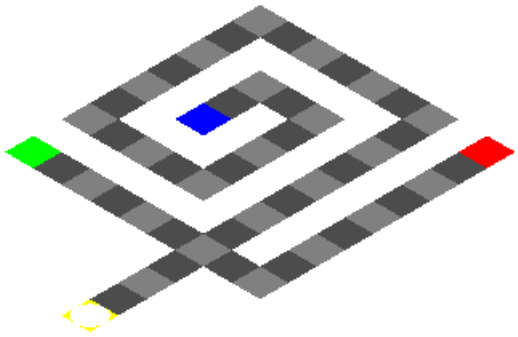
Departamento de Informática
Licenciatura em Engenharia Informática

Trabalho prático 1 - OpenGL

(Cadeira de Computação Gráfica)

2ºAno



Daniel Gonçalo de Jesus Ramos, 29423

Évora, 2013

Índice

Índice de figuras

Introdução

O objectivo deste trabalho é aplicar os conhecimentos adquiridos durante as aulas de computação gráfica, como por exemplo, transformações geométricas, projecções isométricas e ortonormadas, detecção da cor de um pixel, bem como desenhar figuras geométricas no ecrã.

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.

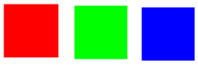


Figura 1: Sequência de cores para completar o nivel

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 em certos quadrados que contenham tons de cinzento, vermelho, verde, azul ou amarelo. A área do nível que está a preto não permite movimentos por parte do jogador.

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.

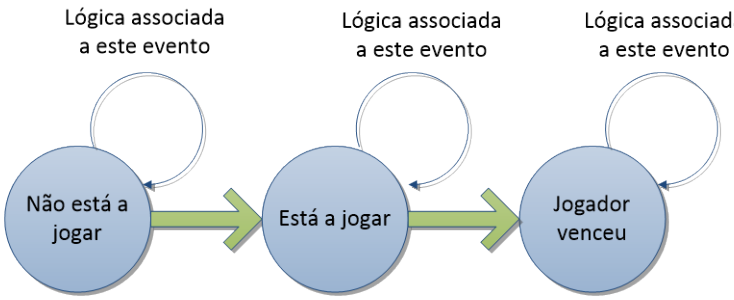


Figura 2: Maquina de estados finita para o estado do jogo

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.

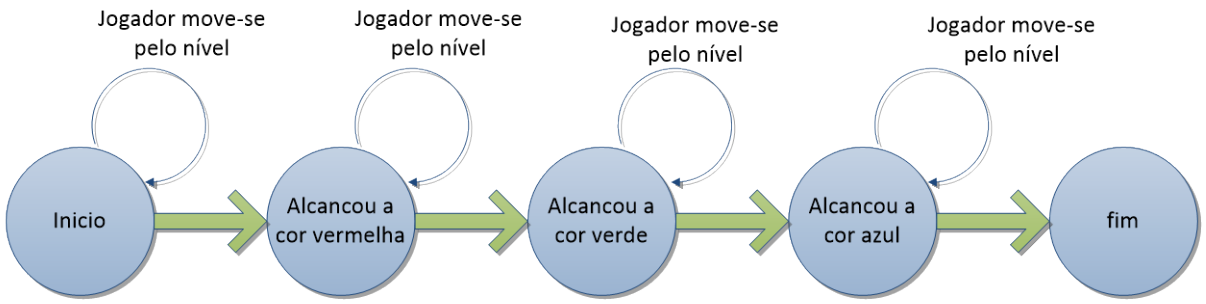


Figura 3: Maquina de estados finita para a sequência de cores

As funções e a lógica por detrás do jogo serão discutidas noutra parte deste relatório.

Desenvolvimento

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.

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

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.

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 contem todos os dados do nível que ficam armazenados numa array bidimensional bem como os métodos necessários para carregar e desenhar um nível no ecrã. Um nível pode ser facilmente construído usando apenas letras específicas para cada tipo de quadrado como mostra a figura 4.

```
15 11 0
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 4: estructura do nivel

Os dados iniciais do nível permite saber o tamanho do nível em largura e altura bem como o modo de jogo. De notar que o nível armazenado no ficheiro deverá ficar rodeado de “quadrados” bloqueados (tipo B) para facilitar os cálculos e evitar que o jogador saia fora da área visível do

ecrã. Caso seja necessário pode-se acrescentar quadrados bloqueados extra para que o tabuleiro fique totalmente visível no ecrã ou para que o tabuleiro não dificulte a visualização do texto que contem os objectivos. Este pequeno erro deve-se por vezes a falha no cálculo do clipping que é usado no glOrtho.

Detecção de colisões.

A detecção de colisões é feita através da leitura da cor do pixel que está no ecrã.
É usado o backbuffer para desenhar o conteúdo do nível e posteriormente detectar a cor do pixel. A detecção da cor de um pixel funciona ao fazer-se uma projecção a partir das coordenadas do mundo para as coordenadas da janela.
É também durante a detecção de colisões que é lançado os eventos quando o jogador toca num dos quadrados que fazem parte do objectivos do nível.

Desenho do nível

O desenho do nível é feito ao percorrer uma array bidimensional e por cada posição da array, detecta-se o tipo de quadrado que se deve desenhar. Para os quadrados em que o jogador pode caminhar, alterna-se a cor usando um simples modulus.
A posição de cada quadrado é ajustada baseado no valor calculado do clipping que é usado no glOrtho.
Os quadrados são desenhados segundo a seguinte função:

glRectf(j * squareSize - screenAdjustmentX, i * -squareSize + screenAdjustmentY, j * squareSize + squareSize - screenAdjustmentX, i * -squareSize - squareSize + screenAdjustmentY);

Sendo que o j e i são as posições da array, o squareSize, o tamanho do quadrado (0.2f por defeito) e o screenAdjustment é o valor calculado do clipping.

Cameras

Existe duas cameras disponíveis no jogo que podem ser alternadas usando a tecla R. Uma camera ortonormada (figura 5) e uma camera isométrica (figura 6)
A função glOrtho é automaticamente ajustada em relação ao tamanho do nível para que o nível seja totalmente visível no ecrã, sendo que na vista isométrica os valores são arredondados para o valor mais acima usando a função ceil.

Formula: Clipping = *Tamanho do nível * 0.1f*
glOrtho(-Clipping, Clipping, - Clipping, Clipping, - Clipping, Clipping);
Tem-se sempre em consideração o maior valor em relação à largura ou à altura do nível.

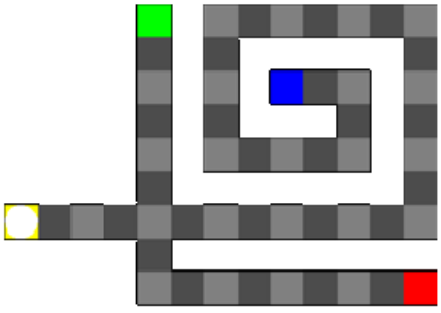


Figura 5: Projecção ortonormada

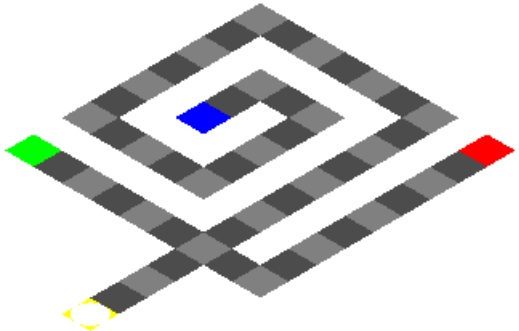


Figura 6: Projeccção isométrica

A vista isométrica conta ainda com um minimap e dois viewports.

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.

Modos de jogo

O jogo conta com dois modos de jogo, o modo normal, que apenas é necessário mover o jogador para a sequência correcta de cores. E o modo "Brain" onde o jogador não pode voltar a passar por quadrados já percorridos previamente, sendo que é necessário escolher o caminho com cuidado.

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

Classes implementadas

Camera

Classe abstracta para implementação de uma camera.

Modifier and Type	Method and Description
	Camera()
virtual	~Camera()
virtual void	PrepareCamera()

IsometricCamera

Classe que prepara uma camera com projecção isométrica.

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

NormalCamera

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

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

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	ReadContent(char *buffer, int row) Lê o conteúdo de cada linha do ficheiro.
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 **	getLevelData() Retorna a array que contem os dados do nível
void	setLevelMode(int m) Atribui o modo de jogo a este nível
int	getLevelMode()

	Retorna o modo de jogo para este nível.
void	UpdateLevelModeAI(float px, float py) Permite mudar o modo de jogo.

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.
void	Update() Actualiza os dados do jogador. Permite desenhar o jogador no mundo.

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.

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.
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.

Código fonte

Camera.h

```
#include "Common.h"

#ifndef CAMERA_H
#define CAMERA_H


/**
 * Classe abstracta para implementacao de uma camera.
 *
 */
class Camera
{
public:
	Camera(void);
	virtual ~Camera(void);
	virtual void prepareCamera() = 0;
};

#endif
```

Camera.cpp

```
#include "Camera.h"

Camera::Camera(void)
{
}

Camera::~Camera(void)
{
}
```

Common.h

```
#ifndef COMMON_H
#define COMMON_H

typedef struct
{
	float x;
	float y;
	float z;

}vector3;

typedef struct
{
	float r, g, b;
}color;
```

```
#define PI 3.14159265358979323846264338327950f

//constantes temporarias
#define GAME_STATE_NOT_PLAYING 2
#define GAME_STATE_PLAYING 3
#define GAME_STATE_PLAYER_WON 4

#define MAP_TILE_SIZE 0.2

//eventos

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
};

#define UI_EVENT_NONE 0
#define UI_EVENT_START_PLAYING 1
#define UI_EVENT_QUIT 2

//erros
#define ERROR_NONE		0
#define ERROR_FAILED_TO_LOAD_LEVEL 1

#define LEVEL_MODE_A_MONEY_COULD_DO_IT 0
#define LEVEL_MODE_USE_BRAIN 1

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
};

#include <stdio.h>
#include <stdlib.h>
#include <istream>
#include <fstream>
#include <iostream>
#include <string>
#include <list>
#include <assert.h>
```

```
using namespace std;

#ifdef WIN32
#include "windows.h"
#include <GL/GL.h>
#include <GL/glut.h>

#endif

#include "Player.h"
#include "Camera.h"
#include "IsometricCamera.h"
#include "NormalCamera.h"
#include "Level.h"
#include "Game.h"

/**
 * Acesso global a metodos e variaveis partilhadas pelo
 * projecto.
 */
class Common
{
public:
	Common();
	~Common();
	static float getClippingFromTile(int tile, bool upperRound);
	static int roundToTile(float position);
	static float tileToPos(int tile);
	static Game * getGame();
private:
	static Game * game;
};

#endif
```

Common.cpp

```
#include "Common.h"

Game * Common::game;

Common::Common()
{
	Common::game = new Game(500,500);

}

Common::~Common()
{
	delete Common::game;

}
```

```
/**
 * Converte uma coordenada do mundo opengl para uma
 * coordenada de uma array.
 * Arredondado para compensar margens de erro.
 */
int Common::roundToTile(float position)
{
	int tile = floor(position*5 + 0.05);
	return tile;
}

/**
 * Obtem a instancia do jogo actual.
 *
 */
Game *Common::getGame()
{
	return game;
}

/**
 * Converte uma posicao na array em coordenadas do
 * mundo opengl.
 * E necessario fazer ajustamentos porque as coordenadas
 * convertidas tem como origem a posicao (0,0).
 */
float Common::tileToPos(int tile)
{
	return tile * MAP_TILE_SIZE;
}

/**
 * Obtem as coordenadas "ideais" para o clipping que
 * sera usado no glortho baseado no tamanho do nivel.
 * Permite arredondar para o numero inteiro acima.
 */
float Common::getClippingFromTile(int tile, bool upperRound)
{
	if(upperRound)
		return ceil(tile * 0.1 + 0.5);
	else
		return tile * 0.1;
}
```

Game.h

```
#include "Common.h"

#ifndef GAME_H
#define GAME_H

/**
 *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 camerasentre outras tarefas.
 *
 */
class Game
{
public:
	Game(int width, int height);
	~Game(void);

	void SetupGL(int argc, char **argv);
	void SetupCameras();

	static void Draw();
	static void reshape (int width, int height);
	static void keyboardEvent(unsigned char c, int x, int y);
	void DrawUI();
	static void Update(int value);
	void Event(int event);
	int getState();
	void setState(int state);

	void drawAxis();

	int getWidth();
	int getHeight();

	void setWidth(int w);
	void setHeight(int h);

	Level * getLevel();
	void setLevel(Level *level);
	Player *getPlayer();

	void setPlayer(Player * player);

	Camera *getCamera();
	Camera *getCamera(int i);
	void setCamera(int i);

	void drawText(char *text, float x, float y, float r, float g, float b, void * font);

	void UpdateViewport(GLint x, GLint y, GLsizei width, GLsizei height);
	int getCurrentCameraId();
```

```
	void setCurrentLevelId(int id);
	int getCurrentLevelId();

private:
	Level * currentLevel;
	Player * currentPlayer;
	Camera * camera[2];
	int	currentCamera;
	int	width;
	int	height;
	int	windowHandle;
	int	state;
	int	currentLevelId;
};

#endif
```

Game.cpp

```
#include "Common.h"

Game::Game(int width, int height)
{
	this->currentPlayer = NULL;
	camera[0] = NULL;
	camera[1] = NULL;
	this->width = width;
	this->height = height;
	this->currentLevel = NULL;
	this->currentLevelId = 1;
	windowHandle = -1;
	state = GAME_STATE_NOT_PLAYING;
	currentCamera = 1;
}

Game::~Game(void)
{
	delete currentPlayer;
	delete currentLevel;
	delete camera[0];
	delete camera[1];
}
```

```
/**
 * Desenha os eixos.
 *
 */
void Game::drawAxis()
{
	glPushMatrix();
	glBegin(GL_LINES);

	//x
	glColor3f(1,0,0);
	glVertex3f(-0.8,0,0);
	glVertex3f(0.8,0,0);

	//y
	glColor3f(0,1,0);
	glVertex3f(0,0.8,0);
	glVertex3f(0,-0.8,0);

	//z
	glColor3f(0,0,1);
	glVertex3f(0,0,0.8);
	glVertex3f(0,0,-0.8);

	glEnd();
	glPopMatrix();
}

/**
 * Configuracao inicial do opengl.
 *
 */
void Game::SetupGL(int argc, char **argv)
{
	glutInit(&argc, argv);
	glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE );
	glutInitWindowSize(this->width, this->height);
	glutInitWindowPosition(this->width / 2, this->height / 2);
	windowHandle = glutCreateWindow("Trabalho 1 Computacao Grafica");
	glutDisplayFunc(Game::Draw);
	glutTimerFunc(10, Game::Update, 10);
	glutKeyboardFunc(Game::keyboardEvent);
	glutReshapeFunc(Game::reshape);
	//actualizar o viewport inicial
	Common::getGame()->UpdateViewport(0, 0, Common::getGame()->getWidth(),
Common::getGame()->getHeight());
}
```

```
/**
 * Configura as cameras.
 *
 */
void Game::SetupCameras()
{
	this->camera[0] = new NormalCamera();
	this->camera[1] = new IsometricCamera();
}

/**
 * Actualiza o viewport.
 *
 */
void Game::UpdateViewport(GLint x, GLint y, GLsizei width, GLsizei height)
{
	glViewport(x, y, width, height);
}

/**
 * Funcao do glut que permite actualizar continuamente
 * o ecra.
 */
void Game::Update(int timeElapsed)
{
	glutPostRedisplay();
	glutTimerFunc(10, Game::Update, 10);
}

/**
 * Desenha a parte do interface grafico, como por exemplo
 * os objectivos do nivel.
 */
void Game::DrawUI()
{
	if(this->getState() == GAME_STATE_NOT_PLAYING)
	{
		drawText("Prima enter para jogar ou escape para sair", -0.7, 0, 1, 1, 1,
GLUT_BITMAP_HELVETICA_18);
		drawText("Use a tecla C para mudar de camera", -0.7, -0.1, 1, 1, 1,
GLUT_BITMAP_HELVETICA_18);
		drawText("Use a tecla R para repetir o nivel", -0.7, -0.2, 1, 1, 1,
GLUT_BITMAP_HELVETICA_18);
	}

	if(this->getState() == GAME_STATE_PLAYING)
	{
		if(getPlayer() && getPlayer()->getObjectiveState() == EVENT_NONE)
		{
			drawText("Procure o quadrado ", -1 , 0.9, 1, 1, 1,
GLUT_BITMAP_TIMES_ROMAN_24);
		  drawText("vermelho", -0.2 , 0.9, 1, 0, 0, GLUT_BITMAP_TIMES_ROMAN_24);
		}
```

```
                if(getPlayer() && getPlayer()->getObjectiveState() ==
EVENT_PLAYER_REACHED_RED)
                {
                        drawText("Procure o quadrado ", -1 , 0.9, 1, 1, 1,
GLUT_BITMAP_TIMES_ROMAN_24);
                   drawText("verde", -0.2 , 0.9, 0, 1, 0, GLUT_BITMAP_TIMES_ROMAN_24);
                }
                if(getPlayer() && getPlayer()->getObjectiveState() ==
EVENT_PLAYER_REACHED_GREEN)
                {
                        drawText("Procure o quadrado ", -1 , 0.9, 1, 1, 1,
GLUT_BITMAP_TIMES_ROMAN_24);
                   drawText("azul", -0.2 , 0.9, 0, 0, 1, GLUT_BITMAP_TIMES_ROMAN_24);
                }
        }
        else if(this->getState() == GAME_STATE_PLAYER_WON)
        {
                drawText("Parabens, chegaste ao fim do jogo!",  -0.7 , 0.1, 1, 1, 1,
GLUT_BITMAP_HELVETICA_18);
                drawText("Prime enter para voltares ao ecra principal",  -0.7 , 0, 1, 1, 1,
GLUT_BITMAP_HELVETICA_18);
        }
}

/**
 * Obtem a instancia actual do jogador.
 *
 */
Player *Game::getPlayer()
{
        return this->currentPlayer;
}

/**
 * Atribui a instancia actual do jogador para
 * acesso posterior.
 */
void Game::setPlayer(Player * player)
{
        this->currentPlayer = player;
}

/**
 * Atribui a instancia actual do nivel para
 * acesso posterior.
 */
void Game::setLevel(Level *level)
{
        this->currentLevel = level;
}
```

```
/**
 * Atribui a largura do ecra.
 *
 */
void Game::setWidth(int w)
{
	this->width = w;
}

/**
 * Atribui a altura do ecra.
 *
 */
void Game::setHeight(int h)
{
	this->height = h;
}

/**
 * Atribui o id do nivel actual.
 *
 */
void Game::setCurrentLevelId(int id)
{
	this->currentLevelId = id;
}

/**
 * Obtem o id do nivel actual.
 *
 */
int Game::getCurrentLevelId()
{
	return this->currentLevelId;
}

/**
 * Gere os eventos do teclado.
 *
 */
void Game::keyboardEvent(unsigned char c, int x, int y)
{
	float delta = 0.2f;
	if(c == '\x1b' && Common::getGame()->getState() == GAME_STATE_NOT_PLAYING)
	{
		exit(0);
	}
	else if(c == '\x1b' && Common::getGame()->getState() == GAME_STATE_PLAYING)
	{
		Common::getGame()->Event(EVENT_PLAYER_NOT_PLAYING);
	}

	else if(c == '\r' && Common::getGame()->getState() == GAME_STATE_PLAYER_WON)
```

```
	{
		Common::getGame()->Event(EVENT_PLAYER_NOT_PLAYING);
	}
	//start the game
	else if(c == '\r' && Common::getGame()->getState() == GAME_STATE_NOT_PLAYING)
	{
		Player * player = new Player();

		string cLevel = "media/nivel/nivel";
		cLevel += "1";
		cLevel += ".txt";

		Level *level = new Level((char*)cLevel.c_str());
		Common::getGame()->setLevel(level);
		level->LoadLevel();

		float cX = Common::getClippingFromTile( Common::getGame()->getLevel()-
>getTileSizeX(), false);
		float cY = Common::getClippingFromTile( Common::getGame()->getLevel()-
>getTileSizeY(), false);


		Common::getGame()->setPlayer(player);
		Common::getGame()->getPlayer()->setPosition( Common::tileToPos(level-
>getPlayerStartX()) - cX,   (cY - 0.2) - Common::tileToPos(level->getPlayerStartY()), 0.0f);
		Common::getGame()->setCamera(1);
		Common::getGame()->getCamera()->prepareCamera();
		player->setColor(0.0, 0.0, 0.0);

		Common::getGame()->Event(EVENT_PLAYER_PLAYING);
	}
	else if(Common::getGame()->getState() == GAME_STATE_PLAYING)
	{
		vector3 v =  Common::getGame()->getPlayer()->getPosition();

		if(c == 'r')
		{
			//recarregar o nivel actual
			string cLevel = "media/nivel/nivel";
			char buffer[100];
			_itoa_s( Common::getGame()->getCurrentLevelId(),buffer, 10);
			cLevel += buffer;
			cLevel += ".txt";

			//apagar o nivel actual
			delete Common::getGame()->getLevel();
			Common::getGame()->setLevel(NULL);
			Level *level = new Level(cLevel);

			//carregar o proximo nivel se existir.
			if(level->LoadLevel() == ERROR_NONE)
			{
```

```
                                Common::getGame()->setCurrentLevelId(  Common::getGame()->getCurrentLevelId());
                                Common::getGame()->setLevel(level);

                                float cX = Common::getClippingFromTile( Common::getGame()->getLevel()->getTileSizeX(), false);
                                float cY = Common::getClippingFromTile( Common::getGame()->getLevel()->getTileSizeY(), false);

                                //reset dos objectivos e posicao
                                Common::getGame()->getPlayer()->setObjectiveState(EVENT_NONE);
                                Common::getGame()->getPlayer()->setPosition( Common::tileToPos(level->getPlayerStartX()) - cX,   (cY - 0.2) - Common::tileToPos(level->getPlayerStartY()), 0.0f);
                        }
                        else
                        {
                                //erro critico, nunca devera acontecer
                                Common::getGame()->Event(EVENT_PLAYER_NOT_PLAYING);
                        }
                }

                if(c == 'c')
                {
                        Common::getGame()->setCamera( (Common::getGame()->getCurrentCameraId() + 1) % 2);
                        Common::getGame()->getCamera()->prepareCamera();

                }

                if(c == 'd')
                {
                        if(!Common::getGame()->getLevel()->Collides(v.x + delta, v.y))
                        {
                                Common::getGame()->getLevel()->UpdateLevelModeAI(v.x, v.y);
                                v.x = v.x + delta;
                                Common::getGame()->getPlayer()->setPosition(v);
                        }
                }
                if(c == 'a')
                {
                        if(!Common::getGame()->getLevel()->Collides(v.x - delta, v.y))
                        {
                                Common::getGame()->getLevel()->UpdateLevelModeAI(v.x, v.y);
                                v.x = v.x - delta;
                                Common::getGame()->getPlayer()->setPosition(v);
                        }
                }
                if(c == 'w')
                {
                        if(!Common::getGame()->getLevel()->Collides(v.x, v.y + delta))
                        {
```

```
                                Common::getGame()->getLevel()->UpdateLevelModeAI(v.x, v.y);
                                v.y = v.y + delta;
                                Common::getGame()->getPlayer()->setPosition(v);
                        }
                }
                if(c == 's')
                {
                        if(!Common::getGame()->getLevel()->Collides(v.x, v.y - delta))
                        {
                                Common::getGame()->getLevel()->UpdateLevelModeAI(v.x, v.y);
                                v.y = v.y - delta;
                                Common::getGame()->getPlayer()->setPosition(v);
                        }
                }
                //debugging
                //printf("px: %f py: %f\n",  v.x, v.y);
        }
}

/**
 * Obtem o estado do jogo actual.
 *
 */
int Game::getState()
{
        return state;
}

/**
 * Gere os eventos lancados no jogo.
 *
 */
void Game::Event(int event)
{
        switch(event)
        {
        case EVENT_PLAYER_NOT_PLAYING:
                {
                        Common::getGame()->setCurrentLevelId(1);
                        delete this->currentPlayer;
                        this->currentPlayer = NULL;
                        delete this->currentLevel;
                        this->currentLevel = NULL;
                        state = GAME_STATE_NOT_PLAYING;
                } break;
        case EVENT_PLAYER_PLAYING:
                {
                        state = GAME_STATE_PLAYING;
                } break;
                //FSM muita feia, era preferivel ter 1 flag com os bits necessarios para transitar
de estado zzzzz.
        case EVENT_PLAYER_REACHED_BLUE:
                {
```

```
                        if(getPlayer()->getObjectiveState() ==
EVENT_PLAYER_REACHED_GREEN)
                                getPlayer()-
>setObjectiveState(EVENT_PLAYER_REACHED_BLUE);
                }
                break;
        case EVENT_PLAYER_REACHED_GREEN:
                {
                        if(getPlayer()->getObjectiveState() == EVENT_PLAYER_REACHED_RED)
                                getPlayer()-
>setObjectiveState(EVENT_PLAYER_REACHED_GREEN);
                }
                break;
        case EVENT_PLAYER_REACHED_RED:
                {
                        if(getPlayer()->getObjectiveState() == 0)
                                getPlayer()->setObjectiveState(EVENT_PLAYER_REACHED_RED);
                }
                break;
        case EVENT_PLAYER_FINISHED_LEVEL:
                {
                        //load next level ou vitoria
                        string cLevel = "media/nivel/nivel";
                        char buffer[100];
                        _itoa_s( Common::getGame()->getCurrentLevelId() + 1,buffer, 10);
                        cLevel += buffer;
                        cLevel += ".txt";

                        //apagar o nivel actual
                        delete Common::getGame()->getLevel();
                        this->currentLevel = NULL;
                        Level *level = new Level(cLevel);

                        //carregar o proximo nivel se existir.
                        if(level->LoadLevel() == ERROR_NONE)
                        {
                                Common::getGame()->setCurrentLevelId(  Common::getGame()-
>getCurrentLevelId() + 1);
                                Common::getGame()->setLevel(level);

                                float cX = Common::getClippingFromTile( Common::getGame()-
>getLevel()->getTileSizeX(), false);
                                float cY = Common::getClippingFromTile( Common::getGame()-
>getLevel()->getTileSizeY(), false);

                                Common::getGame()->getPlayer()-
>setObjectiveState(EVENT_NONE);
                                Common::getGame()->getPlayer()->setPosition(
Common::tileToPos(level->getPlayerStartX()) - cX,   (cY - 0.2) - Common::tileToPos(level-
>getPlayerStartY()), 0.0f);
                                Common::getGame()->setCamera(1);
                                Common::getGame()->getCamera()->prepareCamera();
```

```
                                state = GAME_STATE_PLAYING;
                        }
                        else
                        {
                                //fim de jogo.
                                state = GAME_STATE_PLAYER_WON;
                        }
                }
                break;

        default:break;
        };
}

/**
 * Obtem a altura do ecra.
 *
 */
int Game::getHeight()
{
        return this->height;
}

/**
 * Obtem a largura do ecra.
 *
 */
int Game::getWidth()
{
        return this->width;
}

/**
 * Obtem a instancia do nivel actual.
 *
 */
Level * Game::getLevel()
{
        return currentLevel;
}

/**
 * Obtem a camera actual activa.
 *
 */
Camera *Game::getCamera()
{
        return this->camera[currentCamera];
}
```

```
/**
 * Atribui o id de uma das cameras disponiveis.
 *
 */
void Game::setCamera(int i)
{
	this->currentCamera = i;
}

/**
 * Obtem o id da camera actual.
 *
 */
int Game::getCurrentCameraId()
{
	return this->currentCamera;
}

/**
 * Obtem a camera actual activa dado o seu índice.
 *
 */
Camera *Game::getCamera(int i)
{
	return this->camera[i];
}

/**
 * Permite desenhar texto no ecra.
 *
 */
void Game::drawText(char *text, float x, float y, float r, float g, float b, void * font)
{
	glMatrixMode(GL_PROJECTION);
	glPushMatrix();
	glLoadIdentity();
	glDisable(GL_TEXTURE_2D);
	glColor3f(r,g,b);
	glRasterPos2f(x, y);
	for(unsigned int i = 0; i < strlen(text); i++)
		glutBitmapCharacter(font, text[i]);
	glEnable(GL_TEXTURE_2D);
	glPopMatrix();
	glMatrixMode(GL_MODELVIEW);
}
```

```
/**
 * Faz o draw do nivel.
 * Desenha a parte visivel e iteractiva no front buffer.
 * Desenha o nivel no backbuffer para deteccao de colisoes.
 */
void Game::Draw()
{
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
	//game logic
	//jogador atingiu os objectivos.


	if(Common::getGame()->getState() == GAME_STATE_NOT_PLAYING)
	{
		Common::getGame()->UpdateViewport(0, 0, Common::getGame()->getWidth(),
Common::getGame()->getHeight());
		Common::getGame()->DrawUI();
		glutSwapBuffers();

	}
	else if(Common::getGame()->getState() == GAME_STATE_PLAYER_WON)
	{
		Common::getGame()->UpdateViewport(0, 0, Common::getGame()->getWidth(),
Common::getGame()->getHeight());
		Common::getGame()->DrawUI();
		glutSwapBuffers();

	}
	else if(Common::getGame()->getState() == GAME_STATE_PLAYING)
	{
		if(Common::getGame()->getPlayer() && Common::getGame()->getPlayer()-
>getObjectiveState() == EVENT_PLAYER_REACHED_BLUE)
		{
			Common::getGame()->Event(EVENT_PLAYER_FINISHED_LEVEL);
			glutSwapBuffers();
			return;
		}

		assert(Common::getGame() && Common::getGame()->getLevel() &&
Common::getGame()->getPlayer());
		Common::getGame()->getCamera()->prepareCamera();
		Common::getGame()->UpdateViewport(0, 0, Common::getGame()->getWidth(),
Common::getGame()->getHeight());
		//Common::getGame()->drawAxis();
		//desenhar o nivel
		Common::getGame()->currentLevel->Update();
		//desenhar o jogador
		Common::getGame()->getPlayer()->Update();
		Common::getGame()->DrawUI();

		// minimap apenas para a segunda camera
		if(Common::getGame()->getCurrentCameraId() == 1)
		{
```

```
                        Common::getGame()->getCamera(0)->prepareCamera();
                        Common::getGame()->UpdateViewport(0, 0, 150, 150);
                        Common::getGame()->currentLevel->Update();
                        Common::getGame()->getPlayer()->Update();
                }

                glutSwapBuffers();
                //desenhar no backbuffer.
                glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
                //desenhar o nivel
                Common::getGame()->currentLevel->Update();
        }
}


void Game::reshape (int width, int height)
{
        Common::getGame()->setHeight(height);
        Common::getGame()->setWidth(width);
}
```

IsometricCamera.h

```
#include "Common.h"

#ifndef ISOMETRICCAM_H
#define ISOMETRICCAM_H

/**
 * Classe que prepara uma camera com projecção isométrica.
 *
 */
class IsometricCamera : public Camera
{
public:
        IsometricCamera(void);
        ~IsometricCamera(void);

        void prepareCamera();
};
#endif
```

IsometricCamera.cpp

```
#include "IsometricCamera.h"


IsometricCamera::IsometricCamera(void)
{
}
```

```
IsometricCamera::~IsometricCamera(void)
{
}

/**
 * Prepara a camera em modo isometrico
 *
 */
void IsometricCamera::prepareCamera()
{
	float cX, cY, cS;

	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	cX = Common::getClippingFromTile( Common::getGame()->getLevel()->getTileSizeX(),
true);
	cY = Common::getClippingFromTile( Common::getGame()->getLevel()->getTileSizeY(),
true);
	cS = (cX > cY ? cX : cY);
	glOrtho(-cS, cS, -cS, cS, -cS, cS);
	gluLookAt(0.25, 0.25, 0.25, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0);
	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
}
```

Level.h

```
#include "Common.h"

#ifndef LEVEL_H
#define LEVEL_H
/**
 *Contem métodos que permitem carregar um nível e desenha-lo no ecrã.
 *
 */
class Level
{
public:
	Level(string filename);
	~Level(void);

	void Update();
	int LoadLevel();
	void ReadContent(char *buffer, int row);
	int getPlayerStartX();
	int getPlayerStartY();
	int getTileSizeX();
	int getTileSizeY();

	bool Collides(float x, float y);

	int **getLevelData();
```

```
	void setLevelMode(int m);
	int getLevelMode();
	void UpdateLevelModeAI(float px, float py);


private:
	string filename;
	int levelMode;
	int **data;
	int size_x;
	int size_y;
	int start_x;
	int start_y;
};

#endif
```

Level.cpp

```
#include "Level.h"

Level::Level(string filename)
{
	this->filename = filename;
	this->data = NULL;
	this->size_x = 0;
	this->size_y = 0;
	this->start_x = 0;
	this->start_y = 0;
}

Level::~Level(void)
{
	for(int i = 0; i < size_x; i++)
	{
		delete[] data[i];
	}
	delete[] data;
}

/**
 * Carrega os dados de um nivel caso seja possivel.
 *
 */
int Level::LoadLevel()
{
	std::ifstream stream;

	stream.open(filename, std::ifstream::in);
	if(!stream.is_open())
	{
```

```
		return ERROR_FAILED_TO_LOAD_LEVEL;
	}

	string buffer;
	//ler o tamanho
	getline(stream, buffer);
	string s_x = buffer.substr(0, buffer.find(" "));
	string s_y = buffer.substr(buffer.find(" "));
	string s_mode = buffer.substr(buffer.find(" ") + 3, buffer.find(" "));
	int l_mode = atoi(s_mode.c_str());
	this->levelMode = l_mode;

	size_x = atoi(s_x.c_str());
	size_y = atoi(s_y.c_str());

	data = new int*[size_x];

	for(int i = 0; i < size_x; i++)
	{
		data[i] = new int[size_y];
	}

	for(int i = 0; i < size_x; i++)
	{
		for(int j = 0; j < size_y; j++)
		{
			data[i][j] = 0;
		}
	}

	int line = 0;
	while(!stream.eof())
	{
		char bufferb[255];
		//ler os dados do nivel
		stream.getline(bufferb, 255);
		//sacar os dados
		ReadContent(bufferb, line);
		line++;
	}

	stream.close();

	return ERROR_NONE;
}
```

```
/**
 * Obtem a posicao x do jogador na array do tabuleiro.
 * Apenas util para operacoes de conversao para coordenadas
 * do mundo opengl ou para cameras.
 *
 */
int Level::getPlayerStartX()
{
	return this->start_x;
}

/**
 * Obtem a posicao y do jogador na array do tabuleiro.
 * Apenas util para operacoes de conversao para coordenadas
 * do mundo opengl ou para cameras.
 *
 */
int Level::getPlayerStartY()
{
	return this->start_y;
}

/**
 * Le o conteudo de cada linha do ficheiro.
 *
 */
void Level::ReadContent(char *buffer, int row)
{
	int column = 0;
	while(*buffer != '\0')
	{
		char c = *buffer;
		switch(c)
		{
		case 'B':
			data[column][row] = LEVEL_PATH_BLOCKED;
			break;
		case 'P':
			data[column][row] = LEVEL_PATH_PASSABLE;
			break;
		case 'R':
			data[column][row] = LEVEL_PATH_OBJECTIVE_RED;
			break;
		case 'V':
			data[column][row] = LEVEL_PATH_OBJECTIVE_GREEN;
			break;
		case 'A':
			data[column][row] = LEVEL_PATH_OBJECTIVE_BLUE;
			break;
		case 'S':
			data[column][row] = LEVEL_PATH_PLAYER_START;
			this->start_x = column;
			this->start_y = row;
```

```
                        break;
                default:
                        printf("dados desconhecidos ao ler o nivel\n");
                        break;
                };
                column++;
                buffer++;
        }
}

/**
 * Obtem a largura do tabuleiro que sera usado para
 * criar arrays bidimensionais.
 *
 */
int Level::getTileSizeX()
{
        return this->size_x;
}

/**
 * Obtem a altura do tabuleiro que sera usado para
 * criar arrays bidimensionais.
 *
 */
int Level::getTileSizeY()
{
        return this->size_y;
}

/**
 * Desenha o nivel.
 *
 */
void Level::Update()
{
        float screenAdjustmentX = Common::getClippingFromTile(size_x, false);
        float screenAdjustmentY = Common::getClippingFromTile(size_y, false);

        //draw the level
        int modColorPicker = 1;
        float squareSize = 0.2f;
        //desenhar os quadrados excepto os que estao bloqueados
        glPushMatrix();
        for(int i = 0; i < size_y; i++)
        {
                if( size_x % 2 == 0) { modColorPicker = (modColorPicker + 1) % 2; }
                for(int j = 0; j < size_x; j++)
                {

                        //caminho onde se pode passar.
                        switch(this->data[j][i])
                        {
```

```
                        case LEVEL_PATH_BLOCKED:

                                //glColor3f(1, 1, 1);
                                //glRectf( j * squareSize - screenAdjustmentX, i * -squareSize +
screenAdjustmentY, j * squareSize + squareSize - screenAdjustmentX, i * -squareSize -
squareSize + screenAdjustmentY);
                                break;
                        case LEVEL_PATH_PASSABLE:
                                modColorPicker ? glColor3f(0.5, 0.5, 0.5) : glColor3f(0.3, 0.3, 0.3);
                                glRectf( j * squareSize - screenAdjustmentX, i * -squareSize +
screenAdjustmentY, j * squareSize + squareSize - screenAdjustmentX, i * -squareSize -
squareSize + screenAdjustmentY);
                                break;
                        case LEVEL_PATH_OBJECTIVE_BLUE:
                                glColor3f(0.0, 0.0, 1.0);
                                glRectf( j * squareSize - screenAdjustmentX, i * -squareSize +
screenAdjustmentY, j * squareSize + squareSize - screenAdjustmentX, i * -squareSize -
squareSize + screenAdjustmentY);
                                break;
                        case LEVEL_PATH_OBJECTIVE_RED:
                                glColor3f(1.0, 0.0, 0.0);
                                glRectf( j * squareSize - screenAdjustmentX, i * -squareSize +
screenAdjustmentY, j * squareSize + squareSize - screenAdjustmentX, i * -squareSize -
squareSize + screenAdjustmentY);
                                break;
                        case LEVEL_PATH_OBJECTIVE_GREEN:
                                glColor3f(0.0, 1.0, 0.0);
                                glRectf( j * squareSize - screenAdjustmentX, i * -squareSize +
screenAdjustmentY, j * squareSize + squareSize - screenAdjustmentX, i * -squareSize -
squareSize + screenAdjustmentY);
                                break;
                        case LEVEL_PATH_PLAYER_START:
                                glColor3f(1.0, 1.0, 0.0);
                                glRectf( j * squareSize - screenAdjustmentX, i * -squareSize +
screenAdjustmentY, j * squareSize + squareSize - screenAdjustmentX, i * -squareSize -
squareSize + screenAdjustmentY);
                                break;
                        default:
                                printf("Aviso: dados desconhecidos ao desenhar o nivel\n");
                                break;
                        };
                        modColorPicker = (modColorPicker + 1) % 2;
                }
        }
        glPopMatrix();
}
```

```
/**
 * Retorna a array que contem os dados do nivel.
 *
 */
int ** Level::getLevelData()
{
	return this->data;
}

/**
 * Atribui o modo de jogo a este nivel.
 *
 */
void Level::setLevelMode(int m)
{
	this->levelMode = m;
}

/**
 * Retorna o modo de jogo para este nivel.
 *
 */
int Level::getLevelMode()
{
	return this->levelMode;
}

/**
 *Verifica se as coordenadas dadas colidem com um
 *tile bloqueado.
 *Usa pixel color colision pelo backbuffer.
 *Lanca tambem os eventos que sejam detectados.
 */
bool Level::Collides(float x, float y)
{
	float data[3];
	GLint viewport[4];
	GLdouble modelview[16];
	GLdouble projection[16];
	glGetIntegerv(GL_VIEWPORT, viewport);
	glGetDoublev( GL_MODELVIEW_MATRIX, modelview );
	glGetDoublev( GL_PROJECTION_MATRIX, projection );

	GLdouble winx, winy, winz;
	glutPostRedisplay();
	glReadBuffer(GL_BACK);
	gluProject(x + 0.1, y + 0.1, 0.0, modelview, projection,viewport, &winx, &winy, &winz);
	glReadPixels(winx, winy, 1, 1, GL_RGB, GL_FLOAT, data);

	if(data[0] == 0.0f && data[1] == 0.0f && data[2] == 0.0f)
	{
		return true;
	}
```

```
	else if(data[0] == 0.0f && data[1] == 0.0f && data[2] == 1.0f)
	{
		Common::getGame()->Event(EVENT_PLAYER_REACHED_BLUE);
	}
	else if(data[0] == 1.0f && data[1] == 0.0f && data[2] == 0.0f)
	{
		Common::getGame()->Event(EVENT_PLAYER_REACHED_RED);
	}
	else if( data[0] == 0.0f && data[1] == 1.0f && data[2] == 0.0f)
	{
		Common::getGame()->Event(EVENT_PLAYER_REACHED_GREEN);
	}
	return false;
}

/**
 * Permite mudar a jogabilidade do nivel baseado no seu modo
 * de jogo.
 *
 */
void Level::UpdateLevelModeAI(float px, float py)
{
	//elimina os quadrados onde o jogador ja passou
	if(Common::getGame()->getLevel()->getLevelMode() == LEVEL_MODE_USE_BRAIN)
	{
		float cX = Common::getClippingFromTile( Common::getGame()->getLevel()-
>getTileSizeX(), false);
		float cY = Common::getClippingFromTile( Common::getGame()->getLevel()-
>getTileSizeY(), false);
		int tx = abs(Common::roundToTile(px + cX));
		int ty = abs(Common::roundToTile(py - cY + 0.2));
		Common::getGame()->getLevel()->getLevelData()[tx][ty] =
LEVEL_PATH_BLOCKED;
	}
}
```

NormalCamera.cpp

```
#include "Common.h"

#ifndef NORMALCAMERA_H
#define NORMALCAMERA_H

/**
 * Classe que prepara uma camera com projecção ortonormada(paralela).
 *
 */
class NormalCamera : public Camera
{
public:
	NormalCamera(void);
	~NormalCamera(void);
```

```
	void prepareCamera();
};

#endif
```

NormalCamera.h

```
#include "NormalCamera.h"


NormalCamera::NormalCamera(void)
{
}


NormalCamera::~NormalCamera(void)
{
}


/**
 * Prepara a camera em modo nao isometrico.
 *
 */
void NormalCamera::prepareCamera()
{
	float cX, cY, cS;

	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	cX = Common::getClippingFromTile( Common::getGame()->getLevel()->getTileSizeX(),
false);
	cY = Common::getClippingFromTile( Common::getGame()->getLevel()->getTileSizeY(),
false);
	cS = (cX > cY ? cX : cY) + 0.2f;
	glOrtho(-cS, cS, -cS, cS, -cS, cS);

	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();

}
```

Player.h

```
#include "Common.h"

#ifndef PLAYER_H
#define PLAYER_H
```

```
class Player
{
public:
	Player(void);
	~Player(void);

	int getObjectiveState();
	void setObjectiveState(int state);
	void setPosition(float x, float y, float z);
	void setPosition(vector3 pos);
	void setColor(float r, float g, float b);
	vector3& getPosition();
	color& getColor();
	void Update();
private:
	int objectiveState;
	vector3 position;
	color color;
};

#endif
```

Player.cpp

```
#include "Common.h"


Player::Player(void)
{
	this->objectiveState = EVENT_NONE;
}


Player::~Player(void)
{
}

/**
 * Retorna o estado dos objectivos alcancados pelo
 * jogador.
 *
 */
int Player::getObjectiveState()
{
	return this->objectiveState;
}
```

```
/**
 * Actualiza os objectivos alcancados pelo jogador.
 *
 */
void Player::setObjectiveState(int state)
{
	this->objectiveState = state;
}

/**
 * Retorna a posicao actual do jogador.
 *
 */
vector3& Player::getPosition()
{
	return position;
}

/**
 * Retorna a cor do jogador.
 *
 */
color& Player::getColor()
{
	return color;
}


/**
 * Atribui a posicao do jogador.
 *
 */
void Player::setPosition(vector3 pos)
{
	position.x = pos.x;
	position.y = pos.y;
	position.z = pos.z;
}

/**
 * Atribui a posicao do jogador.
 *
 */
void Player::setPosition(float x, float y, float z)
{
	position.x = x;
	position.y = y;
	position.z = z;
}
```

```
/**
 * Atribui a cor do jogador.
 *
 */
void Player::setColor(float r, float g, float b)
{
	color.r = r;
	color.g = g;
	color.b = b;
}


/**
 * Actualiza os dados do jogador.
 * Permite desenhar o jogador no mundo.
 *
 */
void Player::Update()
{
	float radius = 0.1f;

	//draw aqui
	//draw do circulo
	glPushMatrix();
	glColor3f(1.0, 1.0, 1.0);
	glTranslatef(this->position.x, this->position.y, 0.0f);
	glBegin(GL_POLYGON);
	for(float i = 0; i < (2 * PI); i += PI / 24)
		glVertex3f(cos(i) * radius + 0.1,  sin(i) * radius + 0.1, 0.0);
	glEnd();
	glPopMatrix();
}
```

Cg_trab1.cpp

```
#include "Common.h"

int main(int argc, char **argv)
{
	Common *common = new Common();
	Common::getGame()->SetupGL(argc, argv);
	Common::getGame()->SetupCameras();

	glutMainLoop();
	delete common;

	return 0;
}
```

Conclusão

O trabalho não foi deveras complicado, mas penso que os conhecimentos adquiridos até agora foram uma maior valia, visto que agora já se tem uma ideia de como construir um simples jogo, que embora sejam simples, as técnicas aplicadas neste trabalho, podem ser usadas para construir jogos decentes (jogos de plataformas, rpg 2d, etc.). Penso que a parte mais difícil deste trabalho foi sem dúvida encontrar uma estrutura para desenhar os níveis de forma eficaz sem estar a programar o nível inteiro no próprio jogo ou a por correcções a nível de estética.

De notar que foi fornecida alguma ajuda a alguns colegas que se encontravam em dificuldades, especialmente na parte das colisões e no desenho do nível no ecrã, logo poderá aparecer algum código com a mesma ideia base. O código que permite fazer o draw do texto neste trabalho foi feito com algum auxílio de alguns colegas na parte em que o texto está sempre na mesma posição do ecrã, independentemente da camera.

Referências

1. http://www.opengl.org/discussion_boards/showthread.php/160784-Drawing-Circles-in-OpenGL (acedido em 30/3/2013)
2. http://www.opengl.org/sdk/docs/man2/xhtml/glOrtho.xml (acedido em 13/4/2013)