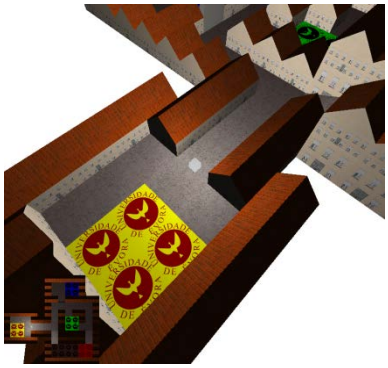
UNIVERSIDADE DE ÉVORA

Departamento de Informática
Licenciatura em Engenharia Informática

Trabalho prático 2 - OpenGL

(Cadeira de Computação Gráfica)

2ºAno



Daniel Gonçalo De Jesus Ramos, 29423
Marcus Vinicius Coelho Santos, 29764

Évora 2013

ÍNDICE

INTRODUÇÃO 0
METODOLOGIA 1
Compilação 1
Compilação em Windows 8 1
Compilação em Mac OSX ..... **Erro! Marcador não definido.**
Outros recursos usados no trabalho 1
DESCRIÇÃO 2
Organização do programa 2
Funcionamento do programa 2
Carregamento dos níveis e estrutura interna 2
Criação de modelos 3D em ficheiro 2
Criação de materiais em ficheiro 4
Criação de texturas em ficheiro 5
Implementação do mapa de colisões e objectivos em ficheiro 5
Desenho do nível em 3D 6
Implementação das Camaras 6
Eventos do jogo 7
Desenho de texto no ecrã 8
Estrutura do programa ..... **Erro! Marcador não definido.**
Classes implementadas ..... **Erro! Marcador não definido.**
CONCLUSÃO 20
BIBLIOGRAFIA 21
ANEXOS 22
Anexo 1: Código fonte 22

ÍNDICE DE FIGURAS

Figura 1: Sequência de cores para completar o nível. 0
Figura 2: Máquina de estados finita para o estado do jogo 0
Figura 3: Máquina de estados finita para a sequência de cores 0
Figura 4: Mapa de modelos 3
Figura 5: Implementação de um modelo em ficheiro 4
Figura 6: Implementação de materiais em ficheiro 4
Figura 7: Implementação da lista de texturas em ficheiro 5
Figura 8: Mapa de colisões e objectivos em ficheiro. 5
Figura 9: Bounding Box 6
Figura 10: Camara com perspectiva isométrica. 6
Figura 11: Camara em primeira pessoa. 7
Figura 12: Camara em terceira pessoa. 7
Figura 13: Camara topo. 7

ÌNDICE DE QUADROS

INTRODUÇÃO

O objectivo deste trabalho é aplicar os conhecimentos adquiridos durante as aulas de computação gráfica, como por exemplo, transformações geométricas, vários tipos de camaras, luzes, texture mapping, bem como desenhar figuras geométricas no ecrã em 3D.

Para aplicar tais conhecimentos, será então necessário criar um simples jogo onde o objectivo do mesmo é permitir ao jogador percorrer uma sequência de cores específica para que seja possível alcançar os objectivos de cada nível. O jogador deverá seguir a sequência de cores descrita na figura 1.

Figura 1: Sequência de cores para completar o nível.

O jogador começa no quadrado amarelo e deverá encontrar o quadrado vermelho, verde e azul respectivamente. Caso o jogador não siga a ordem correcta, nada acontecerá.

O jogador apenas pode andar nas ruas onde existam casas, sendo que o jogador não pode entrar dentro de casas, mas pode abrir certos tipos de portas implementadas no jogo.

Para que o jogo funcione correctamente e seja possível obter a sequência correcta de cores, o jogo conta com duas máquinas de estado finitas (figura 2 e 3).

A primeira máquina de estados finita permite controlar o estado do jogo (figura 2), permitindo saber se o jogador está a jogar num nível, se está no ecrã principal (não está a jogar), se alcançou os objectivos de um nível ou até sair do jogo completamente.

Figura 2: Máquina de estados finita para o estado do jogo

A segunda máquina de estados finita permite controlar os objectivos do jogo que devem ser alcançados pelo jogador. É a partir desta que o jogo sabe qual e a seguinte cor na sequência de cores descrita na figura 1.

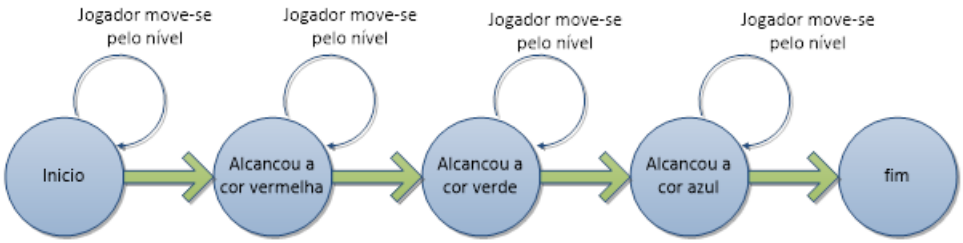


Figura 3: Máquina de estados finita para a sequência de cores

METODOLOGIA

Compilação

O programa foi desenvolvido a partir do IDE Microsoft Visual Studio 2012 para Windows 8. De seguida seguem os passos necessários para compilar o código fonte fornecido com este relatório.

Compilação em Windows 8

1. Fazer o download do glut source code em http://www.opengl.org/resources/libraries/glut/glut_downloads.php
2. Extrair o conteúdo do arquivo e copiar o conteúdo da pasta include para C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC\include
3. Fazer o download do glut precompiled binaries 3.7 em http://www.opengl.org/resources/libraries/glut/glutdlls37beta.zip
4. Copiar os ficheiros com a extensão .lib para C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC\lib
5. Copiar os ficheiros com a extensão .dll para a pasta C:\Windows\System

Outros recursos usados no trabalho

Foram usadas várias imagens retiradas do site cgtextures.com para que fosse possível mapear texturas decentes para os modelos 3D implementados. As texturas foram posteriormente editadas no programa Adobe Photoshop.

DESCRIÇÃO

Organização do programa

Com o objectivo de facilitar não só a organização do código mas também facilitar a criação de novos níveis, o código foi dividido em várias classes para que seja possível reutilizar partes do código sem necessidade de duplicação do mesmo. Uma breve descrição do que cada método faz será descrita mais a frente.

Funcionamento do programa

Carregamento dos níveis e estrutura interna

O carregamento dos níveis é feito a partir de um ficheiro para cada nível disponível, o jogador só ganha quando não houver mais níveis para carregar.

A classe Level contém todos os dados do nível que ficam armazenados em arrays bidimensionais e Maps, sendo que o conteúdo dos mesmos varia entre mapas de colisão, mapas de modelos, lista de texturas e lista de materiais. A classe Level contém ainda os métodos necessários para carregar e desenhar um nível no ecrã.

Os modelos 3D e a maioria do conteúdo do nível são implementados directamente no ficheiro e não no próprio programa, facilitando a sua manipulação.

Um nível pode ser "facilmente" construído, bastando implementar todo um conjunto de propriedades/keywords bem delimitadas no ficheiro do nível que serão descriminadas na secção apropriada.

Criação de modelos 3D em ficheiro

Permite criar um modelo 3D com certas propriedades. O modelo criado pode ser usado, bastando usar o seu ID no mapa de modelos (figura 4). Cada letra está associada a um modelo com excepção da letra N que é ignorada.

A utilidade de ignorar uma letra no mapa de modelos está relacionada com o tamanho dos modelos ou simplesmente de não desenhar nenhum modelo naquele sítio. Por exemplo é possível desenhar um modelo que ocupe mais que um quadrado, bastando para isso incluir a letra N nos quadrados que irão ficar ocupados por esse modelo.

```
MODEL_MAP
NNNNNNHHHHHHHH
NNNNNNHFFFFTTH
NNNNNNHFFFFTTH
DDDDDNHIHHHHIH
DSSCDDDIHHHHID
DSSCCCCCDVVCCD
DCCCDDDCDVVDCD
DDDDDNDCDCDDCD
NNNNNNDCCCCCCD
NNNNNNDDPDDDDD
NNNNNNDCCUUDNN
NNNNNNDCCUUDNN
NNNNNNDDDDDDNN
ENDMM
```

Figura 4: Mapa de modelos

Foram implementadas as seguintes keywords/propriedades para que se possa desenhar um modelo 3D no jogo. Um exemplo de implementação pode ser visto na figura 5.

Keyword: Model

Propriedade	Descrição
ID:	Identificação única do modelo.
BASE_MODEL_ID:	Identificação única do modelo base.
NAME:	Nome do modelo
DATA	Keyword associado aos dados do modelo, nomeadamente os QUADS.

Keyword: ANIMATION

Usado na keyword Model para definir uma animação. A animação pode ser feita nos três eixos XYZ e contem as seguintes propriedades.

Propriedade	Descrição
EVENT:	Nome do evento que acciona a animação.
INCREMENT_X:	Valor que é incrementado a cada ciclo no eixo dos X.
INCREMENT_Y:	Valor que é incrementado a cada ciclo no eixo dos Y.
INCREMENT_Z:	Valor que é incrementado a cada ciclo no eixo dos Z.
DISTANCE_TO_MOVE_X:	Distancia máxima que o model se mode no eixo dos X.
DISTANCE_TO_MOVE_Y:	Distancia máxima que o model se mode no eixo dos Y.
DISTANCE_TO_MOVE_Z:	Distancia máxima que o model se mode no eixo dos Z.

Keyword: DATA

Usado na keyword Model para dizer que existe dados nos modelos a serem lidos.

Keyword: QUAD

Usado na keyword DATA para descrever um GL QUAD

Propriedade	Descrição
MATERIAL:	Nome do material a ser usado.
TEXTURE:	Nome da textura a ser usado no QUAD
UV[1-4]:	Coordenadas UV para texture mapping.
N:	Normal.
V[1-4]:	Vértices do QUAD.

```
Model
        ID: C
        BASE_MODEL_ID: -1
        NAME: CHAO1
        DATA
                QUAD
                        MATERIAL: material_chao1
                        TEXTURE: chao1.bmp
                        C: 1.0 1.0 1.0
                        UV1: 1 0
                        UV2: 1 1
                        UV3: 0 1
                        UV4: 0 0
                        N: 0 0 1
                        V1: 1 0 0
                        V2: 1 1 0
                        V3: 0 1 0
                        V4: 0 0 0
                ENDQ
        ENDD
ENDM
```

Figura 5: Implementação de um modelo em ficheiro

Criação de materiais em ficheiro

Permite criar um material que pode ser usado num QUAD.

Foram implementadas as seguintes keywords/propriedades para que se possa criar um material no jogo. Um exemplo de implementação pode ser visto na figura 6.

```
MATERIALS

        MATERIAL
                NAME: material_chao1
                SHININESS: 100
                SPECULAR: 0.0 0.0 0.0 1.0
                DIFFUSE: 1.0 1.0 1.0 1.0
                AMBIENT: 0.3 0.2 0.2 1.0
                POSITION: 0 0 1 1
                DIRECTION: 0 0 1
        ENDMATERIAL
ENDMATERIALS
```

Figura 6: Implementação de materiais em ficheiro

Keyword: MATERIALS

Define que irá ser criado uma lista de materiais.

Keyword: MATERIAL

Usado na keyword MATERIALS para descrever um material.

Propriedade	Descrição
NAME:	Nome do material.
SHININESS:	Brilho do material.
SPECULAR:	Cor especular do material.
DIFFUSE:	Cor difusa do material.
AMBIENT:	Cor ambiente do material.
POSITION:	Posição da luz.

Criação de texturas em ficheiro

Permite carregar uma lista de texturas e atribui-las a QUADS.

Foram implementadas as seguintes keywords/propriedades para que se possa criar uma lista de texturas a serem carregadas. Um exemplo de implementação pode ser visto na figura 7.

```
TEXTURES
        TEXTURE: chao1.bmp
        TEXTURE: uevora.bmp
        TEXTURE: casa1_parede1.bmp
        TEXTURE: casa1_parede2.bmp
        TEXTURE: casa1_parede3.bmp
        TEXTURE: casa1_telhado.bmp
ENDTEXTURES
```

Figura 7: Implementação da lista de texturas em ficheiro

Implementação do mapa de colisões e objectivos em ficheiro

O mapa de colisões (figura 8) permite saber quais as zonas do jogo em que o jogador não pode passar bem como saber onde estão os objectivos que o jogador deve alcançar.

```
15 11
BBBBBBBBBBBBBBB
BBBBBVBPPPPPPPB
BBBBBPBPBBBBBPB
BBBBBPBPBAPPBPB
BBBBBPBPBBBPBPB
BBBBBPBPPPPPBPB
BBBBBPBBBBBBBPB
BSPPPPPPPPPPPPB
BBBBBPBBBBBBBBB
BBBBBPPPPPPPPRB
BBBBBBBBBBBBBBB
```

B - Caminho bloqueado
P - Caminho onde se pode andar
V - Quadrado verde
A - Quadrado azul
R - Quadrado vermelho
S - Ponto de partida do jogador

Figura 8: Mapa de colisões e objectivos em ficheiro.

A detecção de colisões é feita ao converter as coordenadas do mundo opengl em coordenadas inteiras que podem ser associadas a uma posição de uma array bidimensional. Sempre que uma posição contenha um B, o jogador não poderá avançar por esse caminho.

Para que a colisão seja mais precisa, foi criada uma bounding box com 4 pontos (figura 9) usando um vector 2D que fica associada ao jogador. Quando o jogador se move pelo mapa. É testado esses 4 pontos tendo em conta a posição para onde o jogador se quer mover. Caso algum desses pontos falhe no teste de colisão, isto é, colidiu com um quadrado que contem um B na array do mapa de colisões, o jogador fica impedido de prosseguir.

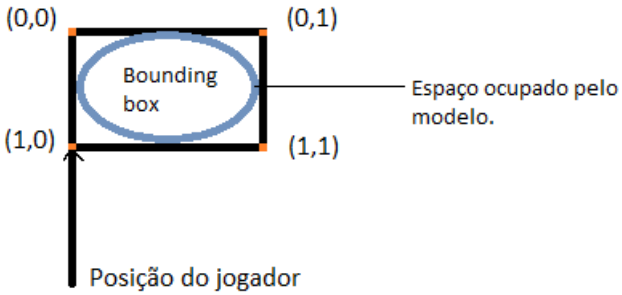


Figura 9: Bounding Box

Desenho do nível em 3D

O desenho do nível é feito ao percorrer uma array bidimensional de **objectos** que contem modelos 3D. Por cada **objecto** é feito o draw dos modelos correspondentes se estes existirem.

Implementação das Camaras

Existe quatro cameras disponíveis no jogo que podem ser alternadas usando a tecla C. Uma camera ortonormada com vista de topo (Figura n), uma camara com perspectiva isométrica, uma camara em primeira pessoa e uma camara em terceira pessoa.

Figura 10: Camara com perspectiva isométrica.

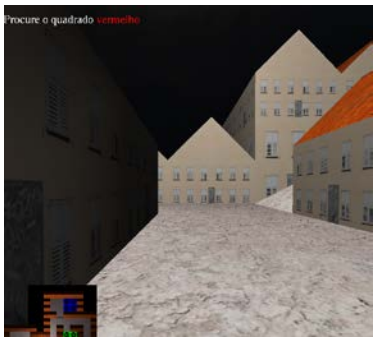


Figura 11: Camara em primeira pessoa.

Figura 12: Camara em terceira pessoa.

Figura 13: Camara topo.

Eventos do jogo

Para que o jogo possa funcionar e o jogador possa completar os objectivos na ordem adequada, existe uma função chamada Event que pertence à classe Game que gere todos os eventos principais do jogo. Quando um evento ocorre, a função recebe o evento e executa o código relacionado com esse evento, como por exemplo, actualizar os objectivos alcançados pelo jogador.

É também possível accionar eventos manualmente ao premir a tecla P. Por exemplo quando se quer abrir uma porta. Quando se prime a tecla P, é enviado um evento para os nodes vizinhos da array bidimensional, accionando assim o evento daquele node se existir.

No caso das portas, o jogador só poderá passar quando a animação acabar, o que indica que o evento terminou.

Desenho de texto no ecrã

O desenho de texto no ecrã é feito com a função drawText em que para evitar que a posição do texto seja independente da camera usada, é feito uma mudança no modo da matrix para GL_PROJECTION e então é desenhado o texto sem afectar as outras partes do mundo, usando glLoadIdentity() para usar a matrix identidade ao fazer os cálculos, retomando de seguida o modo da matrix para GL_MODELVIEW

Estrutura do programa

Definições de variáveis globais.

Constantes

```
#define CAMERA_MINIMAP 0
#define CAMERA_ISOMETRIC 1
#define CAMERA_FIRST_PERSON 2
#define CAMERA_THIRD_PERSON 3
```

Constantes que definem o tipo de camara a ser usado.

```
#define DIRECTION_X_POSITIVE 0
#define DIRECTION_Y_POSITIVE 1
#define DIRECTION_X_NEGATIVE 2
#define DIRECTION_Y_NEGATIVE 3
#define DIRECTION_NONE 5
```

Constantes que definem a direcção da camara.

```
#define PI 3.14159265358979323846264338327950f
```

Constantes que contem o valor do PI.

```
#define CHAR_TO_INT(C) C - 0x21
```

Macro que converte um char num inteiro.

```
#define GAME_STATE_NOT_PLAYING 2
#define GAME_STATE_PLAYING 3
#define GAME_STATE_PLAYER_WON 4
```

Constantes que definem o estado do jogo actual.

```
#define MAP_TILE_SIZE 1
```

Constante que define o tamanho de cada quadrado onde pode estar um ou vários modelos.

```
enum EVENT_TRIGGER
{
        EVENT_TRIGGER_NO_ACTION = 0x00000000,
```

```
        EVENT_TRIGGER_OPEN      = 0x00000001,
};
```

Enumerator que contem os tipos de eventos que podem ser accionados ao clicar em P.

```
enum GameEvents
{
        EVENT_NONE = 0,
        EVENT_PLAYER_PLAYING = 1,
        EVENT_PLAYER_NOT_PLAYING,
        EVENT_PLAYER_REACHED_RED,
        EVENT_PLAYER_REACHED_GREEN,
        EVENT_PLAYER_REACHED_BLUE,
        EVENT_PLAYER_FINISHED_LEVEL,
        EVENT_PLAYER_WON
};
```

Enumerator que define os eventos que são permitidos em jogo para o jogador concluir os objectivos do jogo.

```
#define ERROR_NONE                0
#define ERROR_FAILED_TO_LOAD_LEVEL 1
```

Constantes que definem os erros retornados ao carregar um nivel.

```

enum LevelPath
{
        LEVEL_PATH_PASSABLE = 0,
        LEVEL_PATH_BLOCKED,
        LEVEL_PATH_OBJECTIVE_RED,
        LEVEL_PATH_OBJECTIVE_BLUE,
        LEVEL_PATH_OBJECTIVE_GREEN,
        LEVEL_PATH_PLAYER_START
};
```

Enumerator que contem as constantes relacionadas com o tipo de caminho que esta associado a um quadrado no mapa de colisões.

Estruturas

Estrutura COLOR

Estrutura que representa uma cor específica

Estrutura Vector2

Estrutura que representa um vector 2D.

Estrutura BoundingBox

Estrutura que representa um quadrado com 4 pontos 2D para colisões.

Estructura VECTOR3

Estrutura que representa um vector 3D.

Estrutura Quad

Estrutura que representa um OPENGL QUAD.

Estrutura ModelAnimation

Estrutura que representa uma animação de um modelo.

Estrutura Material.

Estrutura que representa um material.

Classes implementadas
Estrutura que representa um modelo 3D.

Camera
Classe abstracta para implementação de uma camara.

Modifier and Type	Method and Description
	Camera()
virtual	~Camera()
virtual void	PrepareCamera()
virtual void	updateLookAt(float xCenter, float yCenter, float zCenter); Actualiza o vector para onde a camera olha.
Virtual void	updateLookFrom(float xEye, float yEye, float zEye) Actualiza o vector de onde a camara olha.
void	setEye(vector3 v) Atribui o vector para o olho da camara.
vector3	getEye() Obtem o vector para o olho da camara.
void	setCenter(vector3 v) Atribui o centro da camara.
Vector3	getCenter() Obtém o centro da camara.
void	setAngle(int angle)

	Atribui o ângulo da camara.
int	getAngle()
void	move(int cameraID, int direction) Move a camara na direcção indicada.
void	rotate(int degree, int direction) Roda a camara na direcção indicada

IsometricCamera
Classe que prepara uma camara com projecção isométrica.

Modifier and Type	Method and Description
	IsometricCamera()
	~IsometricCamera()
void	PrepareCamera() Prepara a camera em modo isométrico

FirstPersonCamera
Classe que prepara uma camara em primeira pessoa.

Modifier and Type	Method and Description
	FirstPersonCamera() ~ FirstPersonCamera()
void	PrepareCamera() Prepara a camera em modo primeira pessoa

ThirdPersonCamera
Classe que prepara uma camara em terceira pessoa.

Modifier and Type	Method and Description
	ThirdPersonCamera() ~ ThirdPersonCamera()
void	PrepareCamera() Prepara a camera em modo terceira pessoa

TopCamera
Classe que prepara uma camera com projecção ortonormada (paralela).

Modifier and Type	Method and Description
	TopCamera()
	~TopCamera()
void	PrepareCamera() Prepara a camera em modo topo para mini mapas e vista de topo.

Light
Classe que define uma luz no jogo.

Modifier and Type	Method and Description
	Light()
	~Ligth()
void	prepareLight() Prepara as propriedades da luz.
void	setAmbLight(float a, float b, float c) Define a cor da luz ambiente.
void	setDiffLight(float a, float b, float c) Define a luz difusa.
	setSpecLight(float a, float b, float c) Define a luz specular.
	setLightPosition(float x, float y, float z) Define a posição da luz.

Object
Classe que define um objecto que contem modelos.

Modifier and Type	Method and Description
	Object()
	~Object()

Model*	getModel() Retorna o modelo associado a este objecto.
void	setModel(Model *m) Atribui o modelo ao objecto.
vector3&	getPosition() Retorna a posicao do objecto.
void	EventTrigger(EVENT_TRIGGER event) Evora um evento.
void	setOriginalPosition(vector3 v) Atribui a posição original do objecto.
Vector3	getOriginalPosition() Retorna a posição original do objecto.
	DrawModel(Model *m) Faz o draw dos modelos associados a este objecto.
void	Update(int time) Actualiza o objecto.

Level
Contem métodos que permitem carregar um nível e desenha-lo no ecrã.

Modifier and Type	**Method and Description**
	Level() ~Level()
void	Update () Desenha o nível.
Int	LoadLevel() Carrega os dados de um nível caso seja possível.
void	ReadColisionContent(char *buffer, int row) Lê o conteúdo do mapa de colisoes
int	getPlayerStartX() Obtém a posição x do jogador na array do tabuleiro. Apenas útil para operações de conversão para coordenadas

	do mundo opengl ou para cameras.
int	getPlayerStartY() Obtém a posição y do jogador na array do tabuleiro. Apenas útil para operações de conversão para coordenadas do mundo opengl ou para cameras.
int	getTileSizeX() Obtém a largura do tabuleiro que será usado para criar arrays bidimensionais.
int	getTileSizeY() Obtém a altura do tabuleiro que será usado para criar arrays bidimensionais.
bool	Collides(float x, float y) Verifica se as coordenadas dadas colidem com um tile bloqueado. Usa pixel color collision pelo backbuffer. Lança também os eventos que sejam detectados.
Int **	getColisionMap() Retorna a array que contem os dados do nível
void	ReadModelMalContent(char *buffer, int row) Lê o conteúdo do mapa de modelos
Object ***	getModelMap() Retorna a array que contem o mapa de modelos.
void	FreeResources(); Liberta os recursos associados ao nível.
std::map<string, int>&	getTextureList()
Std::map<string, Material*>&	getMaterials() Retorna a lista de materiais.
Model *	getModel(int id) retorna um modelo baseado no seu id.
	signalEventToNearByNodes(float x, float y, EVENT_TRIGGER event) Lança um evento aos nós mais próximos do jogador.

Player
Contem métodos que permite manipular o jogador.

Modifier and Type	Method and Description
	Player()
	~Player()
int	getObjectiveState() Retorna o estado dos objectivos alcançados pelo jogador.
void	setObjectiveState(int state) Actualiza os objectivos alcançados pelo jogador.
void	setPosition(float x, float y, float z) Atribui a posição do jogador.
void	setPosition(vector3 pos) Atribui a posição do jogador.
void	setColor(float r, float g, float b) Atribui a cor do jogador.
vector3&	getPosition() Retorna a posição actual do jogador.
color&	getColor() Retorna a cor do jogador.
int	getFront() Retorna a direcção para onde o jogador está a olhar.
void	setFront(int front) Atribui a direcção para onde o jogador está a olhar.
void	Update() Actualiza os dados do jogador. Permite desenhar o jogador no mundo.
void	UpdateZPosition() Detecta rampas e actualiza a posição Z automaticamente.
bool	Collides(float x, float y) Testa se existe colisões usando a Bounding Box.

Common

Classe que contem métodos partilhados pelo programa inteiro e permite aceder a uma instância única da classe Game.

Modifier and Type	Method and Description
	Common()
	~Common()
static float	getClippingFromTile(int tile, bool upperRound) Obtém as coordenadas "ideais" para o clipping que será usado no glortho baseado no tamanho do nível. Permite arredondar para o numero inteiro acima.
static int	roundToTile(float position) Converte uma coordenada do mundo opengl para uma coordenada de uma array. Arredondado para compensar margens de erro.
static float	tileToPos(int tile) Converte uma posição na array em coordenadas do mundo opengl. E necessário fazer ajustamentos porque as coordenadas convertidas tem como origem a posição (0,0).
static game *	getGame() Obtém a instância do jogo actual.
static bool	`checkMapBounds(int tileX, int tileY)` Método que verifica se uma coordenada no mapa de modelos ou colisões está out of bounds.

Game
Classe que contem os métodos necessários para que o jogo se torne jogável, seja possível mostrar o tabuleiro no ecrã, configurar o OpenGL, Glut e cameras entre outras tarefas.

Modifier and Type	Method and Description
	Game(int width, int height)
	~Game()
void	SetupGL(int argc, char **argv) Configuracao inicial do opengl.
void	SetupCameras() Configura as cameras.
int	getShading() retorna o shading activo.
void	setShading(int shading) Atribui o shading activo.
void	relativeMovementForward() Permite mover o jogador dependendo da direcção.
static Void	Draw() Faz o draw do nivel. Desenha a parte visivel e iteractiva no front buffer. Desenha o nivel no backbuffer para deteccao de colisoes.
static void	reshape (int width, int height) Actualiza o viewport com a nova largura e altura do ecra.
static void	keyboardEvent(unsigned char c, int x, int y) Gere os eventos do teclado.
void	DrawUI() Desenha a parte do interface grafico, como por exemplo * os objectivos do nivel.
static void	Update(int value) Funcao do glut que permite actualizar continuamente o ecra.
void	Event(int event)

	Gere os eventos lancados no jogo.
int	getState() Obtem o estado do jogo actual.
void	drawAxis() Desenha os eixos.
int	getWidth() Obtem a largura do ecra.
int	getHeight() Obtem a altura do ecra.
void	setWidth(int w) Atribui a largura do ecra.
void	setHeight(int h) Atribui a altura do ecra.
Level *	getLevel() Obtem a instancia do nivel actual.
Void	setLevel(Level *level) Atribui a instancia actual do nivel para acesso posterior.
Player *	getPlayer() Obtem a instancia actual do jogador.
void	setPlayer(Player * player) Atribui a instancia actual do jogador para acesso posterior.
Camera *	getCamera() Obtem a camera actual activa.
Camera *	getCamera(int i) Obtem a camera actual activa dado o seu índice.
void	setCamera(int i) Atribui o id de uma das cameras disponiveis.
void	drawText(char *text, float x, float y, float r, float g, float b, void * font) Permite desenhar texto no ecra.
void	UpdateViewport(GLint x, GLint y, GLsizei width, GLsizei height)

	Actualiza o viewport.
int	getCurrentCameraId() Obtem o id da camera actual.
void	setCurrentLevelId(int id) Atribui o id do nivel actual.
int	getCurrentLevelId() Obtem o id do nivel actual.

CONCLUSÃO

O trabalho teve algum grau de complexidade, mas os conhecimentos adquiridos, penso que valeram o esforço. Tivemos algumas complicações em relação ao MAC em que as luzes por algum motivo especifico, nunca funcionam bem em ambos os sistemas operativos(Windows 8 e mac). Os níveis contem duas luzes activadas, uma spot e uma omni mas so se nota a spotlight.

Alguns bugs poderão ser visíveis nas camaras em termos de clipping ou até nas rampas implementadas, mas pensamos que a rampa implementada está boa em termos de calculo no eixo dos Z.

De notar que foi fornecida alguma ajuda a alguns colegas que se encontravam em dificuldades como também tivemos alguma ajuda de colegas.

BIBLIOGRAFIA

1. http://www.opengl.org/sdk/docs/man2/xhtml/glOrtho.xml
2. http://programming-technique.blogspot.pt/2012/02/lightning-in-opengl-with-spot-light.html

ANEXOS

Anexo 1: Código fonte