  _wclasses[i].basic.type = W_NONE;      /* Desaloca os vetores alocados: */
  Wfree(_wclasses[i].basic.vertices);
}
```

¶ Uma última coisa que iremos querer fazer com relação às definições de classes é evitar uma mensagem de vazamento de memória ao encerrar o programa. Um usuário pode tanto escolher desalocar manualmente as suas classes ou não. Caso ele não desaloque, quando o programa se encerrar, iremos desalocá-las automaticamente. Desta forma, quando encerrarmos o nosso gerenciador de memória, ele não encontrará memória não-desalocada na forma de vetores de vértices:

281 ⟨ API Weaver: Desalocações 281 ⟩ ≡

```
{
  int i;
  for (i = W_MAX_CLASSES − 1; i ≥ 0; i−−) {
    if (_wclasses[i].basic.type ≡ W_BASIC) {
      Wfree(_wclasses[i].basic.vertices);
      _wclasses[i].basic.type = W_NONE;
      continue;
    }
    ⟨ Desalocação Automática de Classes 359 ⟩
  }
}
```

This code is used in chunk 82.

¶ Ainda assim, a única forma de evitar mensagens que acusam memória desalocada na ordem errada é realmente desalocar manualmente a definição de classes.

7.2 Criando Instâncias de Objetos Básicos

Criar uma nova instância geralmente é fácil. Se existirem menos instâncias que o permitido, é só percorrer o vetor de instâncias de uma classe, encontrar um vazio e marcá-lo como não-vazio. Se tudo já estiver preenchido e a classe for essencial, então simplesmente retornamos Λ. O único caso mais complicado é quando tudo já está preenchido e estamos diante de uma classe não-essencial. Neste caso, percorremos todas as instâncias e decrementamos o seu número. A instâncias que ficar com um -1 é a mais antiga e será removida. Reinicializamos todos os seus valores. E ajustamos o seu número como sendo `W_MAX_INSTANCES` − 1:

283 ⟨ Wobject: Declaração 277 ⟩ +≡
```
union Wobject *_new_basic_object(union Wclass *wclass);
```

284 ¶⟨ Wobject: Definição 278 ⟩ +≡
```
union Wobject *_new_basic_object(union Wclass *wclass)
{
  int i;      /* Caso 1: Tem espaço pra mais um objeto */
  if (wclass→basic.number_of_objects < W_MAX_INSTANCES) {
    for (i = 0; i < W_MAX_INSTANCES; i++) {
      if (wclass→basic.instances[i].basic.type ≡ W_NONE) {
        wclass→basic.instances[i].basic.type = W_BASIC;
        wclass→basic.instances[i].basic.number =
            wclass→basic.number_of_objects;
        wclass→basic.number_of_objects++;
        return &(wclass→basic.instances[i]);
      }
    }
    return Λ;
  }     /* Caso 2: Não tem e é uma classe essencial */
  else if (wclass→basic.essential) return Λ;
        /* Caso 3: Não tem e não é uma classe essencial */
  else {
    int k, l;
    union Wobject *ptr;
    for (i = 0; i < W_MAX_INSTANCES; i++) {
      wclass→basic.instances[i].basic.number--;
      if (wclass→basic.instances[i].basic.number ≡ −1)
        wclass→basic.instances[i].basic.number = W_MAX_INSTANCES − 1;
      ptr = &(wclass→basic.instances[i]);
    }
    ptr→basic.x = 0;
    ptr→basic.y = 0;
    ptr→basic.z = 0;
    ptr→basic.angle_x = 0;
```

```
    ptr→basic.angle_y = 0;
    ptr→basic.angle_z = 0;
    ptr→basic.scale_x = 1.0;
    ptr→basic.scale_y = 1.0;
    ptr→basic.scale_z = 1.0;
    ptr→basic.visible = 1;
      /* Ininicializando as matrizes de rotação e translação: */
    for (k = 0; k < 4; k++)
      for (l = 0; l < 4; l++) {
        if (k ≡ l) {
          ptr→basic.rotation_x[k][l] = 1;
          ptr→basic.rotation_y[k][l] = 1;
          ptr→basic.rotation_z[k][l] = 1;
          ptr→basic.rotation_total[k][l] = 1;
          ptr→basic.translation[k][l] = 1;
          ptr→basic.scale_matrix[k][l] = 1;
          ptr→basic.model_matrix[k][l] = 1;
          ptr→basic.normal_matrix[k][l] = 1;
          ptr→basic.model_view_matrix[k][l] = 1;
        }
        else {
          ptr→basic.rotation_x[k][l] = 0;
          ptr→basic.rotation_y[k][l] = 0;
          ptr→basic.rotation_z[k][l] = 0;
          ptr→basic.rotation_total[k][l] = 0;
          ptr→basic.translation[k][l] = 0;
          ptr→basic.scale_matrix[k][l] = 0;
          ptr→basic.model_view_matrix[k][l] = 0;
          ptr→basic.model_matrix[k][l] = 0;
          ptr→basic.normal_matrix[k][l] = 0;
        }
      }
    return ptr;
  }
}
```

¶ Já destruir um objeto é algo um pouco mais direto. Marca-se o objeto como desalocado, decrementa o contador de objetos da classe e decrementa-se o número de todos os objetos da classe que tinham um número maior que o objeto destruído:

285 ⟨ Wobject: Declaração 277 ⟩ +≡

```
void _destroy_basic_object(union Wobject *wobj);
```

286 ¶⟨ Wobject: Definição 278 ⟩ +≡

```
void _destroy_basic_object(union Wobject *wobj)
{
  union Wclass *wclass;
  int number, i;
  wclass = wobj→basic.wclass;
  number = wobj→basic.number;
  wobj→basic.type = W_NONE;
  wclass→basic.number_of_objects--;
  for (i = 0; i < W_MAX_INSTANCES; i++) {
    if (wclass→basic.instances[i].basic.number > number)
      wclass→basic.instances[i].basic.number--;
  }
}
```

¶

7.3 Processando Objetos no Loop Princpal

Qundo estamos em um loop principal, temos que processar os objetos. Isso envolve desenhá-los na tela se forem visíveis e para objetos mais sofisticados, movê-los, realizar colisões e coisas assim. O modo de fazer isso é percorrer o vetor de classes e cada um de seus objetos e fazer as operações adequadas para cada um deles no loop principal:

288 ⟨ API Weaver: Loop Principal 47 ⟩ +≡

```
{
  int i, j;
  for (i = 0; i < W_MAX_CLASSES; i++) {
    switch (_wclasses[i].basic.type) {
    case W_NONE: continue;
    case W_BASIC:
      for (j = 0; j < W_MAX_INSTANCES; j++) {
        if (_wclasses[i].basic.instances[j].basic.type ≡ W_NONE) continue;
        ⟨ Transformação Linear de Objeto (i, j) 304 ⟩
        glVertexAttribPointer(_shader_vPosition, 3, GL_FLOAT, GL_FALSE,
            6 * sizeof(float), (void *) 0);
        glVertexAttribPointer(_shader_VertexNormal, 3, GL_FLOAT,
            GL_FALSE, 6 * sizeof(float), (void *)(sizeof(float) * 3));
        glEnableVertexAttribArray(_shader_vPosition);
        glEnableVertexAttribArray(_shader_VertexNormal);
        glBindVertexArray(_wclasses[i].basic._vertex_object);
            /* Note que abaixo ignoramos o primeiro vértice. Seu valor não
               deve ser usado conforme mencionado na definição de classe: */
        glDrawArrays(GL_POINTS, 1, _wclasses[i].basic.number_of_vertices);
      }
      continue;
      ⟨ Desenho de Objetos no Loop Principal 363 ⟩
    }
  }
}
```

¶

7.4 Escala de Objetos

Objetos podem ser esticados ou comprimidos ao longo dos eixos x, y e z. Se ele for esticado ou comprimido a mesma quantidade nos três eixos ele cresce ou encolhe mantendo a proporção. Caso contrário, ele sofre uma deformação. A possibilidade de podermos fazer esta transformação com ele é o motivo de cada objeto possuir valores *scale_x*, *scale_y* e *scale_z*, e também o de possuir uma matriz 4×4 chamada *scale_matrix*.

A matriz serve para representar a própria transformação linear que representa a escala de um objeto. Por exemplo, assumindo que queremos deixar um vetor $(x, y, z, 1)$ ao todo a vezes maior no eixo x, b vezes maior no eixo y e c vezes maior no eixo z, então podemos representar a transformação por meio da seguinte multiplicação de matrizes:

$$\begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} ax \\ by \\ cz \\ 1 \end{bmatrix}$$

A matriz será o que será passada para a GPU para o cálculo. Já os valores *scale_x*, *scale_y* e *scale_z* será mais útil para a CPU. Modificar a escala de um objeto pode ser feito então com o seguinte código:

290 ⟨ Wobject: Declaração 277 ⟩ +≡

```
void Wscale(union Wobject *wobj, float scale_x, float scale_y, float
    scale_z);
```

291 ¶⟨ Wobject: Definição 278 ⟩ +≡

```
void Wscale(union Wobject *wobj, float scale_x, float scale_y, float
        scale_z)
{
  wobj→basic.scale_x = scale_x;
  wobj→basic.scale_y = scale_y;
  wobj→basic.scale_z = scale_z;
  wobj→basic.scale_matrix[0][0] = scale_x;
  wobj→basic.scale_matrix[1][1] = scale_y;
  wobj→basic.scale_matrix[2][2] = scale_z;
  _regenerate_model_matrix(wobj);
}
```

¶ A última linha da função na qual invocamos a função ainda não definida *_regenerate_model_matrix* serve para que a matriz modelo de nosso objeto seja atualizada. Esta matriz representa a multiplicação de todas as matrizes que representam transformações lineares pelas quais nosso objeto irá passar. Sendo assim, toda vez que uma das matrizes do objeto for modificada, ela precisará ser

gerada novamente. Por representar a união de todas as transformações lineares de um objeto, essa é a matriz que realmente será passada para a GPU.

7.5 Translação de Objetos

A translação é usada para mover todos os pontos de um objeto no eixo XYZ. Ela é algo que ocorre para cada um dos vértices dentro da GPU durante o shader de vértice. Como é algo feito pela GPU, então é algo feito de modo mais eficiente se for expresso como uma multiplicação de matrizes. Para realizar uma translação de um ponto (x, y, z) em um espaço cartesiano tridimensional, movendo-o (a, b, c) posições, realizamos a seguinte multiplicação:

$$\begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x+a \\ y+b \\ z+c \\ 1 \end{bmatrix}$$

Como nós armazenamos esta matriz 4×4, nem mesmo seria necessário fazer com que os objetos tivessem atributos x, y e z. Tais variáveis existem só por questão de conveniência de acesso das coordenadas dos objetos.

Um função que realiza a translação de um objeto pode ser definida então da seguinte forma:

293 ⟨ Wobject: Declaração 277 ⟩ +≡
```
void Wtranslate(union Wobject *wobj, float x, float y, float z);
```

294 ¶⟨ Wobject: Definição 278 ⟩ +≡
```
void Wtranslate(union Wobject *wobj, float x, float y, float z)
{
  wobj→basic.x += x;
  wobj→basic.y += y;
  wobj→basic.z += z;
  wobj→basic.translation[0][3] += x;
  wobj→basic.translation[1][3] += y;
  wobj→basic.translation[2][3] += z;
  _regenerate_model_matrix(wobj);
}
```

¶

7.6 Rotação de Objetos

Rotacionar um objeto é girá-lo ao redor de um eixo que passa pelo seu próprio centro. Os eixos nos quais permitiremos rotação são o x, y e z. Como o objeto já está inicialmente centralizado em $(0, 0, 0)$, a matriz para rotacioná-lo em um ângulo θ no eixo x é:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y\ cos\theta - z\ sin\theta \\ y\ sin\theta + z\ cos\theta \\ 1 \end{bmatrix}$$

E no eixo y:

$$\begin{bmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x\ cos\theta + z\ sin\theta \\ y \\ -x\ sin\theta + z\ cos\theta \\ 1 \end{bmatrix}$$

E finalmente, no eixo z:

$$\begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x\ cos\theta - y\ sin\theta \\ x\ sin\theta + y\ cos\theta \\ z \\ 1 \end{bmatrix}$$

E para modificarmos estas matrizes, podemos então definir a função *Wrotate*, análoga à *Wtranslate*:

296 ⟨ Wobject: Declaração 277 ⟩ +≡

```
void Wrotate(union Wobject *wobj, float x, float y, float z);
```

297 ¶⟨ Wobject: Definição 278 ⟩ +≡

```
void Wrotate(union Wobject *wobj, float x, float y, float z)
{
  float aux[4][4];
  wobj→basic.angle_x += x;
  wobj→basic.angle_y += y;
  wobj→basic.angle_z += z;
  if (x ≠ 0) {
    wobj→basic.rotation_x[1][1] = cosf(wobj→basic.angle_x);
    wobj→basic.rotation_x[1][2] = sinf(wobj→basic.angle_x);
    wobj→basic.rotation_x[2][1] = -sinf(wobj→basic.angle_x);
    wobj→basic.rotation_x[2][2] = cosf(wobj→basic.angle_x);
  }
  if (y ≠ 0) {
    wobj→basic.rotation_y[0][0] = cosf(wobj→basic.angle_y);
    wobj→basic.rotation_y[0][2] = sinf(wobj→basic.angle_y);
```

```
        wobj→basic.rotation_y[2][0] = -sinf(wobj→basic.angle_y);
        wobj→basic.rotation_y[2][2] = cosf(wobj→basic.angle_y);
    }
    if (z ≠ 0) {
        wobj→basic.rotation_z[0][0] = cosf(wobj→basic.angle_z);
        wobj→basic.rotation_z[0][1] = -sinf(wobj→basic.angle_z);
        wobj→basic.rotation_z[1][0] = sinf(wobj→basic.angle_z);
        wobj→basic.rotation_z[1][1] = cosf(wobj→basic.angle_z);
    }       /* Multiplicamos agora as matrizes. Primeiro a rotação X pela Y: */
    _matrix_multiplication4x4(wobj→basic.rotation_x, wobj→basic.rotation_y,
            aux);       /* E depois multiplicamos o resultado por Z: */
    _matrix_multiplication4x4(aux, wobj→basic.rotation_z,
            wobj→basic.rotation_total);
        /* Por fim, atualizamos a matriz de modelo: */
    _regenerate_model_matrix(wobj);
}
```

¶ Definiremos a multiplicação de matrizes com outras funções auxiliares ao fim do capítulo.

7.7 A Matriz de Modelo

Tendo já definido as várias transformações lineares possíveis para um objeto, agora já podemos combinar todas elas em uma só matriz. Para isso, é só finalmente definirmos a função *_regenerate_model_matrix*. Ela envolve apenas a multiplicação de várias matrizes até obtermos a nossa matriz de modelo. A única coisa com a qual temos de nos preocupar é com a ordem das multiplicações. Os efeitos são diferentes dependendo de como multiplicamos as matrizes. A ordem que usaremos será:

$$v \times (T \times R \times S)$$

Onde v é o vértice dentro do shader, T é a translação, R é a rotação e S é a escala. A ordem é invertida devido à forma pela qual o vértice e as matrizes são multiplicadas. A translação fica mais próxima do vértice porque ela deve ser feita separadamente da rotação e da escala pelo fato de mudar a origem do nosso modelo do centro da figura para o centro do mundo no qual estamos. A rotação e a escala funcionam assumindo que a origem é o centro do objeto que elas transformam.

299 ⟨ Wobject: Declaração 277 ⟩ +≡
```
void _regenerate_model_matrix(union Wobject *wobj);
```

300 ¶⟨ Wobject: Definição 278 ⟩ +≡
```
void _regenerate_model_matrix(union Wobject *wobj)
{
  float aux[4][4];
  _matrix_multiplication4x4(wobj→basic.translation,
      wobj→basic.rotation_total, aux);
  _matrix_multiplication4x4(aux, wobj→basic.scale_matrix,
      wobj→basic.model_matrix);
  _regenerate_model_view_matrix(wobj);
}
```

¶ Note que toda vez que geramos novamente a matriz de modelo de um objeto, geramos novamente a sua matriz de modelo e visualização. A matriz de modelo e visualização tem tanto informações sobre os movimentos feitos sobre um objeto como sobre os movimentos feitos pela câmera. Por causa disso, esta é a matriz que nós realmente passamos para o shader.

Agora vamos declarar no shader de vértice a matriz de modelo e visualização que será modificada toda vez que formos renderizar um novo objeto:

301 ⟨ Shader de Vértice: Declarações 232 ⟩ +≡
```
uniform mat4 Wmodelview_matrix;
```

¶ Durante a inicialização o programa em C vai precisar obter a localização desta variável GLSL:

302 ⟨ API Weaver: Definições 149 ⟩ +≡
```
static GLfloat _shader_model_matrix;
```

303 ¶⟨ API Weaver: Inicialização 81 ⟩ +≡
```
{
  _shader_model_matrix = glGetUniformLocation(_program,
      "Wmodelview_matrix");
}
```

¶ O lugar de atualizar o valor desta matriz no programa em C é imediatamente antes de renderizar cada objeto. Atualizar esta matriz é realizar a transformação linear do objeto:

304 ⟨ Transformação Linear de Objeto (i, j) 304 ⟩ ≡
```
{
  float *p = (float *)
      &_wclasses[i].basic.instances[j].basic.model_view_matrix;
  glUniformMatrix4fv(_shader_model_matrix, 1, GL_FALSE, p);
}
```
This code is used in chunks 288 and 363.

¶ Dentro do shader de vértice aplicamos a matriz de modelo como sendo o primeiro tratamento para cada vértice:

305 ⟨ Shader de Vértice: Aplicar Matriz de Modelo 305 ⟩ ≡
```
gl_Position *= Wmodelview_matrix;
```
This code is used in chunk 226.

¶

7.8 Translação e Rotação da Câmera

Outra coisa que vamos precisar fazer é, além de mover objetos, mover também a câmera. Isso implica que será útil para nós armazenarmos a coordenada atual da câmera. Para isso definiremos um novo arquivo de código-fonte e declararemos as estruturas necessárias nele:

307 ⟨ `project/src/weaver/camera.h` 307 ⟩ ≡
```
#ifndef _camera_h_
#define _camera_h_
#ifdef __cplusplus
  extern "C" {
#endif
      ⟨ Inclui Cabeçalho de Configuração 42 ⟩
      ⟨ Câmera: Cabeçalho 310 ⟩
      ⟨ Câmera: Declaração 312 ⟩
#ifdef __cplusplus
      }
#endif
#endif
```

308 ¶⟨ Cabeçalhos Weaver 45 ⟩ +≡
```
#include "camera.h"
```

309 ¶⟨ `project/src/weaver/camera.c` 309 ⟩ ≡
```
#include "weaver.h"
  ⟨ Câmera: Definição 311 ⟩
```

¶ Assim como os objetos, a câmera também terá matrizes representando a transformação de translação e rotação (mas não escala). E também uma matriz que representa a união das outras transformações (a matriz de visualização):

310 ⟨ Câmera: Cabeçalho 310 ⟩ ≡
```
  extern float Wcamera_x, Wcamera_y, Wcamera_z;
  extern float Wcamera_angle_x, Wcamera_angle_y, Wcamera_angle_z;
  extern float _view_matrix[4][4];
```
This code is used in chunk 307.

311 ¶⟨ Câmera: Definição 311 ⟩ ≡
```
  float Wcamera_x, Wcamera_y, Wcamera_z;
  float Wcamera_angle_x, Wcamera_angle_y, Wcamera_angle_z;
  static float _camera_translation[4][4];
  static float _camera_rotation_x[4][4], _camera_rotation_y[4][4];
  static float _camera_rotation_z[4][4], _camera_rotation_total[4][4];
```

```
float _view_matrix[4][4];
```

See also chunks 313, 316, 318, and 320.

This code is used in chunk 309.

¶ Na inicialização da API Weaver inicializamos o valor da posição da câmera e inicializamos todas as matrizes. Definiremos uma função de inicialização de câmera para nos ajudar:

312 ⟨ Câmera: Declaração 312 ⟩ ≡

```
void _initialize_camera(void);
```

See also chunks 315, 317, and 319.

This code is used in chunk 307.

313 ¶⟨ Câmera: Definição 311 ⟩ +≡

```
void _initialize_camera(void)
{
  int i, j;
  Wcamera_x = Wcamera_y = Wcamera_z = 0.0;
  Wcamera_angle_x = Wcamera_angle_y = Wcamera_angle_z = 0.0;
  for (i = 0; i < 4; i++)
    for (j = 0; j < 4; j++)
      if (i ≡ j) {
        _camera_translation[i][j] = 1.0;
        _camera_rotation_x[i][j] = 1.0;
        _camera_rotation_y[i][j] = 1.0;
        _camera_rotation_z[i][j] = 1.0;
        _camera_rotation_total[i][j] = 1.0;
        _view_matrix[i][j] = 1.0;
      }
      else {
        _camera_translation[i][j] = 0.0;
        _camera_rotation_x[i][j] = 0.0;
        _camera_rotation_y[i][j] = 0.0;
        _camera_rotation_z[i][j] = 0.0;
        _camera_rotation_total[i][j] = 0.0;
        _view_matrix[i][j] = 0.0;
      }
}
```

314 ¶⟨ API Weaver: Inicialização 81 ⟩ +≡

```
_initialize_camera( );
```

¶ Agora quanto a realizar translação de câmeras e de objetos, as duas coisas são muito semelhantes. De fato, mover a câmera para a direita é equivalente a mover todos os objetos para a esquerda, e vice-versa. Portanto, caso a câmera sofre rotação, nós atualizamos a sua matriz de maneira idêntica. Só que com valores invertidos, pois tal matriz será depois multiplicada com a matriz de modelo de cada objeto para assim termos a matriz de modelo e visualização.

O nosso código de translação de câmera é:

315 ⟨ Câmera: Declaração 312 ⟩ +≡
```
void Wtranslate_camera(float x, float y, float z);
```

316 ¶⟨ Câmera: Definição 311 ⟩ +≡
```
void Wtranslate_camera(float x, float y, float z)
{
  Wcamera_x += x;
  Wcamera_y += y;
  Wcamera_z += z;
  _camera_translation[0][3] = -Wcamera_x;
  _camera_translation[1][3] = -Wcamera_y;
  _camera_translation[2][3] = -Wcamera_z;
  _regenerate_view_matrix();
}
```

¶ Assim como fizemos na definição de transformação d eobjetos, definiremos posteriormente a função *_regenerate_view_matrix*.

A rotação da câmera envolve girar todos os demais objetos ao redor da câmera no sentido inverso do pedido. Para isso, basta simplesmente rotacionarmos os objetos depois que as suas coordenadas estiverem com a origem onde está a câmera.

A função de rotacionar a câmera então é semelhante à rotação de um objeto e envolve modificar as matrizes relacionadas à câmera. Com a diferença de que invertemos os ângulos antes de passarmos para a matriz:

317 ⟨ Câmera: Declaração 312 ⟩ +≡
```
void Wrotate_camera(float x, float y, float z);
```

318 ¶⟨ Câmera: Definição 311 ⟩ +≡
```
void Wrotate_camera(float x, float y, float z)
{
  float aux[4][4];
  Wcamera_angle_x -= x;
  Wcamera_angle_y -= y;
  Wcamera_angle_z -= z;
  if (x ≠ 0) {
```

```
    _camera_rotation_x[1][1] = cosf(Wcamera_angle_x);
    _camera_rotation_x[1][2] = -sinf(Wcamera_angle_x);
    _camera_rotation_x[2][1] = sinf(Wcamera_angle_x);
    _camera_rotation_x[2][2] = cosf(Wcamera_angle_x);
  }
  if (y ≠ 0) {
    _camera_rotation_y[0][0] = cosf(Wcamera_angle_y);
    _camera_rotation_y[0][2] = sinf(Wcamera_angle_y);
    _camera_rotation_y[2][0] = -sinf(Wcamera_angle_y);
    _camera_rotation_y[2][2] = cosf(Wcamera_angle_y);
  }
  if (z ≠ 0) {
    _camera_rotation_z[0][0] = cosf(Wcamera_angle_z);
    _camera_rotation_z[0][1] = -sinf(Wcamera_angle_z);
    _camera_rotation_z[1][0] = sinf(Wcamera_angle_z);
    _camera_rotation_z[1][1] = cosf(Wcamera_angle_z);
  }     /* Multiplicamos agora as matrizes. Primeiro a rotação X pela Y: */
  _matrix_multiplication4x4(_camera_rotation_x, _camera_rotation_y, aux);
     /* E depois multiplicamos o resultado por Z: */
  _matrix_multiplication4x4(aux, _camera_rotation_z, _camera_rotation_total);
  _regenerate_view_matrix();
}
```

¶ Agora enfim iremos definir a função para gerar novamente a matriz de visualização toda vez que a câmera sofrer rotação e translação. Ela é basicamente uma multiplicação das matrizes de rotação e translação. Mas além disso, toda vez que modificamos esta matriz, precisamos também percorrer todos os objetos e gerar novamente a sua matriz de modelo e visualização.

319 ⟨ Câmera: Declaração 312 ⟩ +≡

```
void _regenerate_view_matrix(void);
```

320 ¶⟨ Câmera: Definição 311 ⟩ +≡

```
void _regenerate_view_matrix(void)
{
  int i, j;
  _matrix_multiplication4x4(_camera_translation, _camera_rotation_total,
      _view_matrix);
  for (i = 0; i < W_MAX_CLASSES; i++)
    for (j = 0; j < W_MAX_INSTANCES; j++)
      _regenerate_model_view_matrix(&_wclasses[i].basic.instances[j]);
}
```

¶ E agora por fim definimos a função que gera novamente a matriz de modelo e visualização para cada objeto, a qual funciona simplesmente multiplicando as matrizes de modelo e visualização. O único detalhe adicional que fazemos aqui também é atualizar a matriz normal do objeto, a qual é útil para calcularmos a rotação e translação dos efeitos de luz e sombra do objeto. A matriz normal de um objeto é a transposta da inversa da matriz de modelo-visualização:

321 ⟨ Wobject: Declaração 277 ⟩ +≡
```
void _regenerate_model_view_matrix(union Wobject *wobj);
```

322 ¶⟨ Wobject: Definição 278 ⟩ +≡
```
void _regenerate_model_view_matrix(union Wobject *wobj)
{
  int i, j;
  _matrix_multiplication4x4(_view_matrix, wobj→basic.model_matrix,
      wobj→basic.model_view_matrix);
  for (i = 0; i < 4; i++)
    for (j = 0; j < 4; j++)
      wobj→basic.normal_matrix[i][j] = wobj→basic.model_view_matrix[i][j];
  _matrix_inverse4x4(wobj→basic.normal_matrix);
  _matrix_transpose4x4(wobj→basic.normal_matrix);
}
```

¶ As funções de inverter e transpor matrizes 4×4 serão definidas ao fim do capítulo.

7.9 A Projeção de Objetos

Após realizar todas as transformações necessárias sobre um objeto, colocarmos ele em sua posição relativa em relação à câmera, a última coisa que temos a fazer é definir como será feita a projeção de seus pontos na tela. Existem muitos tipos de projeção diferentes. A mais comum é a projeção em perspectiva, que tenta imitar mais fielmente a visão humana fazendo com que objetos mais distantes pareçam menores. Alguns jogos, por outro lado, baseiam-se em uma projeção ortográfica, onde objetos mais distantes não ficam menores (Sim City, por exemplo). Podem haver muitas outras formas de projeção para criar diferentes tipos de efeitos visuais. O jogo **Animal Crossing: New Leaf**, por exemplo, possui uma projeção peculiar que faz com que o espaço em si tenha uma curvatura cilíndrica.

Independente da projeção, assumimos que no ponto $(0, 0, 0)$ está a nossa câmera. Na visão em perspectiva temos uma visão piramidal, onde a ponta da pirâmide fica bem no seu ponto focal $(0, 0, 0)$, e a base da pirâmide é um quadrado projetado em algum ponto distante. A pirâmide pode ser cortada em qualquer ponto do eixo z e assim obtemos um quadrado. A proporção de um objeto na tela é a proporção dele em relação ao quadrado obtido seccionando a nossa pirâmide no eixo z na mesma posição em que o objeto está. Desta forma, quanto mais próximo um objeto estiver do nosso ponto focal, maior ele será, e quanto mais distante estiver, menor ele parecerá. Na visão ortogonal, a nossa região de visão simplesmente é um cuboide. A proporção ocupada na tela por um objeto então é sempre a mesma, independente da distância.

Entretanto, dependendo da projeção, não poderemos representar objetos próximos demais de nosso ponto focal. Na visão em perspectiva, à medida que um objeto se aproxima dela, o seu tamanho tenderá ao infinito. Deve existir então uma distância mínima que um objeto deve estar para ser representado (o plano próximo). E independente da projeção não podemos ficar representando objetos distantes demais. Se algo está longe demais, geralmente não tem tanta relevância para a cena. Então será um desperdício ficarmos renderizando ele, principalmente se ele for formado por muitos polígonos. Além do mais, como o buffer z usado para detectar quais objetos estão na frente dos outros tem uma precisão de apenas 8 bits, podemos acabar perdendo a precisão desta noção quando objetos estão distantes demais. por isso, se um objeto está além de um ponto no eixo z (o plano distante), ele também não será renderizado.

Isso faz com que precisemos de 3 valores diferentes que precisam ser configurados. Primeiro a menor distância da câmera que um objeto pode estar para ser detectado (`W_NEAR_PLANE`, ou Z_{near}), a máxima distância que a câmera pode captar (`W_FAR_PLANE`, ou Z_{far}) e também o tamanho máximo que um quadrado deve ter para ser visto por inteiro quando está na menor distância possível da câmera (`W_CAMERA_SIZE`, ou n). Estes três valores devem estar definidos e ser configurados no `conf/conf.h`.

Tendo tais valores, o método de se obter a projeção em perspectiva é multiplicando os vetores pela seguinte matriz:

$$\begin{bmatrix} \frac{Z_{near}}{n/2} & 0 & 0 & 0 \\ 0 & \frac{Z_{near}}{n/2} & 0 & 0 \\ 0 & 0 & -\frac{Z_{far}+Z_{near}}{Z_{far}-Z_{near}} & \frac{-2Z_{far}Z_{near}}{Z_{far}-Z_{near}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

E para obtermos uma projeção ortográfica, usamos a seguinte matriz:

$$\begin{bmatrix} \frac{1}{n/2} & 0 & 0 & 0 \\ 0 & \frac{1}{n/2} & 0 & 0 \\ 0 & 0 & -\frac{1}{2(Z_{far}-Z_{near})} & -\frac{Z_{far}+Z_{near}}{Z_{far}-Z_{near}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Qual destas matrizes iremos usar? Isso também é algo que deve ser configurável no `conf/conf.h`. Vamos definir um significado para as macros `W_PERSPECTIVE` e `W_ORTHOGONAL` que poderão ser usadas neste arquivo:

324 ⟨ `project/src/weaver/conf_begin.h` 43 ⟩ +≡

```
#define W_PERSPECTIVE 2
#define W_ORTHOGONAL 3
```

¶ Ambos os valores podem ser definidos para a macro `W_PROJECTION` no `conf/conf.h`

Como a matriz de projeção é inicializada só no começo do programa e nunca mais é mudada, vamos declará-la como estática no mesmo arquivo onde está a função de inicialização, e na inicialização aplicamos os valores:

325 ⟨ API Weaver: Definições 149 ⟩ +≡

```
static float _projection_matrix[4][4];
```

326 ¶⟨ API Weaver: Inicialização 81 ⟩ +≡

```
{
  int i, j;
  for (i = 0; i < 4; i++)
    for (j = 0; j < 4; j++) _projection_matrix[i][j] = 0.0;
        /* Inicializando os valores diferentes de 0: */
#if W_PROJECTION ≡ W_PERSPECTIVE
  _projection_matrix[0][0] = W_NEAR_PLANE/(W_CAMERA_SIZE/2);
  _projection_matrix[1][1] = W_NEAR_PLANE/(W_CAMERA_SIZE/2);
  _projection_matrix[2][2] = -(W_FAR_PLANE+W_NEAR_PLANE)/(W_FAR_PLANE-
      W_NEAR_PLANE);
  _projection_matrix[2][3] = (-2.0 * W_FAR_PLANE *
      W_NEAR_PLANE)/(W_FAR_PLANE - W_NEAR_PLANE);
  _projection_matrix[3][2] = -1.0;
#elif W_PROJECTION ≡ W_ORTHOGONAL
  _projection_matrix[0][0] = 1.0/(W_CAMERA_SIZE/2);
```

```
    _projection_matrix[1][1] = 1.0/(W_CAMERA_SIZE/2);
    _projection_matrix[2][2] = −1.0/((W_FAR_PLANE − W_NEAR_PLANE)/2.0);
    _projection_matrix[2][3] = −(W_FAR_PLANE+W_NEAR_PLANE)/(W_FAR_PLANE−
        W_NEAR_PLANE);
    _projection_matrix[3][3] = 1.0;
#endif
  }
```

¶ Mas não basta apenas termos a matriz. Nós precisamos também informar ao OpenGL a posição de `W_FAR_PLANE` e `W_NEAR_PLANE` para que o servidor possa ignorar os objetos que estiverem fora do alcance da câmera por estarem muito longe ou muito perto. Isso é feito invocando na inicialização a seguinte função:

327 ⟨ API Weaver: Inicialização 81 ⟩ +≡

```
  {
    glDepthRangef(W_NEAR_PLANE, W_FAR_PLANE);
  }
```

¶ Agora o shader precisa estar ciente da nova matriz que usaremos:

328 ⟨ Shader de Vértice: Declarações 232 ⟩ +≡

```
  uniform mat4 Wprojection_matrix;
```

¶ E precisamos inicializar tal matriz no shader durante a inicialização do programa. E também precisamos da variável do programa que vai armazenar a localização de tal matriz dentro do shader:

329 ⟨ API Weaver: Inicialização 81 ⟩ +≡

```
  {
    GLuint _shader_projection_address;
    float *ptr = (float *) &_projection_matrix;
    _shader_projection_address = glGetUniformLocation(_program,
        "Wprojection_matrix");
    glUniformMatrix4fv(_shader_projection_address, 1, GL_FALSE, ptr);
  }
```

¶ Por fim, usaremos tal matriz dentro do Shader multiplicando cada um dos vértices por ela:

330 ⟨ Shader de Vértice: Câmera (Perspectiva) 330 ⟩ ≡

```
  gl_Position *= Wprojection_matrix;
```

This code is used in chunk 226.

¶

7.10 Funções Auxiliares

Vamos definir um arquivo que irá conter funções auxiliares:

332 ⟨ project/src/weaver/aux.h 332 ⟩ ≡
```
#ifndef _aux_h_
#define _aux_h_
#ifdef __cplusplus
  extern "C" {
#endif
      ⟨ Inclui Cabeçalho de Configuração 42 ⟩
      ⟨ Funções Auxiliares: Declaração 336 ⟩
#ifdef __cplusplus
      }
#endif
#endif
```

333 ¶⟨ project/src/weaver/aux.c 333 ⟩ ≡
```
#include "weaver.h"
  ⟨ Funções Auxiliares: Definição 337 ⟩
```

334 ¶⟨ Cabeçalhos Weaver 45 ⟩ +≡
```
#include "aux.h"
```

¶

7.10.1 Multiplicação de Matrizes 4×4

E a nossa multiplicação de matrizes 4x4 será a primeira função que irá para tal arquivo:

336 ⟨ Funções Auxiliares: Declaração 336 ⟩ ≡
```
  void _matrix_multiplication4x4 (float a[4][4], float b[4][4], float result[4][4]);
```
See also chunks 339, 342, 345, 366, and 371.

This code is used in chunk 332.

337 ¶⟨ Funções Auxiliares: Definição 337 ⟩ ≡
```
  void _matrix_multiplication4x4 (float a[4][4], float b[4][4], float result[4][4])
  {
    int i, j, k;
```

```
    for (i = 0; i < 4; i++)
      for (j = 0; j < 4; j++) {
        result[i][j] = 0;
        for (k = 0; k < 4; k++) {
          result[i][j] += a[i][k] * b[k][j];
        }
      }
  }
```

See also chunks 340, 343, 346, 367, and 372.

This code is used in chunk 333.

¶

7.10.2 Calcular a Inversa de Matrizes 4×4

Como estamos querendo calcular a inversa apenas de matrizes 4×4 e não de outros tamanhos, podemos apenas usar uma fórmula "hard-coded" que apesar de feia é testada pelo tempo e irá funcionar:

339 ⟨ Funções Auxiliares: Declaração 336 ⟩ +≡

```
  void _matrix_inverse4x4(float m[4][4]);
```

340 ¶⟨ Funções Auxiliares: Definição 337 ⟩ +≡

```
  void _matrix_inverse4x4(float m[4][4])
  {
    float aux[4][4];
    float multiplier;
    int i, j;
    for (i = 0; i < 4; i++)
      for (j = 0; j < 4; j++) aux[i][j] = m[i][j];
    multiplier = 1.0/_matrix_determinant4x4(m);
      /* m[0][0] = aux[1][1] * aux[2][2] * aux[3][3] + aux[1][2] * aux[2][3] *
       aux[3][1] + aux[1][3] * aux[2][1] * aux[3][2]; m[0][0] -= aux[1][1] *
       aux[2][3] * aux[3][2] - aux[1][2] * aux[2][1] * aux[3][3] - aux[1][3] *
       aux[2][2] * aux[3][1]; m[0][1] = aux[0][1] * aux[2][3] * aux[3][2] +
       aux[0][2] * aux[2][1] * aux[3][3] + aux[0][3] * aux[2][2] * aux[3][1];
       m[0][1] -= aux[0][1] * aux[2][2] * aux[3][3] - aux[0][2] * aux[2][3] *
       aux[3][1] - aux[0][3] * aux[2][1] * aux[3][2]; m[0][2] = aux[0][1] *
       aux[1][2] * aux[3][3] + aux[0][2] * aux[1][3] * aux[3][1] + aux[0][3]
       * aux[1][1] * aux[3][2]; m[0][2]-= aux[0][1] * aux[1][3] * aux[3][2] -
       aux[0][2] * aux[1][1] * aux[3][3] - aux[0][3] * aux[1][2] * aux[3][1];
       m[0][3] = aux[0][1] * aux[1][3] * aux[2][2] + aux[0][2] * aux[1][1] *
       aux[2][3] + aux[0][3] * aux[1][2] * aux[2][1]; m[0][3] -= aux[0][1] *
       aux[1][2] * aux[2][3] - aux[0][2] * aux[1][3] * aux[2][1] - aux[0][3] *
```

```
aux[1][1] * aux[2][2]; m[1][0] = aux[1][0] * aux[2][3] * aux[3][2] +
aux[1][2] * aux[2][0] * aux[3][3] + aux[1][3] * aux[2][2] * aux[3][0];
m[1][0] -= aux[1][0] * aux[2][2] * aux[3][3] - aux[1][2] * aux[2][3] *
aux[3][0] - aux[1][3] * aux[2][0] * aux[3][2]; m[1][1] = aux[0][0] *
aux[2][2] * aux[3][3] + aux[0][2] * aux[2][3] * aux[3][0] + aux[0][3]
* aux[2][0] * aux[3][2]; m[1][1] -= aux[0][0] * aux[2][3] * aux[3][2] -
aux[0][2] * aux[2][0] * aux[3][3] - aux[0][3] * aux[2][2] * aux[3][0];
m[1][2] = aux[0][0] * aux[1][3] * aux[3][2] + aux[0][2] * aux[1][0] *
aux[3][3] + aux[0][3] * aux[1][2] * aux[3][0]; m[1][2] -= aux[0][0] *
aux[1][2] * aux[3][3] - aux[0][2] * aux[1][3] * aux[3][0] - aux[0][3] *
aux[1][0] * aux[3][2]; m[1][3] = aux[0][0] * aux[1][2] * aux[2][3] +
aux[0][2] * aux[1][3] * aux[2][0] + aux[0][3] * aux[1][0] * aux[2][2] -
aux[0][0] * aux[1][3] * aux[2][2] - aux[0][2] * aux[1][0] * aux[2][3] -
aux[0][3] * aux[1][2] * aux[2][0]; m[2][0] = aux[1][0] * aux[2][1] *
aux[3][3] + aux[1][1] * aux[2][3] * aux[3][0] + aux[1][3] * aux[2][0]
* aux[3][1] - aux[1][0] * aux[2][3] * aux[3][1] - aux[1][1] * aux[2][0]
* aux[3][3] - aux[1][3] * aux[2][1] * aux[3][0]; m[2][1] = aux[0][0] *
aux[2][3] * aux[3][1] + aux[0][1] * aux[2][0] * aux[3][3] + aux[0][3]
* aux[2][1] * aux[3][0] - aux[0][0] * aux[2][1] * aux[3][3] - aux[0][1]
* aux[2][3] * aux[3][0] - aux[0][3] * aux[2][0] * aux[3][1]; m[2][2] =
aux[0][0] * aux[1][1] * aux[3][3] + aux[0][1] * aux[1][3] * aux[3][0] +
aux[0][3] * aux[1][0] * aux[3][1] - aux[0][0] * aux[1][3] * aux[3][1] -
aux[0][1] * aux[1][0] * aux[3][3] - aux[0][3] * aux[1][1] * aux[3][0];
m[2][3] = aux[0][0] * aux[1][3] * aux[2][1] + aux[0][1] * aux[1][0] *
aux[2][3] + aux[0][3] * aux[1][1] * aux[2][0] - aux[0][0] * aux[1][1] *
aux[2][3] - aux[0][1] * aux[1][3] * aux[2][0] - aux[0][3] * aux[1][0] *
aux[2][1]; m[3][0] = aux[1][0] * aux[2][2] * aux[3][1] + aux[1][1] *
aux[2][0] * aux[3][2] + aux[1][2] * aux[2][1] * aux[3][0]; m[3][0] -=
aux[1][0] * aux[2][1] * aux[3][2] - aux[1][1] * aux[2][2] * aux[3][0] -
aux[1][2] * aux[2][0] * aux[3][1]; m[3][1] = aux[0][0] * aux[2][1] *
aux[3][2] + aux[0][1] * aux[2][2] * aux[3][0] + aux[0][2] * aux[2][0]
* aux[3][1] - aux[0][0] * aux[2][2] * aux[3][1] - aux[0][1] * aux[2][0]
* aux[3][2] - aux[0][2] * aux[2][1] * aux[3][0]; m[3][2] = aux[0][0] *
aux[1][2] * aux[3][1] + aux[0][1] * aux[1][0] * aux[3][2] + aux[0][2]
* aux[1][1] * aux[3][0] - aux[0][0] * aux[1][1] * aux[3][2] - aux[0][1]
* aux[1][2] * aux[3][0] - aux[0][2] * aux[1][0] * aux[3][1]; m[3][3] =
aux[0][0] * aux[1][1] * aux[2][2] + aux[0][1] * aux[1][2] * aux[2][0] +
aux[0][2] * aux[1][0] * aux[2][1] - aux[0][0] * aux[1][2] * aux[2][1] -
aux[0][1] * aux[1][0] * aux[2][2] - aux[0][2] * aux[1][1] * aux[2][0]; for(i
= 0; i ¡ 4; i ++) for(j = 0; j ¡ 4; j ++) m[i][j] *= multiplier;*/
}
```

¶

7.10.3 Calcular o Determinante de Matrizes 4×4

Seguido a mesma lógica de usarmos código feio, mas rápido e testado pelo tempo, programaremos a função que retorna o determinante de matrizes 4×4:

342 ⟨ Funções Auxiliares: Declaração 336 ⟩ +≡
```
float _matrix_determinant4x4(float m[4][4]);
```

343 ¶⟨ Funções Auxiliares: Definição 337 ⟩ +≡
```
float _matrix_determinant4x4(float m[4][4])
{
  return m[0][3]*m[1][2]*m[2][1]*m[3][0]-m[0][2]*m[1][3]*m[2][1]*m[3][0]-
      m[0][3]*m[1][1]*m[2][2]*m[3][0]+m[0][1]*m[1][3]*m[2][2]*m[3][0]+
      m[0][2]*m[1][1]*m[2][3]*m[3][0]-m[0][1]*m[1][2]*m[2][3]*m[3][0]-
      m[0][3]*m[1][2]*m[2][0]*m[3][1]+m[0][2]*m[1][3]*m[2][0]*m[3][1]+
      m[0][3]*m[1][0]*m[2][2]*m[3][1]-m[0][0]*m[1][3]*m[2][2]*m[3][1]-
      m[0][2]*m[1][0]*m[2][3]*m[3][1]+m[0][0]*m[1][2]*m[2][3]*m[3][1]+
      m[0][3]*m[1][1]*m[2][0]*m[3][2]-m[0][1]*m[1][3]*m[2][0]*m[3][2]-
      m[0][3]*m[1][0]*m[2][1]*m[3][2]+m[0][0]*m[1][3]*m[2][1]*m[3][2]+
      m[0][1]*m[1][0]*m[2][3]*m[3][2]-m[0][0]*m[1][1]*m[2][3]*m[3][2]-
      m[0][2]*m[1][1]*m[2][0]*m[3][3]+m[0][1]*m[1][2]*m[2][0]*m[3][3]+
      m[0][2]*m[1][0]*m[2][1]*m[3][3]-m[0][0]*m[1][2]*m[2][1]*m[3][3]-
      m[0][1]*m[1][0]*m[2][2]*m[3][3]+m[0][0]*m[1][1]*m[2][2]*m[3][3];
}
```

¶

7.10.4 Calcular a Transposição de Matrizes 4×4

Transpor uma matriz é só trocar as coordenadas de linhas e colunas de cada valor:

345 ⟨ Funções Auxiliares: Declaração 336 ⟩ +≡
```
void _matrix_transpose4x4(float m[4][4]);
```

346 ¶⟨ Funções Auxiliares: Definição 337 ⟩ +≡
```
void _matrix_transpose4x4(float m[4][4])
{
  int i, j;
  float aux[4][4];
  for (i = 0; i < 4; i++)
    for (j = 0; j < 4; j++) aux[i][j] = m[i][j];
  for (i = 0; i < 4; i++)
    for (j = 0; j < 4; j++) m[i][j] = aux[j][i];
}
```

¶

Capítulo 8

Formas Geométricas

Vamos agora definir um tipo de objeto que possui seus vértices relativamente fixos em relação uns aos outros (pode ser que eles tenham estruturas especiais que simulem movimento de ossos, mas mesmo assim eles tem uma forma padrão fixa e cujo movimento, se existir, sempre será algo bem-definido). Além disso, este tipo de objeto, quando visualizado, possui arestas e faces. Não apenas vértices.

Internamente, representaremos uma forma geométrica simples, que possui apenas as características listadas, pelo número 2:

348 ⟨ Wobject: Cabeçalho 268 ⟩ +≡ /∗ Tipo de Wobject: ∗/
#**define** `W_SHAPE` 2

¶ Mas podem haver outros tipos de formas geométricas. Todas elas sempre serão um múltiplo de 2. Sendo assim, serão fáceis de serem identificadas:

349 ⟨ Wobject: Cabeçalho 268 ⟩ +≡
#**define** *Wis_shape*(*wobj*) ((*wobj*→*type* % 2) ≡ 0)

¶ As classes que são formas geométricas são iguais às classes de objetos. A diferença é que elas possuem também como atributo interno uma lista de índices que especifica a ordem de cada vértice em cada face do polígono que será desenhado. Usa-se o número `#FFFF` para separar os dados de uma face da outra. E o índice de cada vértice é a ordem na qual ele aparece na lista de vértices. Podemos nos referir ao número mágico de separar faces por `W_FACE_BREAK`:

350 ⟨ Wobject: Cabeçalho 268 ⟩ +≡
#**define** `W_FACE_BREAK` `#ffff`

¶ Para que este valor possa ser usado como separador de índices de vértices, durante a inicialização do programa precisamos chamar a seguinte função:

351 ⟨ API Weaver: Inicialização 81 ⟩ +≡
```
{
  glEnable(GL_PRIMITIVE_RESTART);
}
```

¶ Além disso, precisamos também de uma variável para armazenar o ID de um buffer de índices de elementos OpenGL, que será a ordem na qual cada vértice será percorrido na hora de desenhar.

Todas as faces de polígonos desenhados sempre deverão ser convexas. Faces côncavas só poderão ser desenhadas caso sejam tratadas como duas ou mais faces. Essa restrição simplifica muito o algoritmo de desenho. Tudo o que precisamos fazer é pedir para o servidor OpenGL desenhar triângulos e passar os vértices na ordem que temos. E assim obtemos a face desejada:

352 ⟨ Wobject: Tipo de Classe 272 ⟩ +≡
```
struct {      /* Típico de objetos: */
  int type;
  int number_of_objects;
  int number_of_vertices;
  int essential;
  float *vertices;
  GLuint _vertex_object, _buffer_object;
  float width, height, depth;
  union Wobject instances[W_MAX_INSTANCES];
     /* Específico de Formas Geométricas: */
  int number_of_indices;
  GLushort * indices;
  GLuint _element_object;
} shape;
```

¶ Já instâncias de tais classes seriam idênticas às instâncias de objetos básicos. Não seria necessário nenhum atributo adicional:

353 ⟨ Wobject: Tipo de Objeto 273 ⟩ +≡
```
struct {      /* Geral de Todos os Objetos: */
  int type;
  int number;
  int visible;
  float x, y, z;
  float scale_x, scale_y, scale_z;
  float translation[4][4];
  float angle_x, angle_y, angle_z;
  float rotation_x[4][4], rotation_y[4][4], rotation_z[4][4];
  float rotation_total[4][4];
  float scale_matrix[4][4];
```

```
    float model_matrix[4][4];
    float model_view_matrix[4][4];
    union Wclass *wclass;
  } shape;
```

¶

8.1 Definindo Classe e Criando Instâncias

Definir uma forma geométrica manualmente é como definir um objeto básico. Só precisamos depois preencher a ordem dos vértices de acordo com a especificação de cada face:

355 ⟨ Wobject: Declaração 277 ⟩ +≡
```
union Wclass *define_shape(int number_of_vertices, float *vertices, int
    number_of_faces, unsigned int *faces);
```

356 ¶⟨ Wobject: Definição 278 ⟩ +≡
```
union Wclass *define_shape(int number_of_vertices, float *vertices, int
      number_of_faces, unsigned int *faces) { int count = 0,
      number_of_indices = 0, i;
    /* Will store the number of adjacent faces for each vertex. Later
      will be used to compute the vertex normal: */
  int *number_of_adjacent_faces;
  union Wclass *new_class = _define_basic_object(number_of_vertices,
      vertices);
  if (new_class ≡ Λ) return Λ;
  new_class→shape.type = W_SHAPE;
    /* Obtendo o número de índices: */
  for (number_of_indices = 0; count < number_of_faces;
      number_of_indices++) {
    if (faces[number_of_indices] ≡ W_FACE_BREAK) count++;
  }
  new_class→shape.number_of_indices = number_of_indices;
    /* Alocando os índices: */
  new_class→shape.indices = ( GLushort * ) Walloc(sizeof
      (GLushort) * number_of_indices);
  if (new_class→shape.indices ≡ Λ) {
    _undefine_basic_object(new_class);
    return Λ;
  }    /* Alocando a contagem do número de faces adjacentes por
      vértice: */
  number_of_adjacent_faces = (int *)
      Walloc(sizeof(int) * number_of_vertices + 1);
  if (number_of_adjacent_faces ≡ Λ) {
    Wfree(new_class→shape.indices);
    _undefine_basic_object(new_class);
    return Λ;
  }    /* inicializando a contagem de faces adjacentes: */
  for (i = 0; i < number_of_vertices; i++)
    number_of_adjacent_faces[i] = 0;
      /* Inicializando os índices e a contagem de faces adjacentes: */
```

```
{     /* A normal da face atual que percorremos ficará armazenada na
         variável abaixo. Usaremos a informação para calcular a normal
         de cada vértice para o cálculo de iluminação. */
  float normal[3];
     /* Começamos o cálculo do vetor normal à primeira face: */

  if (number_of_faces > 0) {
       /* Passamos sempre três vértices consecutivos da face para o
          cálculo do seu vetor normal. Podemos então assumir estarmos
          calculando a normal de um triângulo: */
    _normal_vector_to_triangle(&normal[0], &vertices[faces[0] * 3],
        &vertices[faces[1] * 3], &vertices[faces[2] * 3]);
  }
  for (i = 0; i < number_of_indices; i++) {
    if (faces[i] ≡ W_FACE_BREAK) {
      new_class→shape.indices[i] = (GLushort)0;
         /* Se existe uma próxima face, calculamos a sua normal: */
      if (i + 1 < number_of_indices)
        _normal_vector_to_triangle(&normal[0], &vertices[faces[i+1]*3],
            &vertices[faces[i + 2] * 3], &vertices[faces[i + 3] * 3]);
    }
    else {
      new_class→shape.indices[i] = (GLushort)faces[i] + 1;
         /* Incrementamos a contagem de faces do vértice encontrado:
          */
      number_of_adjacent_faces[faces[i] + 1]++;
         /* Somamos a normal atual ao vetor associado ao vértice: */
      new_class→shape.vertices[(faces[i] + 1) * 6 + 3] += normal[0];
      new_class→shape.vertices[(faces[i] + 1) * 6 + 4] += normal[1];
      new_class→shape.vertices[(faces[i] + 1) * 6 + 5] += normal[2];
    }
  }
}     /* Agora uma iteração para percorrermos os vetores associados a
         cada vértice e dividirmos ele pelo número de faces que contém o
         vértice. Por fim, os normallizamos. E assim teremos terminado
         de calcular a normal de cada vértice: */
for (i = 0; i < number_of_vertices; i++) {
  new_class→shape.vertices[(i+1)*6+3] /= number_of_adjacent_faces[i];
  new_class→shape.vertices[(i+1)*6+4] /= number_of_adjacent_faces[i];
  new_class→shape.vertices[(i+1)*6+5] /= number_of_adjacent_faces[i];
  _normalize(&new_class→shape.vertices[(i + 1) * 6 + 3]);
}     /* Agora podemos enviar os vértices para o servidor openGL: */
glBindVertexArray(new_class→shape._vertex_object);
glBindBuffer(GL_ARRAY_BUFFER, new_class→shape._buffer_object);
glBufferData(GL_ARRAY_BUFFER,
    sizeof(float) * 6 * (number_of_vertices + 1),
```

```
    new_class→shape.vertices, GL_STATIC_DRAW);
  /* Inicializando a lista de índices no servidor OpenGL: */
glGenBuffers(1, &(new_class→shape._element_object));
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,
    new_class→shape._element_object);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, number_of_indices * sizeof
    (GLushort), new_class→shape.indices, GL_STATIC_DRAW);
  /* Limpeza: */
Wfree(number_of_adjacent_faces);
return new_class; }
```

¶ As funções auxiliares para calcular a normal de um triângulo e normalizar um vetor serão apresentadas só ao fim deste capítulo.

Remover a definição de forma geométrica também é como remover a definição de um objeto básico. Novamente, só temos antes que desalocar os índices de cada vértice:

357 ⟨ Wobject: Declaração 277 ⟩ +≡
```
void undefine_shape(union Wclass *wclass);
```

358 ¶⟨ Wobject: Definição 278 ⟩ +≡
```
void _undefine_shape(union Wclass *wclass)
{
  Wfree(wclass→shape.indices);
  _undefine_basic_object(wclass);
}
```

¶ Da mesma forma, para que não seja estritamente necessário remover definições de classes, ao término do programa nós desalocamos as formas básicas que encontramos ainda definidas:

359 ⟨ Desalocação Automática de Classes 359 ⟩ ≡
```
if (_wclasses[i].basic.type ≡ W_SHAPE) {
  Wfree(_wclasses[i].shape.indices);
  Wfree(_wclasses[i].shape.vertices);
  _wclasses[i].basic.type = W_NONE;
  continue;
}
```
This code is used in chunk 281.

¶ Por fim, criar e remover instâncias de formas geométricas é trivial, pois tais instâncias são idênticas à instâncias de objetos básicos:

360 ⟨ Wobject: Declaração 277 ⟩ +≡
```
union Wobject *new_shape(union Wclass *wclass);
#define destroy_shape(wobj) _destroy_basic_object(wobj);
```

361 ¶⟨ Wobject: Definição 278 ⟩ +≡

```
union Wobject *new_shape(union Wclass *wclass)
{
  union Wobject *new_obj = _new_basic_object(wclass);
  new_obj→basic.type = W_SHAPE;
  return new_obj;
}
```

¶

8.2 Desenhando Formas no Loop Principal

As formas poderão ser desenhadas quando estivermos em cada iteração do loop principal. Para isso, basta que elas sejam visíveis. Nós precisamos informar o servidor OpenGL do vetor de índices que especifica a ordem de cada vértice no desenho. Fora isso, o procedimento é idêntico ao de objetos básicos:

363 ⟨ Desenho de Objetos no Loop Principal 363 ⟩ ≡

```
case W_SHAPE:
  for (j = 0; j < W_MAX_INSTANCES; j++) {
    if (_wclasses[i].shape.instances[j].basic.type ≡ W_NONE) continue;
    ⟨ Transformação Linear de Objeto (i, j) 304 ⟩
    glVertexAttribPointer(_shader_vPosition, 3, GL_FLOAT, GL_FALSE,
        6 * sizeof(float), (void *) 0);
    glVertexAttribPointer(_shader_VertexNormal, 3, GL_FLOAT, GL_FALSE,
        6 * sizeof(float), (void *)(sizeof(float) * 3));
    glEnableVertexAttribArray(_shader_vPosition);
    glEnableVertexAttribArray(_shader_VertexNormal);
    glBindVertexArray(_wclasses[i].shape._vertex_object);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,
        _wclasses[i].shape._element_object);
    glDrawElements(GL_TRIANGLE_FAN, _wclasses[i].shape.number_of_indices,
        GL_UNSIGNED_SHORT, Λ);
  }
  continue;
```

This code is used in chunk 288.

¶

8.3 Funções Auxiliares

8.3.1 Calcular o Vetor Normal à um Triângulo

Um triângulo é definido por três pontos A, B e C. Definiremos uma função que recebe como argumento quatro vetores de três pontos flutuantes. O primeiro vetor armazenará a resposta. Os próximos três irão conter a coordenada de cada um dos pontos do triângulo:

366 ⟨ Funções Auxiliares: Declaração 336 ⟩ +≡
```
void _normal_vector_to_triangle(float *answer, float *A, float *B, float
    *C);
```

367 ¶⟨ Funções Auxiliares: Definição 337 ⟩ +≡
```
void _normal_vector_to_triangle(float *answer, float *A, float *B, float
        *C) {⟨ Cálculo do Vetor Normal ao Triângulo 368 ⟩
    }
```

¶ Nosso primeiro objetivo é obter dois vetores U e V. Estes vetores são obtidos pegando dois lados do triângulo, colocando uma ponta na origem e obtendo o valor do vetor cuja posição corresponde à outra ponta do lado do triângulo. Os lados do triângulo precisam ser distintos:

368 ⟨ Cálculo do Vetor Normal ao Triângulo 368 ⟩ ≡
```
float U[3], V[3];
{
  int i;
  for (i = 0; i < 3; i++) {
    V[i] = B[i] − A[i];
    U[i] = C[i] − B[i];
  }
}
```
See also chunk 369.

This code is used in chunk 367.

¶ Tendo obtido os vetores U e V, podemos agora calcular o seu produto vetorial usando o "Método de Sarrus":

369 ⟨ Cálculo do Vetor Normal ao Triângulo 368 ⟩ +≡
```
{
  answer[0] = U[1] * V[2] − U[2] * V[1];
  answer[1] = U[2] * V[0] − U[0] * V[2];
  answer[2] = U[0] * V[1] − U[1] * V[0];
}
```

¶ O resultado deste produto vetorial é o vetor normal que queríamos.

8.3.2 Normalizar um Vetor

A função de normalizar um vetor deve receber como argumento um vetor de três posições. Ela irá modificá-lo para deixá-lo normalizado:

371 ⟨ Funções Auxiliares: Declaração 336 ⟩ +≡
```
void _normalize(float *V);
```

¶ Normalizar um vetor é simplesmente obter sua magnitude e dividir por ela cada um de seus elementos:

372 ⟨ Funções Auxiliares: Definição 337 ⟩ +≡
```
void _normalize(float *V)
{
  int i;
  float magnitude = sqrtf(V[0]*V[0]+V[1]*V[1]+V[2]*V[2]);
  for (i=0; i<3; i++) V[0] /= magnitude;
}
```

¶

List of Refinements