Obligatorio 1

Introducción a la Computación Gráfica - 2020

Videojuego sobre OpenGL

Documentación Técnica

Autores
Nestor Etcheverry: 4.201.362-3
Santiago Ferrando: 4.895.212-0
Ignacio Perdomo: 4.421.913-4

Docentes
José Pedro Aguerre
Eduardo Fernandez

Índice

Arquitectura de la aplicación

La aplicación está compuesta de 4 grupos de componentes con competencias bien definidas, los cuales son Core, Lógica, Render y Auxiliares.

El grupo Core está compuesto por los componentes Main, GameManager e InputHandler, que tienen como cometido ejecutar el bucle principal del juego, recibir la entrada de teclado y mouse, actualizar el estado del tablero y llamar al componente de render.

El grupo Lógica está compuesto por los componentes Map, Ficha, Enemy, Player, Settings y ScoreKeeper cuya función es la de mantener actualizado el estado general del juego.

El grupo Render se compone de los componentes Camera y OpenGL Util los cuales se encargan de leer el estado del juego (GameManager) y dibujar en pantalla ese estado.

Por último tenemos el grupo de Auxiliares compuesto por los componentes OBJLoader, Vector3, LATimer y Direction cuya función es ser soporte para las actividades de los otros 3 grupos.

Pasamos a describir en detalle la competencia de cada componente.
Main

- Tiene el bucle principal
- Actualiza el estado del juego mediante el GameManager.
- Controla la condición de victoria y derrota.
- Llama a InputHandler para procesar la entrada.
- Controla el framerate tanto por exceso como por defecto. El framerate está seteado en 60 en el componente Settings, si un loop del bucle demora menos que 1/60 segundos, se espera el tiempo restante para mantener constante el framerate. En caso contrario, si demora más de 1/60 segundos en completar un bucle, el siguiente solamente actualizará el estado del juego mediante el GameManager y salteará el renderizado de ese frame.

GameManager

- Tiene el estado del Mapa, la distribución de cubos en el espacio así como también por cuales cubos qbert ya pasó y por cuáles no (mediante Map).
- Controla el reloj del juego.
- Tiene control sobre Player y Enemy.
- Detecta colisiones a nivel lógico.
- Actualiza el estado del tablero en función de la entrada (desde InputHandler).
- Procesa la condición de victoria (mediante Map).

InputHandler

- Recibe la entrada y la deriva a GameManager o Settings según corresponda.

Map

- Conoce la posición de cada cubo.
- Conoce cuales cubos están pintados y cuáles no.
- Sabe si un movimiento es válido o no.

Ficha

- Es una superclase de Player y Enemy, conoce la posición y dirección de las Fichas en el tablero.
- Sabe actualizar las posiciones lógicas (no discretas, de movimiento) de las fichas.

Enemy

- subclase de Ficha, sabe como dibujar el modelo 3D del enemigo.

Player

- subclase de Ficha, sabe como dibujar el modelo 3D del jugador.

Settings

- Tiene la configuración de los FPS y el estado de las opciones gamespeed, wireframe, texturas, interpolado/facetado y manejo de luces.

Camera

- Controla la cámara del juego en las 3 modalidades solicitadas en la letra.

OpenGLUtil

- Componente encargado de renderizar a nivel físico, esto quiere decir que toma el estado del juego de GameManager y traduce del mundo lógico (abstracción del juego) al mundo físico (lo que se ve en pantalla). Para lograr este cometido además del GameManager se comunica con Map, Camera, Enemy, Player y Settings.

OBJLoader

- Componente encargado de abrir y cargar los archivos obj de modelo del Player y el Enemy.

Vector3

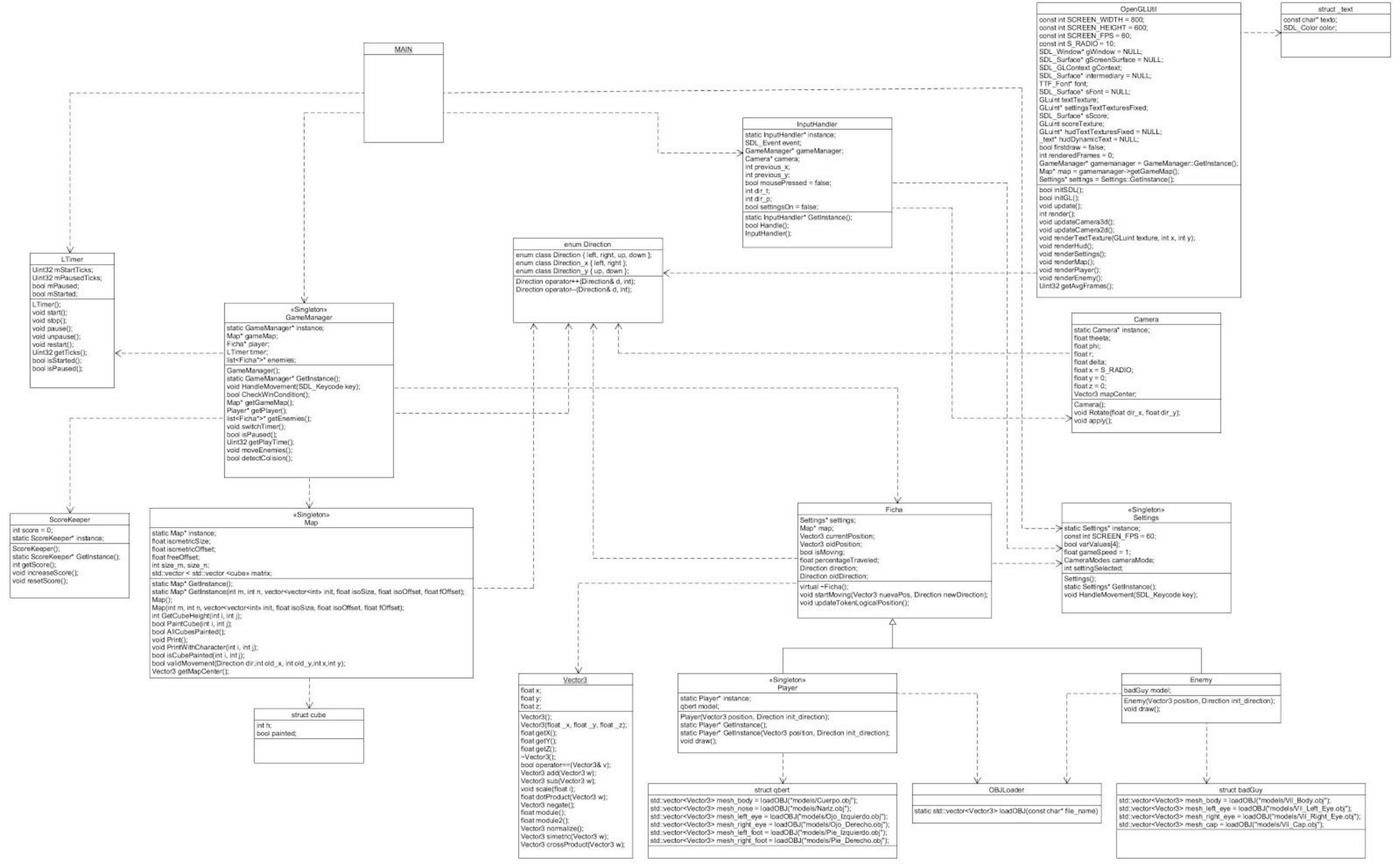
- Coordenadas lógicas utilizadas para modelar el mundo lógico.

LTimer

- Componente encargado de llevar el tiempo de juego, así como también controlar el tiempo entre frames para realizar el framecap.

Direction

- Dirección lógica hacia donde está mirando el Player así como también el Enemy.


MAIN
OpenGLUtil
const int SCREEN_WIDTH = 800;
const int SCREEN_HEIGHT = 600;
const int SCREEN_FPS = 60;
const int S_RADIO = 10;
SDL_Window* gWindow = NULL;
SDL_Surface* gScreenSurface = NULL;
SDL_GLContext gContext;
SDL_Surface* intermediary = NULL;
TTF_Font* font;
SDL_Surface* sFont = NULL;
GLuint textTexture;
GLuint* settingsTextTexturesFixed;
SDL_Surface* sScore;
GLuint scoreTexture;
GLuint* hudTextTexturesFixed = NULL;
_text* hudDynamicText = NULL;
bool firstdraw = false;
int renderedFrames = 0;
GameManager* gamemanager = GameManager::GetInstance();
Map* map = gamemanager->getGameMap();
Settings* settings = Settings::GetInstance();
bool initSDL();
bool initGL();
void update();
int render();
void updateCamera3d();
void updateCamera2d();
void renderTextTexture(GLuint texture, int x, int y);
void renderHud();
void renderSettings();
void renderMap();
void renderPlayer();
void renderEnemy();
Uint32 getAvgFrames();
struct _text
const char* texto;
SDL_Color color;
InputHandler
static InputHandler* instance;
SDL_Event event;
GameManager* gameManager;
Camera* camera;
int previous_x;
int previous_y;
bool mousePressed = false;
int dir_t;
int dir_p;
bool settingsOn = false;
static InputHandler* GetInstance();
bool Handle();
InputHandler();
enum Direction
enum class Direction { left, right, up, down };
enum class Direction_x { left, right };
enum class Direction_y { up, down };
Direction operator++(Direction& d, int);
Direction operator--(Direction& d, int);
LTimer
Uint32 mStartTicks;
Uint32 mPausedTicks;
bool mPaused;
bool mStarted;
LTimer();
void start();
void stop();
void pause();
void unpause();
void restart();
Uint32 getTicks();
bool isStarted();
bool isPaused();
«Singleton»
GameManager
static GameManager* instance;
Map* gameMap;
Ficha* player;
LTimer timer;
list<Ficha*>* enemies;
GameManager();
static GameManager* GetInstance();
void HandleMovement(SDL_Keycode key);
bool CheckWinCondition();
Map* getGameMap();
Player* getPlayer();
list<Ficha*>* getEnemies();
void switchTimer();
bool isPaused();
Uint32 getPlayTime();
void moveEnemies();
bool detectColision();
Camera
static Camera* instance;
float theeta;
float phi;
float r;
float delta;
float x = S_RADIO;
float y = 0;
float z = 0;
Vector3 mapCenter;
Camera();
void Rotate(float dir_x, float dir_y);
void apply();
ScoreKeeper
int score = 0;
static ScoreKeeper* instance;
ScoreKeeper();
static ScoreKeeper* GetInstance();
int getScore();
void increaseScore();
void resetScore();
«Singleton»
Map
static Map* instance;
float isometricSize;
float isometricOffset;
float freeOffset;
int size_m, size_n;
std::vector < std::vector <cube» matrix;
static Map* GetInstance();
static Map* GetInstance(int m, int n, vector<vector<int» init, float isoSize, float isoOffset, float fOffset);
Map();
Map(int m, int n, vector<vector<int» init, float isoSize, float isoOffset, float fOffset);
int GetCubeHeight(int i, int j);
bool PaintCube(int i, int j);
bool AllCubesPainted();
void Print();
void PrintWithCharacter(int i, int j);
bool isCubePainted(int i, int j);
bool validMovement(Direction dir,int old_x, int old_y,int x,int y);
Vector3 getMapCenter();
Ficha
Settings* settings;
Map* map;
Vector3 currentPosition;
Vector3 oldPosition;
bool isMoving;
float percentageTraveled;
Direction direction;
Direction oldDirection;
virtual ~Ficha();
void startMoving(Vector3 nuevaPos, Direction newDirection);
void updateTokenLogicalPosition();
«Singleton»
Settings
static Settings* instance;
const int SCREEN_FPS = 60;
bool varValues[4];
float gameSpeed = 1;
CameraModes cameraMode;
int settingSelected;
Settings();
static Settings* GetInstance();
void HandleMovement(SDL_Keycode key);
struct cube
int h;
bool painted;
Vector3
float x;
float y;
float z;
Vector3();
Vector3(float _x, float _y, float _z);
float getX();
float getY();
float getZ();
~Vector3();
bool operator==(Vector3& v);
Vector3 add(Vector3 w);
Vector3 sub(Vector3 w);
void scale(float i);
float dotProduct(Vector3 w);
Vector3 negate();
float module();
float module2();
Vector3 normalize();
Vector3 simetric(Vector3 w);
Vector3 crossProduct(Vector3 w);
«Singleton»
Player
static Player* instance;
qbert model;
Player(Vector3 position, Direction init_direction);
static Player* GetInstance();
static Player* GetInstance(Vector3 position, Direction init_direction);
void draw();
Enemy
badGuy model;
Enemy(Vector3 position, Direction init_direction);
void draw();
struct qbert
std::vector<Vector3> mesh_body = loadOBJ("models/Cuerpo.obj");
std::vector<Vector3> mesh_nose = loadOBJ("models/Nariz.obj");
std::vector<Vector3> mesh_left_eye = loadOBJ("models/Ojo_Izquierdo.obj");
std::vector<Vector3> mesh_right_eye = loadOBJ("models/Ojo_Derecho.obj");
std::vector<Vector3> mesh_left_foot = loadOBJ("models/Pie_Izquierdo.obj");
std::vector<Vector3> mesh_right_foot = loadOBJ("models/Pie_Derecho.obj");
OBJLoader
static std::vector<Vector3> loadOBJ(const char* file_name)
struct badGuy
std::vector<Vector3> mesh_body = loadOBJ("models/Vil_Body.obj");
std::vector<Vector3> mesh_left_eye = loadOBJ("models/Vil_Left_Eye.obj");
std::vector<Vector3> mesh_right_eye = loadOBJ("models/Vil_Right_Eye.obj");
std::vector<Vector3> mesh_cap = loadOBJ("models/Vil_Cap.obj");

Soluciones gráficas

Dibujado de cubos

Para el dibujado de cubos utilizamos glBegin(GL_QUADS), definiendo los 8 vértices. Para la ubicación, se lee el mapa de Map donde se lee la posición lógica discreta (x,y,z) de cada cubo y esta posición se traduce en los 8 vértices del cubo.

Dibujado de Player y Enemy

Para el dibujado del Player y el Enemy, se realizaron modelos 3D en la aplicación tinkercad y se exportan como archivos .obj, luego se cargaron desde el archivo con el componente OBJLoader, interpretando cada fila y cargando los vértices en vectores que luego fueron dibujados con glBegin y glEnd.
Para el movimiento se utilizan las transformaciones glTranslatef y glRotatef, primero posicionando los dibujos en el origen, luego se le realizan las transformaciones necesarias para que modelen el estado actual de la partida.

Iluminación

Se utilizaron tres luces, una ambiente y dos difusas.
La luz ambiente que comienza con parámteros de color (0.5,0.5,0.5) y se puede cambiar presionando la tecla L, cambiando secuencialmente entre el anteriormente mencionado color, rojo, verde y azul.
Luego, una de las luces difusas está ubicada en la posición (0,0,10,1.0) y color (1.0,0.6,0.6) la cual no es modificable por el usuario. La otra luz difusa tiene posición (0.0,8.0,0.0,0.0) o (8.0,0.0,0.0), modificable con la letra C y color (0.01,0.01,0.01), lo cual hace que se forme una sombra a un lado u otro del mapa dependiendo de en qué posición se encuentre.

Interpolado/facetado

El interpolado se habilita con el comando glShadeModel(GL_SMOOTH) y el facetado con glShadeModel(GL_FLAT). Realizamos un interpolado en las caras laterales de los cubos del mapa con 2 vertices rojos y 2 negros.

Camara/Vistas

Existen dos formas de abordar el control de la cámara: mover la escena para mostrar una vista distinta de ésta mientras la cámara se mantiene fija, o mover la cámara para que capte una vista distinta de la escena; se optó por el segundo enfoque.
En la vista isométrica se multiplica la matriz de proyección por una matriz ortográfica mediante el comando glOrtho. Se mantiene fija la cámara, buscando que el ángulo entre el vector formado por el punto donde se encuentra la cámara y el punto al que mira, sea cuarenta y cinco grados respecto a cada eje; esto se logra ubicando la posición del “ojo” de

la cámara a una misma distancia en los tres ejes. Se configura un offset para la posición del mapa en la pantalla (para subir o bajar el mapa) y un tamaño de mapa.
Tanto en la vista libre como en tercera persona se utiliza proyección en perspectiva. En la vista libre se obtiene la posición y acción del mouse gracias a InputHandler, si existe algún cambio se rota la cámara cambiando la posición del ojo utilizando coordenadas esféricas, manteniendo el punto al que mira en el centro del mapa. Para la vista en tercera persona se coloca el ojo de la camara detras del personaje, así como el punto que mira en la posición del personaje, si este cambia de posición, el ojo se mueve con el.
En todas las vistas tanto el control del ojo de la cámara, como el punto a donde esta mira, se controló mediante gluLookAt. El vector *up* se configuró para los tres casos como (0,0,1).

Dibujado de textos

Para el dibujado en pantallas de los textos se utilizó la librería SDL_ttf. SDL_ttf se encarga de cargar una fuente true type en memoria, para después usarla para crear una SDL_surface con el texto pasado como parámetro; por último, después de generar la textura de opengl, la surface se puede cargar a una textura de opengl mediante glTexImage2D y una surface intermediaria que tenga seteados los canales adecuados para el pasaje. Como este proceso puede llegar a requerir demasiados recursos, se intenta cargar la mayor cantidad de texturas de texto en la inicialización del programa, en el método de inicialización de SDL. En el caso de texturas “dinámicas” que cambian durante ejecución (como el puntaje y el tiempo) se realiza este proceso mientras el programa se encuentra ejecutando, pero se tiene cuidado de liberar recursos (principalmente las SDL_surface).
Para hacer uso de la textura se bindea y dibuja como cualquier textura normal.

Librerías y referencias

Se utilizó SDL 2
La base del componente LTtimer fue sacada de https://lazyfoo.net/
OpenGL Red Book https://www.glprogramming.com/red/
Tutorial C++ OpenGL 3D
https://www.youtube.com/playlist?list=PL6xSOsbVA1eYSZTKBxnoXYboy7wc4yg-Z
Pasaje de SDL_Surface a OpenGL texture
http://www.sdltutorials.com/sdl-tip-sdl-surface-to-opengl-texture
Herramienta de modelado 3D
https://www.tinkercad.com/