U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

LCOM - 2019/2020

Fevereiro de 2020

AUTO DEFENSE

Projeto Final de LCOM

Realizado por:

-Davide Castro – up201806512

-Ricardo Fontão – up201806317

Índice

Resumo

Para o nosso projeto final de LCOM, decidimos desenvolver uma aplicação, mais especificamente, um jogo, em linguagem C, usando os conhecimentos obtidos no decorrer da cadeira, em particular, dispositivos de E/S.

1 – Instruções para o utilizador

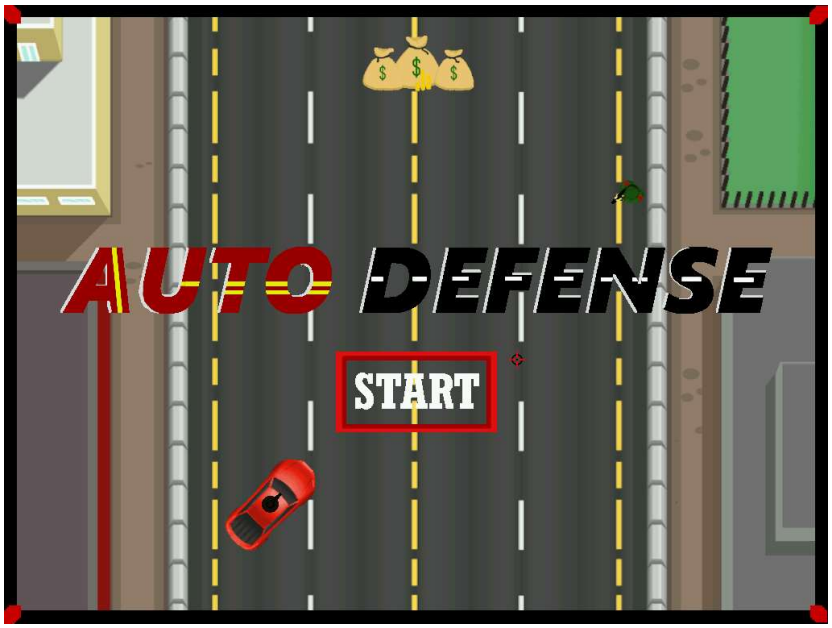
1a) - Iniciar o jogo

Para iniciar o jogo temos duas opções: singleplayer ou multiplayer.

Para iniciar o modo singleplayer basta correr na pasta do projeto “lcom_run proj single”. O modo gráfico inicia e abre o menu inicial (1b)).

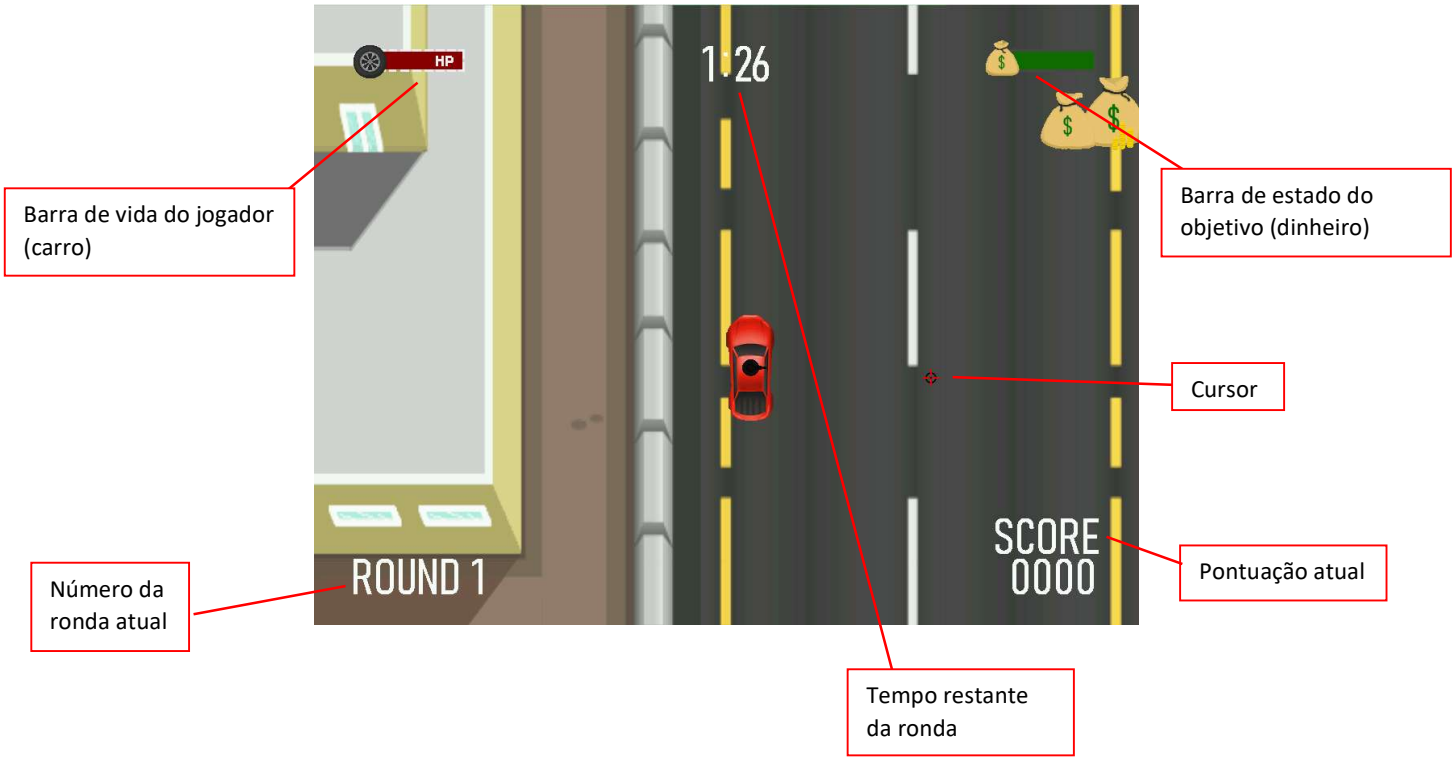
Para iniciar o modo multiplayer temos de ter as duas VMs abertas e numa delas corremos “lcom_run proj player1” e na outra “lcom_run proj player2”. O modo multiplayer funciona igual ao modo singlerplayer mas apenas um jogador consegue usar o rato e o outro o teclado sendo necessária a cooperação de ambos os jogadores.

1b) – Menu inicial



No menu inicial é apresentado o título do jogo e um fundo apresentando uma parte do mapa, bem como um botão de iniciar que pode ser usado ao simplesmente clicar com o botão esquerdo do rato quando o cursor estiver sobre ele. Além disso, a qualquer momento o utilizador pode pressionar a tecla ‘ESC’ para sair do jogo.

1c) – Interface de utilizador



Após iniciar o jogo, o utilizador é deparado com uma 'user interface', indicando, nos cantos superiores, a 'vida' do jogador (o carro vermelho) e do objetivo a proteger (os sacos de dinheiro), através de barras que vão diminuindo à medida que a vida diminui. Também é apresentado o número da ronda atual, a pontuação obtida e o tempo restante na ronda. Adicionalmente, um cursor também é usado para o utilizador saber a posição do mouse.

1d) – Jogabilidade e elementos do jogo

- **Carro e arma:**

O carro e a arma são os únicos elementos controlados pelo jogador. O movimento do carro é efetuado usando as teclas “W”, “A”, “S” e “D” do teclado, para se mover para cima, para a esquerda, para a direita e para baixo, respetivamente.

Para operar a arma do carro, é usado o cursor que pode ser movido com o rato para o alvo pretendido, e premindo o botão esquerdo para disparar.

- **Inimigos:**

Os inimigos aparecerão de vários sítios no mapa do jogo, dirigindo-se durante a ronda ao objetivo a proteger pelo utilizador. Todos os inimigos possuem uma barra de saúde que, ao esgotar-se, faz com que estes desapareçam. No decorrer das rondas, os inimigos serão mais difíceis pois terão mais pontos de saúde.

- **Objetivo:**

O objetivo, representado por sacos de dinheiro, é um dos elementos principais do jogo, sendo este sujeito a um ataque constante por parte dos inimigos e requer proteção. Ao esgotar-se a barra de estado do objetivo, o utilizador perde automaticamente o jogo. Além disso, a posição deste é sempre fixa ao longo do jogo.

2 – Estado do Projeto

2a) – Funcionalidades implementadas

Neste especificação deste projeto comprometemo-nos a implementar todos os dispositivos de E/S que nos eram disponíveis (Timer, KBC, Mouse, Graphics, RTC, Serial Port). Conseguimos implementar todos os dipositivos.

Dispositivos	Uso	Interrupções
Timer	Usado para controlar frame rate	Sim
KBC	Usado para controlar a direção do carro	Sim
Mouse	Usado para apontar e disparar balas	Sim
Video Card	Usado para menu e display do ecrã de jogo	Não
RTC	Usado para contagem de tempo das rondas	Sim
Serial Port	No modo multi, passa scancodes e packets de mouse entre cada jogador	Não

- **Timer**

Neste projeto usamos as funções criadas no Lab 2 desenvolvidas para interface com o dispositivo de E/S Timer. Este foi usado primariamente para manter um frame rate constante de 60 frames por segundo. Foi usado de forma que a cada interrupção gerada pelo timer era desenhado no frame buffer o frame seguinte do jogo. Após isso entrava em ação toda a lógica de jogo envolvida em criar o frame seguinte.

- **KBC**

Neste projeto usamos as funções criadas no lab 3 desenvolvidas para interface com o dispositivo de E/S KBC. Este foi usado para interpretar os cliques do utilizador no teclado de forma a controlar a direção do carro no jogo. Baseado nas teclas carregadas no momento o jogo escolhe o sprite apropriado para a direção escolhida pelo utilizador.

- ## Mouse

Neste projeto usamos as funções criadas no Lab 4 desenvolvidas para interface com o dispositivo de E/S Mouse. Este foi usado para de acordo com os movimentos do rato escolhíamos um sprite apropriado para a direção da arma e caso haja um clique no botão esquerdo do rato o carro dispara uma bala em direção à mira que segue sempre o rato.

- ## Video Card

Neste projeto usamos as funções criadas no lab 5 desenvolvidas para interface com o dispositivo de E/S Video Card. Este foi usado para colocar o Minix em modo gráfico e mostrar os frames criados pelo jogo a cada interrupção do timer. Neste projeto usamos o modo de vídeo 0x117, ou seja, modo direto com modo de cores RGB 5/6/5. Usamos também a técnica de double-buffering, ou seja, criamos um buffer separado do principal que vai sendo criado logo após a demonstração do frame atual e que é copiado a cada interrupção do timer. Isto é usado de forma a reduzir 'visual artifacts' devido a movimento de sprites no caso de usar apenas um buffer de vídeo.

Para detetar colisões de sprites no ecrã, decidimos recorrer a uma matriz de colisões. Neste caso, usamos uma função que implementamos que desenha um objecto na matrix na sua posição respetiva no mapa do jogo. Cada ponto da matriz assim tem um objeto pertencente a um grupo (grupo de inimigos, de balas de inimigos,etc.), cada um com um ID. Sempre que houver colisão a desenhar na matriz, tratámo-la propriamente usando os grupos e ID's para, por exemplo, saber quando uma bala atinge um inimigo e tirar pontos de saúde do mesmo.

Para animar sprites, como a arma e o carro, usamos funções específicas para estes que, usando um array com várias sprites, escolhe a sprite adequada para o objeto de acordo com a sua posição e velocidade.

- ## RTC

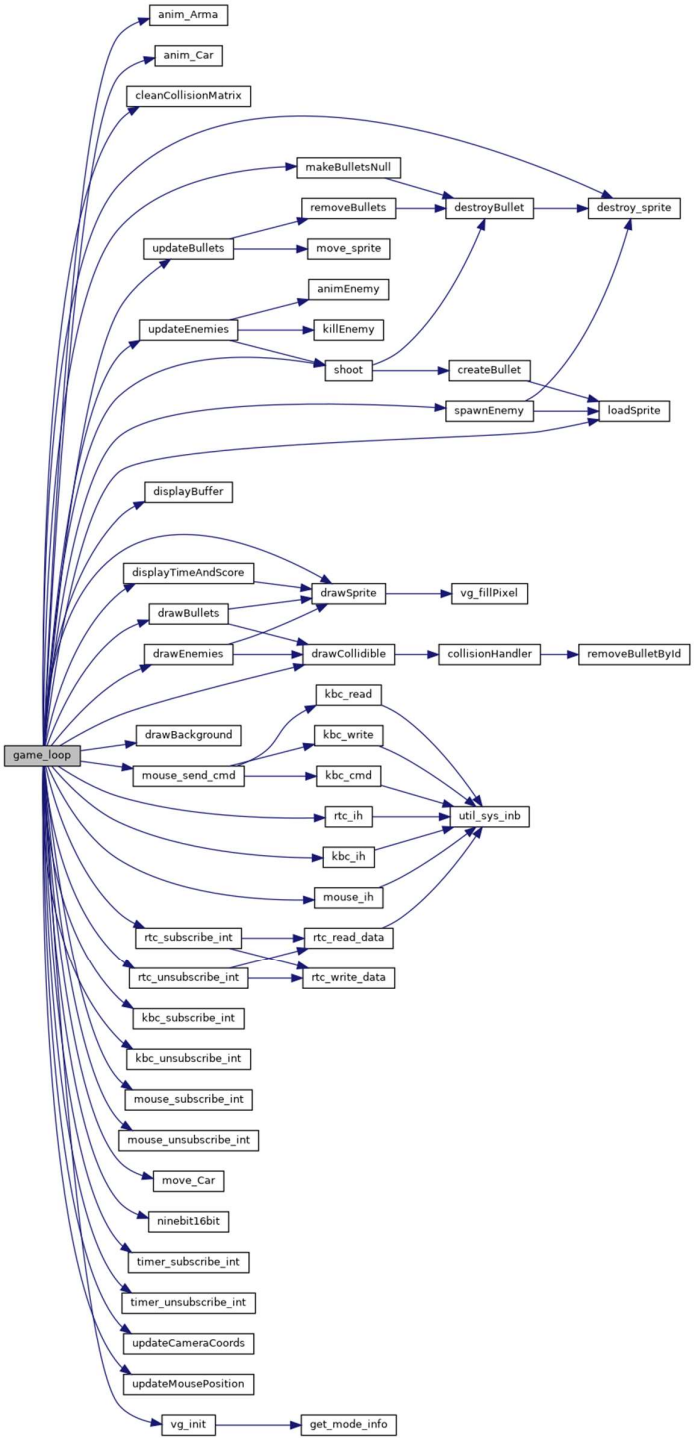
Neste projeto implementámos algumas funções de interface com o dispositivo de E/S RTC, já que não houve nenhum Lab que cobrisse este dispositivo. Este é usado para gerar interrupções periódicas a cada 0.5 segundos de forma a contar o tempo de forma o mais rigorosa possível usado o tempo até que uma certa ronda acabe.

- ## UART (Serial Port)

Neste projeto implementámos algumas funções de interface com o dispositivo de E/S UART, já que não houve nenhum Lab que cobrisse este dispositivo. Este é usado para enviar scancodes e mouse packets entre jogadores de forma a que no modo multi um jogador controle o rato e o outro controle o teclado.

3 – Organização e estrutura do código

Function call graph:



Nota:A call graph anterior refere-se apenas ao game_loop de singleplayer.

Módulos desenvolvidos nas aulas práticas:

Timer – Peso relativo 5%

KBC – Peso relativo 5%

i8042 – Peso relativo 1%

i8254 – Peso relativo 1%

Mouse – Peso relativo 5%

Video Card – Peso relativo 5%

Utils – Peso relativo 1%

Módulos implementados durante o projeto:

Game – Este módulo contém algumas macros importantes relativos ao carro, objetivo e arma. Contém também o loop principal de jogo. Peso relativo 14% (Desenvolvido 50/50).

Enemy – Este módulo contém toda a informação sobre os inimigos e o seu comportamento. Contém o array de inimigos e atualiza cada inimigo consoante o estado do jogo no loop. Peso relativo 10% (Desenvolvido por Davide).

Collisions – Este módulo tem como objetivo tratar tudo o que tenha a ver com colisões de sprites como guardar a matriz de colisões, a função de desenhar um sprite nesta matriz e o handler de colisões. Peso relativo 15 % (Desenvolvido por Ricardo).

Bullet – Este módulo contém toda a informação sobre as balas e o seu comportamento. Foca-se em administrar o array de Bullets que representa as balas presentes em jogo. Tem funções de criar, remover, e modificar caba bala no array. Peso relativo 10% (Desenvolvido por Ricardo).

Sprite – Este módulo trata de tudo que tenha a ver com sprites como criá-los, apagá-los. Trata também da lógica do objeto Carro e do objeto Arma. Peso relativo 20% (Contribuições: Davide: Moves, Anims e Score;Ricardo: Draws e Map scrolling).

RTC – Este módulo permite a interface com o Real Time Clock do Minix. Foram implementadas as interrupções periódicas para fazer a contagem do tempo de cada ronda.Peso relativo 2% (Desenvolvido por Ricardo e incorporado por David).

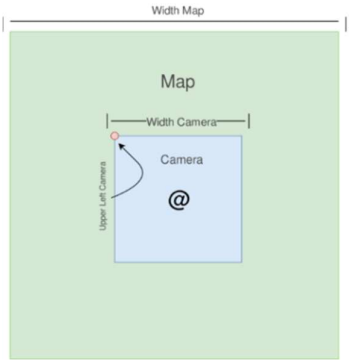
Serial Port – Este módulo permite a interface com a UART do Minix. Foram usados os FIFOs. Uma máquina envia scancodes e a outra envia mouse packets. Peso relativo 5% (Desenvolvida por Davide).

Módulos de terceiros:

Queue – Módulo que implementa uma queue para usar com a serial port. Foi retirado de https://www.geeksforgeeks.org/queue-linked-list-implementation/ em 04/01/2020. Peso relativo 1%(Adaptado por Davide).

4-Detalhes de implementação

Scrolling map – De forma a permitir que tenhamos um mapa de jogo maior que o tamanho da resolução do ecrã decidimos implementar esta técnica que permite a existência de uma mapa maior que o que o jogador consegue atualmente ver e que no nosso caso mantém sempre o carro no centro e vai mexendo baseado na sua posição como no exemplo seguinte.



-A atualização da posição da câmara é feita com base num sprite que é passado por argumento à função de atualização.

```
void updateCameraCoords(Sprite * sprite,int * x,int * y){

        if(sprite->x < hres/2){
                //mouse_x -= 0;
                *x = 0;
        } else if(sprite->x > XRES - hres/2){
                mouse_x += (XRES-hres)-(*x);
                *x = XRES - hres;
        } else {
                mouse_x += (sprite->x - hres/2)-(*x);
                *x = sprite->x - hres/2;
        }

        if(sprite->y < vres/2){
                //mouse_y -= 0;
                *y = 0;
        } else if(sprite->y > YRES - vres/2){
                mouse_y += (YRES-vres)-(*y);
                *y = YRES - vres;
        } else {
                mouse_y += (sprite->y - vres/2)-(*y);
                *y = sprite->y - vres/2;
        }
        ui_x = *x + 50;
        ui_y = *y + 50;
```

Ou seja, nesta função x e y (coordenadas do topo-direito da câmara) são atualizadas baseadas num sprite “sprite” que no nosso carro seria o carro.

Um desafio na implementação desta função é manter o cursor do rato dentro do ecrã e acompanhar a câmara.

Um dos problemas que encontrámos com esta abordagem ao jogo foi o de a especificação de C que usamos limita o tamanho máximo de literais de string, ou seja, como usamos xpm para os sprites e o background, estes têm um tamanho limite. Decidimos então usar uma resolução para o jogo total de 2000x2000 que se encaixa dentro do tamanho possível.

Desenho do buffer com memcpy – De forma a termos uma performance aceitável no desenho de cada frame em vez de copiar do buffer secundário (double buffering) usamos a função memcpy linha a linha de forma a que este se torne um processo eficiente. Este processo precisa de ser linha a linha já que apenas queremos copiar a parte que tem que ser vista pelo jogador.

```
void displayBuffer(void* video_buffer,void* video_mem,int x,int y){

        uint8_t * copyVideo_mem = (uint8_t*)video_mem;
        uint8_t * copyVideo_buffer = (uint8_t*)video_buffer;

        copyVideo_mem += (bitsPerPixel/8) * (XRES * y + x);

        for(size_t i = 0; i < vres; ++i){
                memcpy(copyVideo_buffer,copyVideo_mem,hres*bitsPerPixel/8);
                copyVideo_mem += XRES*bitsPerPixel/8;
                copyVideo_buffer += hres*bitsPerPixel/8;
        }

}
```

Para o desenho do background no buffer secundário usamos uma técnica semelhante mas como este ocupa o tamanho todo do buffer bastou o uso de um único memcpy fazendo um processo muito eficiente.

```
void drawBackground(Sprite * background){

        memcpy(video_mem, background->map, XRES*YRES*bitsPerPixel/8);

}
```

Colisões pixel a pixel – Neste jogo para detetarmos colisões usamos uma matriz de jogo secundária composta de CollisionCells que nos permitem saber, caso dois pixéis coincidam no desenho, o par de sprites que colidiram e o collision handler trata das consequências dessa colisão.

```
typedef struct {
        unsigned short int group;       /**<Group of the collidible */
        unsigned short int id;          /**<ID of the collidible */
} CollisionCell;

#define CARGROUP 1              /** @brief Group of the car object*/
#define ALLYBULLETGROUP 2       /** @brief Group of the bullets shot by the car*/
#define ENEMYBULLETGROUP 3      /** @brief Group of the bullets shot by enemies*/
#define ENEMYGROUP 4            /** @brief Group of the enemies*/
#define OBJECTIVEGROUP 5        /** @brief Group of the objective*/
```

Na imagem podemos então ver a struct CollisionCell e a identificação de cada grupo de sprites.

Cada sprite que pode colidir de alguma forma é desenhado nesta matriz de forma a que quando outro for desenhado por cima o jogo irá detetar a colisão e chamar o collision handler.

```
bool collisionHandler(CollisionCell collider1,CollisionCell collider2, Car* car, Objective* object){

        if(collider2.group == CARGROUP)
        {
                if(collider1.group == ENEMYGROUP)
                {
                        enemies[collider1.id]->hp = 0;
                        return false;
                }
        }
        else if(collider2.group == ALLYBULLETGROUP)
        {
                if(collider1.group == ENEMYGROUP)
                {
                        removeBulletById(bullets, collider2.id);
                        enemies[collider1.id]->hp -= 30 + (round_time/60)*2 ;
                        score += 2;
                        return true;
                }
        }
```

(Excerto do collisionHandler)

"Rotação" de sprites – Neste projeto para dar uma sensação de rotação a cada sprite (Carro, arma e inimigos) decidimos usar vários sprites para cada elemento de forma a que baseado nas condições de input do utilizador e do jogo este seja alterado para dar o efeito desejado. Por exemplo, a arma tem 16 sprites associados e, de acordo com a posição do rato, escolhemos o sprite. Esta escolha é feita com base no declive do vetor gerado pelo centro do carro e a posição do cursor.

```
void anim_Arma(Sprite * arma, Sprite * armaAnim[])
{
        int dx = mouse_x - (arma->x + arma->width/2 - 10);
        int dy = arma->y + arma->height/2 - mouse_y; //y positive variation pointing upwards
        double declive = 0.0;
        if(dx != 0)
                declive = (double)dy/(double)dx;

        if((dx > 0 && declive > tg_72) || (dx < 0 && declive <= -tg_72) || (dx == 0 && dy > 0))
        {
                arma->map = armaAnim[0]->map;
        }
        else if(dx > 0 && declive <= tg_72 && declive > tg_54)
        {
                arma->map = armaAnim[1]->map;
        }
        else if(dx > 0 && declive <= tg_54 && declive > tg_36)
        {
                arma->map = armaAnim[2]->map;
        }
        else if(dx > 0 && declive <= tg_36 && declive > tg_18)
        {
                arma->map = armaAnim[3]->map;
        }
```

(Excerto da função usada para a escolha do sprite da arma)

Da mesma forma a escolha do sprite para o carro é baseado nos inputs do teclado do utilizador. Por outro lado, os inimigos viram-se baseado na posição do carro ou a direção em que precisam de ir ou disparar.

Conclusões

Apesar de termos encontrado vários impasses no decorrer do desenvolvimento do projeto, como, por exemplo, as dificuldades com o funcionamento da serial port, ou a estrutura do código que poderia estar mais organizada, consideramos que este projeto despertou interesse e foi um bom exercício para compreender melhor o funcionamento de dispositivos de entrada e saída e também como estruturar um projeto de programação complexo. Este processo de desenvolvimento foi uma boa base de conhecimentos que pode se revelar importante para o futuro e um incentivo para melhorar.

Sugestões:

- Uso de um sistema diferente de versões (como por exemplo: git).
- Alguns handouts podiam ser mais explicitos e melhor estruturados.

O que gostamos:

- Cadeira interessante que nos permite perceber como funciona a interface low-level com os periféricos de um computador.