POLITÉCNICO DE LEIRIA | ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

Bachelor Degree in Games and Multimedia

Advanced Game Programming Topics

Project 2

2D Game Engine with Xenon clone

Autores:

Francisco Domingues - 2162370

Miguel Monteiro - 2171005

Professor:

Gustavo Reis

Introdução

Para este projeto, foi-nos pedido que criássemos uma game engine de raiz em C++ utilizando os aplicativos *SDL2*, *Box2D* e OpenAL de modo a podermos desenvolver um clone do jogo Xenon 2000 nesta mesma engine.

Neste relatório, iremos detalhar todos os algoritmos que desenvolvemos para tornar isto possível, as nossas justificações por detrás do código que implementamos, os objetivos que atingimos neste projeto, aqueles que não atingimos e a lista de todos os sites e referências que pesquisamos para este projeto.

Game Engine

Objetivos:

Alcançados:

- Event Handling
- Keyboard and Gamepad support
- Game Loop (Logic e Rendering)
- Resource Management

Não alcançados:

- GLSL Shaders

Xenon Clone (jogo)

Objetivos:

Alcançados:

- Spaceship
- Missiles
- Enemies (loner, rusher, drone e asteroids)
- Enemy projectiles
- Companion
- Power Ups
- Vertical Scroller

Extra Credits

Alcançados:

- OpenAL
- Audio

Implemented Algorithms

Engine

Event Handling

Para lidar com toda a lógica dos eventos do SDL, criámos uma classe Input. Esta classe é um singleton que, em cada frame, toma conta de cada evento na event queue, desde teclas pressionadas a detetar um novo gamepad.

Input

O singleton input permite também saber o estado de cada tecla. Este sistema foi inspirado pelo Input Manager da engine Unity e consiste em 3 valores para cada tecla: *isKey*, *isKeyDown* e *isKeyUp*. Ao chamar *getKey()*, *getKeyDown()* ou *getKeyUp()*, e passando o nome da tecla pretendida, consegue-se implementar lógica dependendo se uma tecla está premida, se uma tecla foi premida neste frame ou se uma tecla foi levantada neste frame, respetivamente.

Para o conseguir, foi criada uma struct “Key” e, para cada tecla, foi criada uma variável do tipo Key, populada com as suas informações (nome e *keycode*) que tem as suas variáveis *isKey*, *isKeyDown* e *isKeyUp* atualizadas à medida que recebe eventos das respectivas teclas.

Gamepad Controller Support

Para que se possa usar um *gamepad controller* para controlar um objeto no jogo (neste caso a nave), foi implementada a classe "GameController" que é instanciada como um controller na classe Input e utiliza o mesmo tipo de lógica que foi utilizada nas teclas para os seus botões além de um método para obter o valor de cada eixo (que é dividido por um valor constante máximo, MAX_AXIS).

Rendering Implementation

Para implementar uma rendering pipeline em que a lógica do jogo corra em separado da lógica de rendering, foram criadas as classes Engine e Level. Na primeira classe foi implementado o game loop e na segunda foram criadas listas para guardar os objetos criados pelo jogo e funções que permitem fazer a gestão dos mesmos.

Resource Management

Nas várias classes do projeto tivemos cuidado com a gestão da memória, tentando-a libertar assim que já não seja necessária. Para este objetivo, sempre que é alocada memória para criar um objeto da classe actor, a mesma será libertada quando este for destruído, quer durante o decorrer do jogo quer no final quando o nível é terminado. Por sua vez, sempre que um *actor* é destruído, este destrói as *textures*, *tilemaps* e *animations* associadas ao mesmo, que foram adicionadas aos respetivos vetores no momento após serem alocadas.

Game Loop

O Game Loop encontra-se dentro da classe da Engine e é o que executa a lógica do jogo. Começa por calcular o *deltaTime* (tempo desde a última execução) e, de seguida, atualiza a física no *level* e lida com os *events* do *SDL*. Depois disto, corre o *update* de cada *actor* através da classe Level, atualiza as animações e de seguida desenha os *actors* e *canvases* no ecrã. No final, atualiza a lista de *actors*, *animations* e *canvases* guardadas no Level (removendo e adicionando *actors*, *animations* e *canvases* às respetivas listas) e atualiza o estado de corpos nas físicas.

Level

Para tarefas como a gestão de listas de *actors, animations* e *sounds*, a criação do mundo do Box2D (onde são criados os corpos dos objetos e simuladas as físicas) e a inicialização das ferramentas de áudio foi criada a classe “Level”.

Esta classe é guardada como variável na classe “Engine” de forma a que as suas funções possam ser chamadas no game loop, na altura adequada.

Adicionalmente é também na classe “Level” que existe a função para ler um ficheiro de som loadSoundFile() e a função para pôr a tocar um som playSound() que chamam funções da classe “OpenALWrapper” para implementar estas funcionalidades.

Actor

Para gerir os objetos criados no jogo foi criada a classe Actor que serve de parent para as classes criadas no jogo que representem um objeto do mesmo. Para que cada objeto desta classe seja adicionado à lista de actors guardada no Level, é chamada no construtor da classe a função addActor() do mesmo. Para o objeto ser removido quando necessário foi criada a função destroy() que é chamada no jogo sempre que um objeto precise de ser destruído e que, por sua vez, chama a função addActorToRemove() da classe Level que acrescenta o actor à lista de actors para serem removidos no final da execução do game loop.

Nesta classe foram criadas as funções “update()” e “render()” que são chamadas durante o game loop (através de funções com o mesmo nome na classe Level) para atualizar tanto a posição do objeto, como a representação gráfica do mesmo.

Pawn

Foi criada uma classe pawn para servir de base para o “player”. Esta classe controla o movimento do personagem jogável e é flexível de modo a poder ser estendida e reimplementada.

SDL2 Wrapper classes

Para manusear e utilizar métodos do *SDL2*, foram criadas várias classes para generalizar vários termos como *"Window"*, *"Texture"*, etc. Estes permitem ao game loop e ao jogo não ter contacto direto com o *SDL*, reduzindo a complexidade e aumentando a portabilidade.

Box2D Classes

De forma a poderem ser chamadas funções ao ser detectada uma colisão, foi criada a classe "ContactListener" que estende da classe "b2ContactListener" do Box2D que reimplementa a função "BeginContact" que é chamada no momento em que se inicia um contacto entre dois corpos do Box2D. Para ser usada nesta função foi criada a classe "ContactSensor" que tem a função virtual "onContact" que é chamada aqui sempre que um "ContactSensor" esteja presente num contacto. No caso de ambos os corpos serem *sensors*, é passada pela função "onContact" uma referência ao sensor com o qual este colidiu.

A classe "ContactSensor" serve, por sua vez, para servir de *parent class* para as classes em que queremos que haja algum código a ser executado quando uma colisão é detectada.

Para criar o corpo no qual as físicas vão ser simuladas foi criada a classe "RigidBody" que é usada nas classes em que queremos criar um corpo usando o Box2D. Nesta classe estão definidas 16 categorias (CATEGORY_0 até CATEGORY_15) que são usadas para filtrar as colisões, ou seja, definir a categoria de cada corpo e quais as categorias dos corpos com os quais este pode colidir, implementado na função setCollisionFilter().

Por último, foi criada também a classe "Projectile" que serve para criar um corpo que tenha as propriedades de um *bullet* do Box2D. Nesta classe é criado um "RigidBody" e é definida a velocidade a que esse corpo se começa a mover quando é criado.

Animation

A implementação de animações foi criada através das classes “Texture”, “Tilemap” e “Animation”. Na classe “Texture” são usadas funções do sdl2 para pegar num ficheiro de formato .bmp e criar uma “SDL_Texture” e são ainda guardados dois “Rect” para serem usados pelo renderer para desenhar a textura no ecrã.

Na classe “Tilemap”, através de uma referência à textura, e ao ser especificado o número de linhas e colunas do *tilemap*, é definido o tamanho de cada *tile*. Finalmente, no construtor da classe “Animation” é especificado um “Tilemap”, a ordem de animação das *tiles* pretendida e um valor *boolean* correspondente a se queremos uma animação em *loop* ou não.

Adicionalmente, para ser feita uma gestão de animações (semelhante à dos actors), são usadas as funções addAnimation() e addAnimationToRemove() da classe “Level” no construtor e no destrutor desta classe, respetivamente.

Para controlar o estado da animação foram criadas a funções play() (que faz a animação ser executada), playFromStart() (que reinicia e executa a animação) e stop() (que pára a execução da animação). A acrescentar, foi criada a função destroyAfterEnd() que recebe um “Actor” para destruir quando a animação chegar ao final da sua execução.

UI

Canvas

Para ter a certeza que a UI é rendered sempre por cima do jogo e continua a receber *Updates*, foi criada uma classe *UICanvas* que é semelhante à classe *Actor*, mas sem parâmetros de posição.

Todas as áreas de UI vão estar incluídas numa classe que se estende de *UICanvas*.

Text Rendering

Para fazer text rendering, criámos a classe *BitmapFont* para dividir o bitmap da fonte em cada um dos respectivos caracteres através do uso do tilemap.

Com a fonte agora no formato de tilemap, é possível criar uma nova fonte com o bitmap pretendido e, reutilizar esta mesma sempre que pretendemos escrever algo com essa fonte.

Para escrever o texto no ecrã, criámos o método “renderText” no qual passamos a posição pretendida para o texto e o texto que pretendemos escrever. Esta função corre cada caractere do texto e vai mudando o active frame do tilemap para copiar o caractere correto para o renderer.

Bar

Havia necessidade de criar um elemento que se repetisse um certo número de vezes, como uma barra de vida. Para isso, criámos a class *UIBar*,

Esta classe, funcionalmente, é bastante semelhante à classe *BitmapFont*, sendo criada através de uma textura e *tilemap*, que depois é desenhado no ecrã um segmento de cada vez de acordo com os parâmetros de comprimento (length), *frame* do *tilemap* e espaçamento entre cada repetição *(padding)*.

Game

Collision Filtering

Para filtrar as colisões entre os corpos de objetos no jogo foram usadas as categorias definidas na classe “RigidBody” como se pode ver na tabela abaixo.

Categoria	Objeto	Categorias com que colide
1	Player	2 | 4 | 5 | 6
2	Enemy	1 | 3 | 6
3	Missile	2
4	Enemy Projectile	1 | 6
5	Power Up	1 | 6
6	Companion	1 | 2 | 4 | 5

Spaceship

A “Spaceship” é o nosso jogador. Ela estende da classe “Pawn” e é composta pelas texturas e animações necessárias, além de receber o input do jogador para se mover.

De forma a que a nave faça uma animação quando está a virar para a direita e outra quando está a virar para a esquerda, foram criadas as animações moveRightAnim, returnRightAnim, moveLeftAnim e returnLeftAnim que não dão loop e que representam os movimentos da nave a virar quando está direita e a endireitar quando está virada para uma direção. Estes movimentos são detectados no update através da velocidade horizontal atualizada da nave, chamando as funções animateMoveRight() ou animateMoveLeft() se houver velocidade numa destas direções e as funções animateReturnRight() e animateReturnLeft() se a velocidade for nula. Estas funções, quando chamadas pela primeira vez em cada alteração ao movimento, chamam a função playFromStart() da respectiva animação de forma a esta ser executada a partir do início.

Para implementar o disparo da nave, foi criada a função fire() que é chamada ao ser detectado o respectivo input e, depois de verificar se pode disparar através do valor da variável boolean canFire, cria um objeto da classe “Missile” numa posição em frente à nave com a velocidade definida em missileVelocity. Se a nave tiver companions ativos, é criado mais um míssil na posição de cada companion. O tempo que a nave tem de esperar entre disparos é controlado pela função cooldownCheck() que é executada na thread cooldownThread e que enquanto a nave estiver viva (condição controlada pela variável boolean isAlive) verifica se a função fire() acabou de ser executada (através da variável boolean needsCooldown), em intervalos de 10 milissegundos e, nesse caso, faz a thread esperar a quantidade de milissegundos especificada pela variável cooldown (neste caso 300 milissegundos) antes atribuir o valor true à variável canFire e, consequentemente, permitindo a nave disparar mais uma vez.

Missiles and Enemy Projectiles

No caso das classes *“Missile”* e e “*EnemyProjectile*”, foram atribuídas as classes *“Actor”* e *“Projectile”* como *parent classes*, de forma a que o míssil/projétil possa ter uma textura, possa implementar a função *“onContact()”* e seja criado um *rigidbody* com as propriedades de um míssil/projétil, com valores como a posição, tamanho, densidade e velocidade a serem passados pelo construtor da classe.

Para implementar a explosão de cada míssil/projétil, foi criado uma textura, tilemap e animação adicionais que usam o ficheiro explode16.bmp onde está representada uma explosão. Ao ser detectada uma colisão, esta textura substitui a do míssil e é executada a sua animação. De forma a que o míssil/projétil seja destruído apenas quando a animação tiver terminado, é utilizada a função destroyAfterEnd() da classe "Animation" com uma referência para o míssil/projétil em si.

Se um míssil detetar um contacto com um inimigo, é chamada a função takeDamage() da classe "Enemy" para danificar o inimigo com a quantidade de dano definido na variável missileDamage que é modificada baseado no tipo de míssil definido no construtor da classe para ter o valor definido na variável lightDamage, mediumDamage ou heavyDamage.

Enemies and Hazards

Para implementar inimigos no jogo foi criada a classe *"Enemy"* (*child class* de Actor) que serve de *parent class* para as classes *"Loner"*, *"Rusher"*, "*Drone*" e "*Asteroid*". Nestas classes é depois, no construtor, atribuída a respectiva textura, *tilemap* e é definida a animação correspondente ao inimigo e criado um corpo dinâmico do *Box2D* ao qual é atribuída a respetiva velocidade.

No caso da classe "*Asteroid*" como existem dois tipos de asteróides e três tamanhos para cada, foram criados os enums "*AsteroidType*" (com os valores stone e metal) e "*AsteroidSize*" (com os valores small, medium ou large). Baseado nos valores de variáveis do tipo destes enums passadas no construtor da classe, são atribuídas uma textura e uma animação correspondentes a cada asteroide.

De forma a que os inimigos retirem vida à nave quando colidem com a mesma, é chamada função takeDamage() da classe "Spaceship" para retirar uma quantidade de vida à nave definida na variável attackDamage.

Power Ups

Para implementar os vários power ups do jogo, foi criada a classe "PowerUp" para servir de parent class às classes dos power ups, onde está criada a função virtual applyPower() que é chamada na classe "Spaceship" quando esta colide com um power up e é implementada nas classes de cada power up para aplicar o efeito desejado à nave, especificado abaixo:

ShieldPowerUp

Nesta classe é implementada a função applyBonus() que chama a função addHealth() da classe “Spaceship” que acrescenta vida à nave.

WeaponPowerUp

Nesta classe é chamada a função upgradeMissile() que altera o valor da variável missileType (enum criado na classe “Missile” que define três tipos de missiles: light, medium e heavy) baseado no valor atual, passando a medium se for light e a heavy se for medium.

CompanionPowerUp

Nesta classe é chamada a função attachCompanion() que acrescenta o companion à lista da nave o que faz com que a posição do mesmo seja atualizada para junto da nave e a sua velocidade com o valor da velocidade da nave.

Spawner

Para controlar o aparecimento de inimigos e power ups no nível foi criada a classe “Spawner” onde estão definidos os intervalos entre o cada *spawn* de cada tipo de inimigos e power ups. Esta classe tem como *parent* a classe “Actor”, de forma a poder usar o update para atualizar os valores das *cooldowns* de cada objeto usando a variável deltaTime. No caso de o *cooldown* dos inimigos ter acabado é chamada a função spawnEnemies() que usando um número aleatório entre 0 e 3 decide que tipo de inimigo tenta dar *spawn*, reiniciando o *cooldown* caso tenha sucesso, sendo que não tiver sucesso, o número vai ser gerado as vezes necessárias até suceder. O mesmo método é utilizado com os power ups e a função spawnPowerUp().

De forma a que as posições em que um inimigo ou power up aparece sejam aleatórias, antes de dar *spawn* a um inimigo, é gerado um número aleatório para a posição com as coordenadas relevantes ao tipo de objeto e depois o mesmo é criado utilizando essa posição.

Vertical Scroller

O vertical scroller foi implementado ao criar vários objetos da classe ScrollingBackground que deslizam lentamente pelo ecrã a uma velocidade previamente definida.

Level

O ficheiro “Xenon.cpp” foi usado para criar o nível do jogo, onde é criada a variável engine, e a variável mainLevel que define uma gravidade nula para o nível. De seguida é usada a variável engine para chamar a função “init()” com o nome e dimensão da janela. De seguida é criado o *background*, o game manager e a nave, o spawner e a UI que depois vão ser geridos no game loop que se encontra na função “start()” do engine que é chamada a seguir.

UI

A UI foi composta por duas fontes e duas barras. Fontes estas foram utilizadas para escrever os vários textos e pontuações no ecrã e as barras foram utilizadas para criar a barra de vida e o número de vidas restantes.

Criando um bitmap com 3 cores, conseguimos também mudar a cor da barra de vida dependendo da quantidade de vida restante.

Game Manager

Para auxiliar o controle das vidas do jogador e guardar a pontuação, criámos um *singleton GameManager*. Este *singleton* é acedido pela UI para se informar com os valores necessários (como vida, número de vidas, pontuação e pontuação máxima).

Quando o jogador morre, vai ser *spawnado* novamente se tiver vidas restantes após alguns segundos através do *GameManager*.

Extra Credits

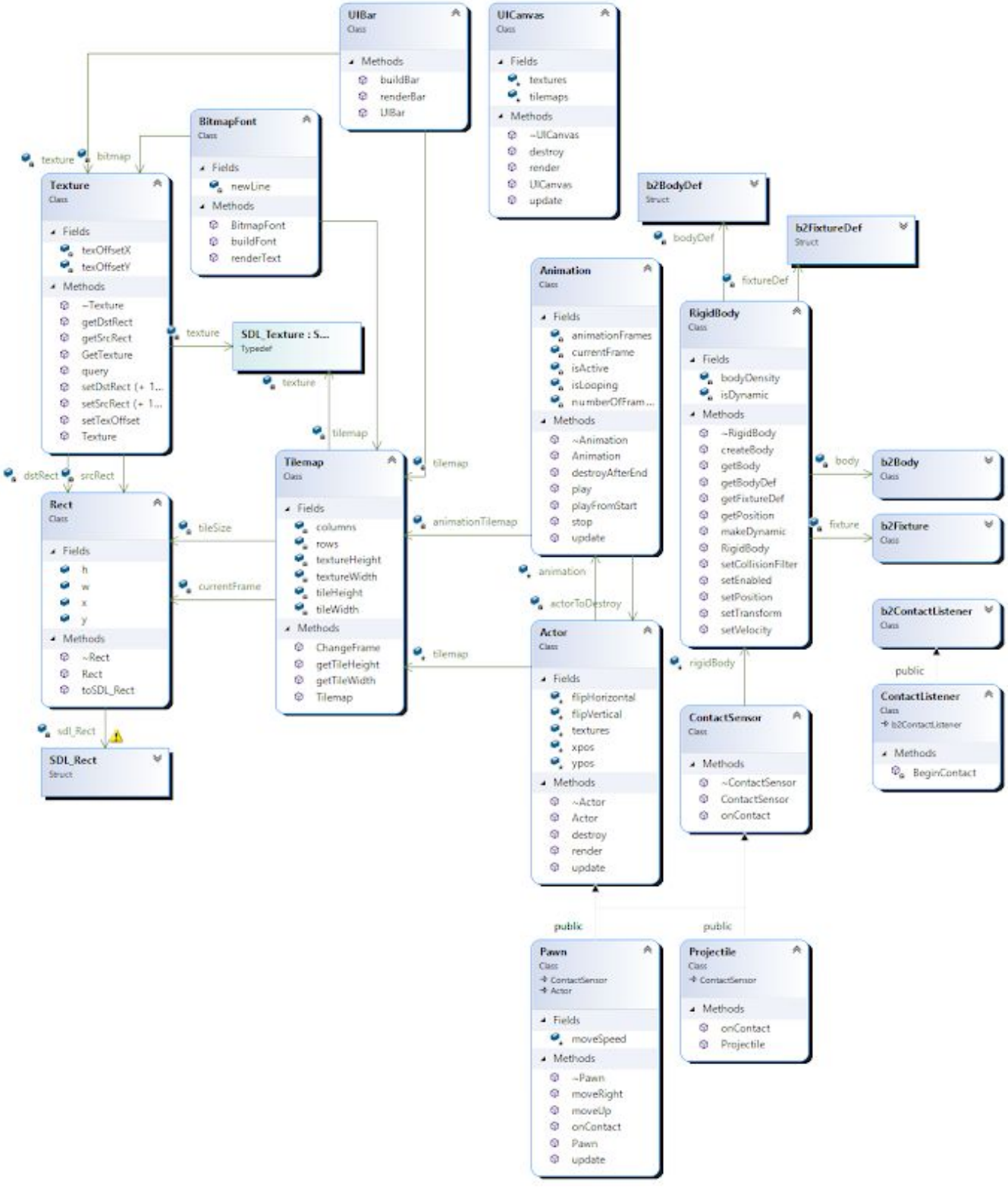
OpenAL Wrapper classes

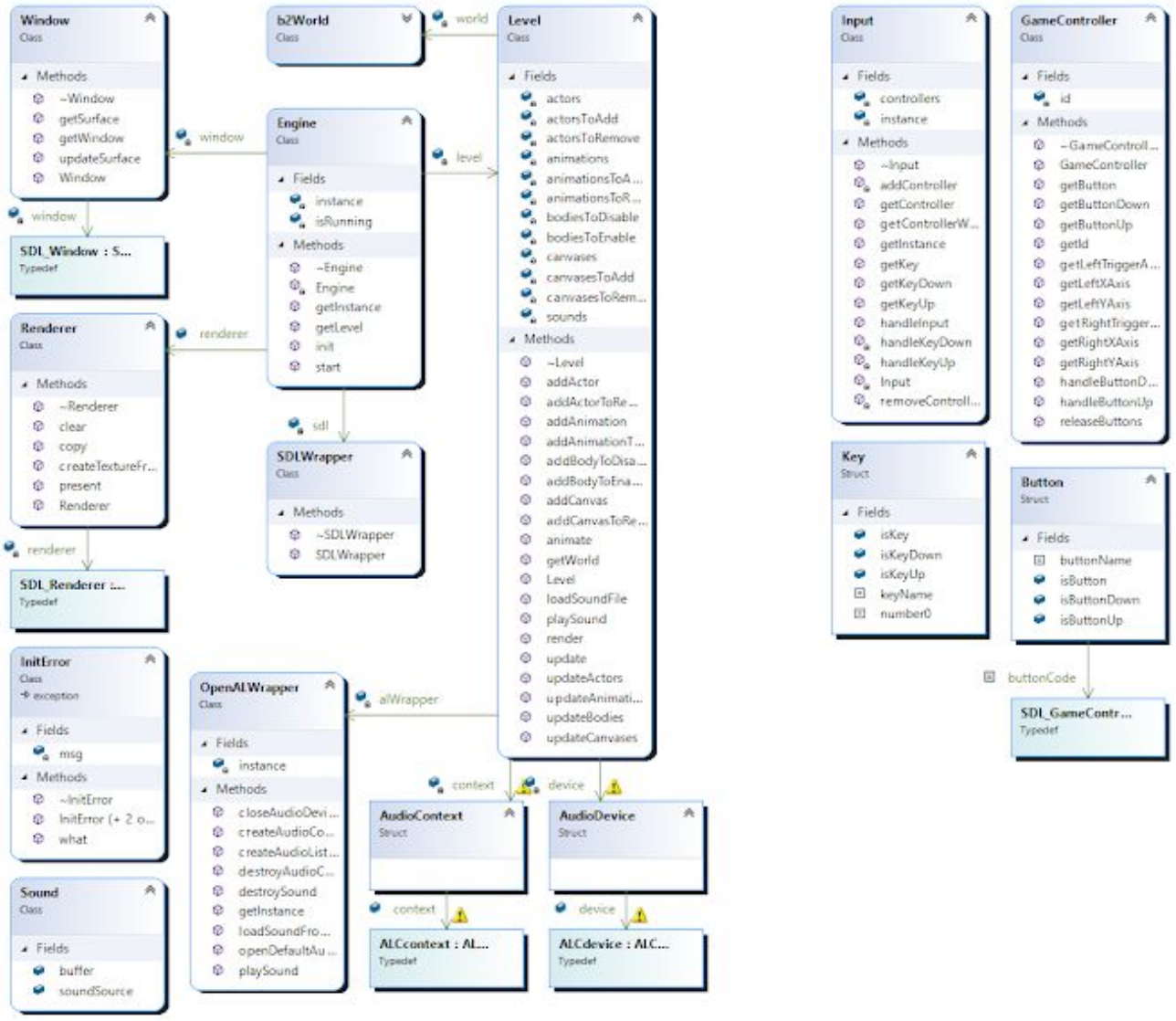
Para implementar o uso de áudio no jogo, foi utilizada a biblioteca “OpenAL Soft” que usa sintaxe semelhante ao OpenGL e que permite usar os dispositivos de áudio para criar um contexto em que possam ser tocados sons. Foi também usada a biblioteca AudioFile para conseguir ler ficheiros de som com a extensão .wav.

De forma a poderem ser chamadas funções destas bibliotecas no nosso engine, foi criada a classe “OpenALWrapper” com funções para abrir e fechar o dispositivo de áudio predefinido, criar e destruir um *context*, criar um *Listener*, ler um ficheiro de som e tocar e apagar um som.

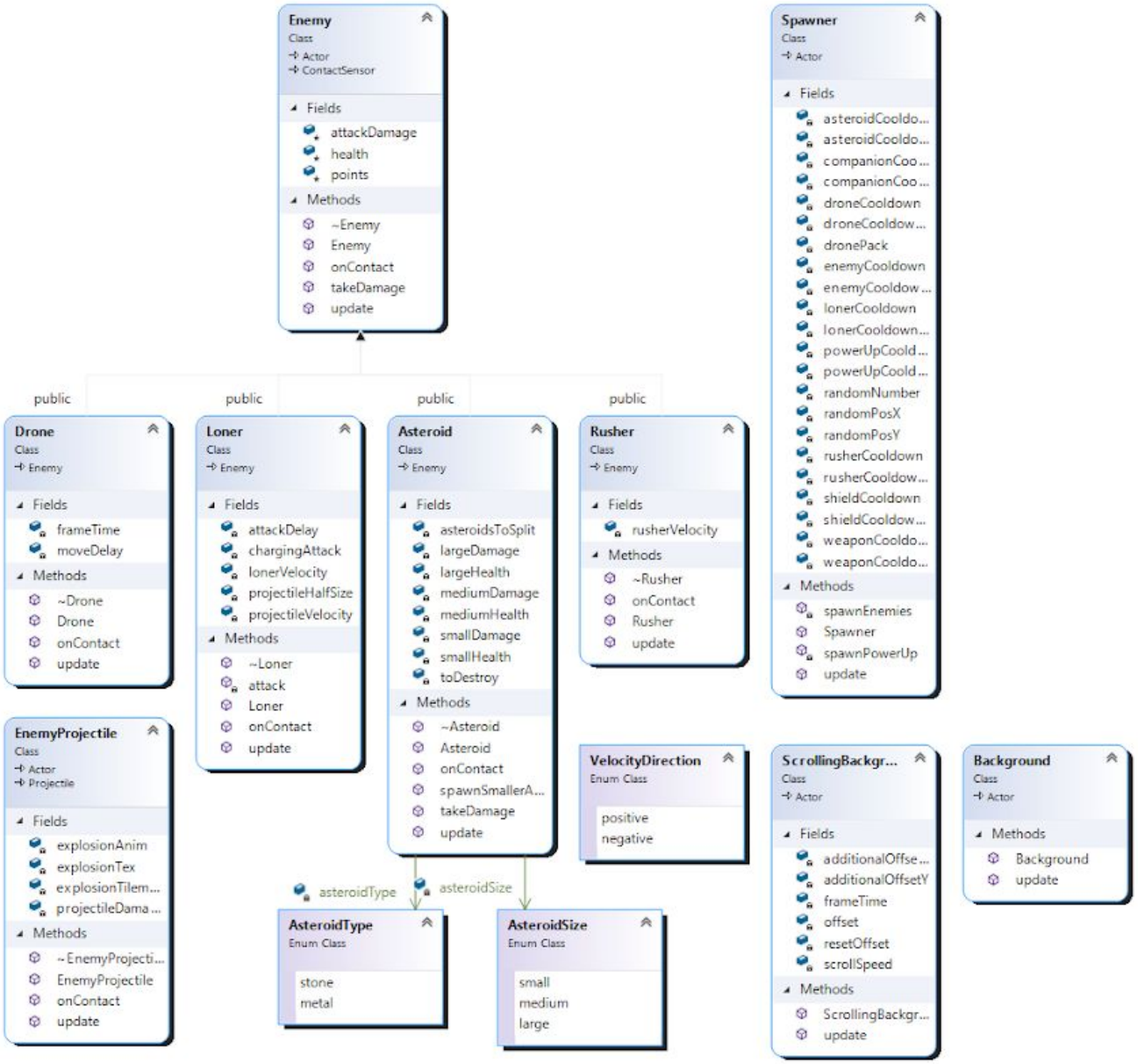
Por último, foi criada a classe “Sound” para ser usada para guardar o identificador de um som e o de uma *source*.

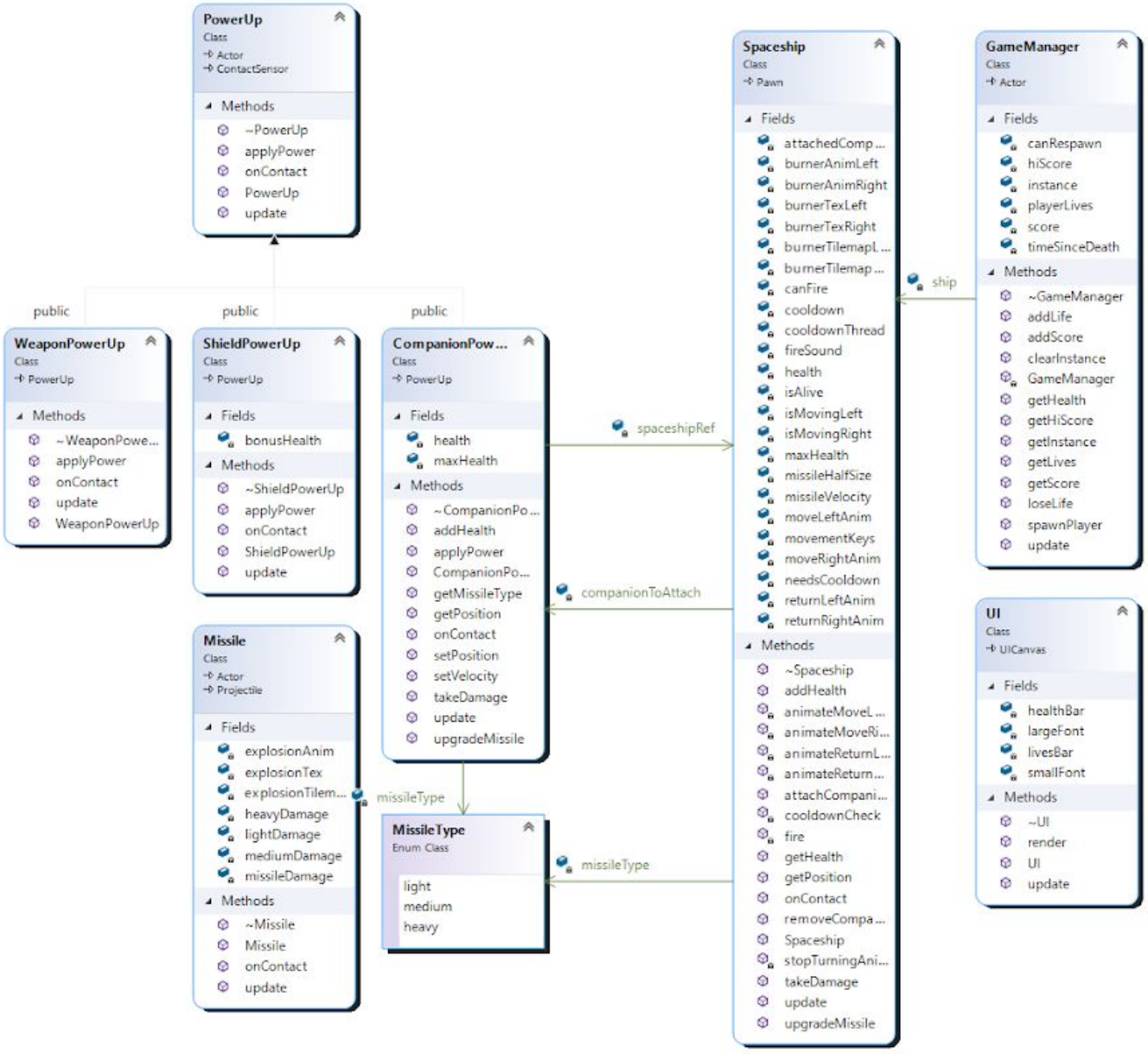
Engine Class Diagram




Window
Class
Methods
~Window
getSurface
getWindow
updateSurface
Window
window
SDL_Window : S...
Typedef
Renderer
Class
Methods
~Renderer
clear
copy
createTextureFr...
present
Renderer
renderer
SDL_Renderer :...
Typedef
InitError
Class
exception
Fields
msg
Methods
~InitError
InitError (+ 2 o...
what
Sound
Class
Fields
buffer
soundSource
b2World
Class
world
Engine
Class
Fields
instance
isRunning
Methods
~Engine
Engine
getInstance
getLevel
init
start
window
level
renderer
sdl
SDLWrapper
Class
Methods
~SDLWrapper
SDLWrapper
OpenALWrapper
Class
Fields
instance
Methods
closeAudioDevi...
createAudioCo...
createAudioList...
destroyAudioC...
destroySound
getInstance
loadSoundFro...
openDefaultAu...
playSound
alWrapper
Level
Class
Fields
actors
actorsToAdd
actorsToRemove
animations
animationsToA...
animationsToR...
bodiesToDisable
bodiesToEnable
canvases
canvasesToAdd
canvasesToRem...
sounds
Methods
~Level
addActor
addActorToRe...
addAnimation
addAnimationT...
addBodyToDisa...
addBodyToEna...
addCanvas
addCanvasToRe...
animate
getWorld
Level
loadSoundFile
playSound
render
update
updateActors
updateAnimati...
updateBodies
updateCanvases
context
device
AudioContext
Struct
AudioDevice
Struct
context
device
ALCcontext : AL...
Typedef
ALCdevice : ALC...
Typedef
Input
Class
Fields
controllers
instance
Methods
~Input
addController
getController
getControllerW...
getInstance
getKey
getKeyDown
getKeyUp
handleInput
handleKeyDown
handleKeyUp
Input
removeControll...
Key
Struct
Fields
isKey
isKeyDown
isKeyUp
keyName
number0
GameController
Class
Fields
id
Methods
~GameControll...
GameController
getButton
getButtonDown
getButtonUp
getId
getLeftTriggerA...
getLeftXAxis
getLeftYAxis
getRightTrigger...
getRightXAxis
getRightYAxis
handleButtonD...
handleButtonUp
releaseButtons
Button
Struct
Fields
buttonName
isButton
isButtonDown
isButtonUp
buttonCode
SDL_GameContr...
Typedef

Xenon Class Diagram




PowerUp
Class
Actor
ContactSensor
Methods
~PowerUp
applyPower
onContact
PowerUp
update
public
public
public
WeaponPowerUp
Class
PowerUp
Methods
~WeaponPowe...
applyPower
onContact
update
WeaponPowerUp
ShieldPowerUp
Class
PowerUp
Fields
bonusHealth
Methods
~ShieldPowerUp
applyPower
onContact
ShieldPowerUp
update
CompanionPow...
Class
PowerUp
Fields
health
maxHealth
Methods
~CompanionPo...
addHealth
applyPower
CompanionPo...
getMissileType
getPosition
onContact
setPosition
setVelocity
takeDamage
update
upgradeMissile
spaceshipRef
companionToAttach
missileType
Missile
Class
Actor
Projectile
Fields
explosionAnim
explosionTex
explosionTilem...
heavyDamage
lightDamage
mediumDamage
missileDamage
Methods
~Missile
Missile
onContact
update
missileType
MissileType
Enum Class
light
medium
heavy
Spaceship
Class
Pawn
Fields
attachedComp...
burnerAnimLeft
burnerAnimRight
burnerTexLeft
burnerTexRight
burnerTilemapL...
burnerTilemap...
canFire
cooldown
cooldownThread
fireSound
health
isAlive
isMovingLeft
isMovingRight
maxHealth
missileHalfSize
missileVelocity
moveLeftAnim
movementKeys
moveRightAnim
needsCooldown
returnLeftAnim
returnRightAnim
Methods
~Spaceship
addHealth
animateMoveL...
animateMoveRi...
animateReturnL...
animateReturn...
attachCompani...
cooldownCheck
fire
getHealth
getPosition
onContact
removeCompa...
Spaceship
stopTurningAni...
takeDamage
update
upgradeMissile
ship
GameManager
Class
Actor
Fields
canRespawn
hiScore
instance
playerLives
score
timeSinceDeath
Methods
~GameManager
addLife
addScore
clearInstance
GameManager
getHealth
getHiScore
getInstance
getLives
getScore
loseLife
spawnPlayer
update
UI
Class
UICanvas
Fields
healthBar
largeFont
livesBar
smallFont
Methods
~UI
render
UI
update

Conclusão

O resultado obtido e objetivos alcançados atingiram as nossas expectativas e foram uma boa ferramenta de aprendizagem.

Aprendemos bastante sobre a linguagem C++, OpenGL e ferramentas como o SDL2, Box2D e OpenAL úteis na criação de um game engine e aprofundamos conhecimento acerca da lógica por detrás do mesmo. Tudo isto acreditamos serem mais valias para o nosso futuro.

Referências

- "ASCII Table and Description." *Ascii Table*, www.asciitable.com/.
- "Beginning Game Programming v2.0." *Lazy Foo' Productions*, lazyfoo.net/tutorials/SDL/index.php.
- Birch, Carl. "How To Make A Game In C++ & SDL2 From Scratch!" *YouTube*, Let's Make Games, 28 May 2017, www.youtube.com/playlist?list=PLhfAbcv9cehhkG7ZQK0nfIGJC_C-wSLrx.
- *Box2D Documentation*, box2d.org/documentation/.
- "Breakout." *LearnOpenGL*, learnopengl.com/In-Practice/2D-Game/Breakout.
- "C And C++ Reference." *Cppreference.com*, en.cppreference.com/w/.
- *Cplusplus.com - The C++ Resources Network*, www.cplusplus.com/.
- "Cross Platform 3D Audio." *OpenAL Documentation*, www.openal.org/documentation/.
- Deckhead. "Complete Guide to OpenAL with C++ Part 1." *IndieGameDev*, 25 Apr. 2020, indiegamedev.net/2020/02/15/the-complete-guide-to-openal-with-c-part-1-playing-a-sound/.
- "Guide on a Random Number Generator C++: The Use of C++ Srand." *BitDegree*, 14 Oct. 2019, www.bitdegree.org/learn/random-number-generator-cpp.
- "Input Manager." *Unity Manual*, docs.unity3d.com/Manual/class-InputManager.html.
- "Introduction - Box2D Tutorials." *iforce2d*, www.iforce2d.net/b2dtut/introduction.
- kcat. "OpenAL Soft." *GitHub*, github.com/kcat/openal-soft.
- "Order of Execution for Event Functions." *Unity Manual*, docs.unity3d.com/Manual/ExecutionOrder.html.
- *SDL Wiki*, wiki.libsdl.org/.
- Stark, Adam. "AudioFile." *GitHub*, github.com/adamstark/AudioFile.
- Stone, Matt. "How to Setup and Use OpenAL for Game Audio in C++ (Using Visual Studio)." *YouTube*, Enigma Tutorials, 29 Aug. 2020, www.youtube.com/watch?v=WvND0djMcfE.
- Stroustrup, Bjarne. *A Tour of C++*. 1st ed., Addison-Wesley, 2013.
- vinnyvicious. "Best Way to Handle Input in SDL?" *GameDev.net*, 5 Jan. 2014, gamedev.net/forums/topic/651896-best-way-to-handle-input-in-sdl/5121262/.

- “Visual Studio Documentation.” *Microsoft Docs*, docs.microsoft.com/en-us/visualstudio/windows/?view=vs-2019.
- “Xenon 2000 Remake by The Bitmap Brothers (Free Game).” *YouTube*, Idealsoft Blog.it, 27 June 2010, www.youtube.com/watch?v=eMWRH_69Xo4.
- Fichas resolvidas nas aulas da unidade curricular.
- Fichas resolvidas nas aulas de Computer Graphics.