UNIVERSITAT DE BARCELONA

Treball Final de Grau

GRAU D'ENGINIERIA INFORMATICA

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

3D Python Engine: DESENVOLUPAR UN MOTOR DE JOC EN PYTHON

Autor: Noel Vázquez Caparrós

Directora: Dr. Anna Puig Puig
Realitzat a: Departament de Matemàtiques i Informàtica

Barcelona, 13 de juny de 2023

Abstract

The video games industry is growing every year, surpassing in 2020 income from cinema and sports industries combined. Many of the mentioned games use tools that are specially designed to develop videogames, called engines, which are already capable of offering common features such as player input, sound control or communication with the GPU to display the game on screen. Notably, Unity and Unreal Engine are some of the better known examples.

In the case of Python, there is a package dedicated to developing games, Pygame, but this has certain shortcomings. In general terms, it is a *wrapper* of SDL (Simple DirectMedia Layer), and this brings with it certain weakness. Fisrt of all, it can only graphically process two-dimensional elements, that is, it only overlaps images with each other. In addition, the vast majority of its tasks only use the CPU instead of the GPU, thus losing efficiency. Finally, to develop games using Pygame is a non easy task for developers who do not usually programme games.Thus, our project aims to cover some of these gaps. First, we present the design and the development of a Python framework that makes the most of the GPU, giving at least the same features as Pygame. Second, we expand Pygame to offer an object architecture ready to develop games easily. Third, regarding the graphics perfomance, we extend the graphics engine to support three-dimensional games. Last but not least, our framework provides an object architecture ready to develop games easily. To sum up, this project presents the analysis and development of the graphic engine capable of easily creating three-dimensional games in Python using the GPU power.

Resum

El món dels videojocs cada any creix més, arribant, al 2020 a superar en ingressos al cinema i els esports junts. Bona part d'aquests jocs usen eines dedicades a desenvolupar jocs, anomenades "engines", o motors, que ja tenen incorporades capacitats mínimes com rebre input de la màquina, emetre sons, o l'ús de la GPU per mostrar elements del joc a pantalla, d'entre altres, els exemples més comuns serien Unreal Engine o Unity.

En el cas de Python hi ha una llibreria dedicada a desenvolupar jocs, Pygame, però aquesta té certes mancances. En termes generals, és un *wrapper* de SDL (Simple DirectMedia Layer), i això comporta certes febleses. En primer lloc, només pot processar gràficament elements bidimensionals, és a dir, tan sols solapa imatges entre sí. A més a més, la gran majoria de les seves tasques les dirigeix a la CPU en comptes de la GPU, perdent així eficiència. En aquest projecte ens plantegem resoldre algunes d'aquestes mancances. El primer objectiu és d'aquest projecte és dissenyar i desenvolupar un framework de python que aprofiti al màxim possible la GPU, donant com a mínim les mateixes prestacions que Pygame. Així mateix, el projecte estén les característiques de Pygame per a oferir una arquitectura d'objectes preparada per a desenvolupar els jocs fàcilment i donant la possibilitat de tenir una interfície visual per a interaccionar amb ells. En relació a les prestacions gràfi-

ques, el motor gràfic suporta jocs tridimensionals. Finalment, es proporcionarà una arquitectura d'objectes preparada per desenvolupar jocs. Així, en aquest projecte es presenta l'anàlisi i desenvolupament del motor gràfic capaç de crear fàcilment jocs tridimensionals en Python. En el millor dels casos, podria arribar a donar com a resultat una engine capaç de crear jocs tridimensionals.

Resumen

El mundo de los videojuegos crece cada año más, llegando, en el 2020 a superar en ingresos al cine y los deportes juntos. Buena parte de estos juegos usan herramientas dedicadas a desarrollar juegos, llamadas "ingenes", o motores, que ya tienen incorporadas capacidades mínimas como recibir input de la máquina, emitir sonidos, o el uso de la GPU para mostrar elementos del juego en pantalla, entre otros, los ejemplos más comunes serían Unreal Engine o Unity.

El mundo de los videojuegos crece cada año más, llegando, en el 2020 a superar en ingresos al cine y los deportes juntos. Buena parte de estos juegos usan herramientas dedicadas a desarrollar juegos, llamadas "ingenes", o motores, que ya tienen incorporadas capacidades mínimas como recibir input de la máquina, emitir sonidos, o el uso de la GPU para mostrar elementos del juego en pantalla, entre otros, los ejemplos más comunes serían Unreal Engine o Unity.

En el caso de Python existe una librería dedicada a desarrollar juegos, Pygame, pero ésta tiene ciertas carencias. En términos generales, es un *wrapper* de SDL (Simple DirectMedia Layer), y esto comporta ciertas debilidades; en primer lugar, sólo puede procesar gráficamente elementos bidimensionales, es decir, tan sólo solapa imágenes entre sí. Además, la gran mayoría de sus tareas las dirige a la CPU en vez de a la GPU, perdiendo así eficiencia. En este proyecto nos planteamos resolver algunas de las debilidades mencionadas. El primer objetivo es diseñar y desarrollar un framework de Python que aproveche lo máximo posible la GPU, dando al menos las mismas prestaciones que Pygame. Asimismo, El proyecto extiende algunas características de Pygame para ofrecer una arquitectura de objetos preparada para desarrollar los juegos fácilmente y dando la posibilidad de tener una interfaz visual para interaccionar con ellos. En relación a las prestaciones gráficas, el motor gráfico soporta juegos tridimensionales.

Índex

Capítol 1

Introducció

L'objectiu d'aquest projecte és usar els recursos disponibles al llenguatge de python per preparar un codi que faciliti el desenvolupament de videojocs tridimensionals que usin les capacitats de les targetes gràfiques actuals d'una manera senzilla per gent amb poc coneixement sobre aquesta vessant.

1.1 Context i Motivació

En aquest projecte, es proposa programar una llibreria (que és com s'anomena al codi preparat per reutilitzar) per desenvolupar videojocs, que utilitzi per mostrar elements a pantalla el renderitzat de polígons 2D i 3D. Els polígons són elements que es formen quan connectem entre si tres o més punts a l'espai, és el que es fa servir als videojocs moderns per representar formes complexes, en contrapart a la limitada capacitat de pygame de mostrar solament imatges rectangulars planes.

L'àlgebra ens permet treballar de manera senzilla amb els cossos poligonals, donat que la manera més simplista de processar les dades dels vèrtexs d'un polígon (els punts que els formen) és fer servir multiplicacions matricials per aplicar translacions, rotacions i escales diferents als valors originals, que farem servir per comunicar-nos amb els gràfics de l'ordinador, i generar les imatges del joc. Aquest concepte és tan pràctic, que fins i tot en els casos on simplement solapar imatges en un pla bidimensional és una opció, s'opta per per fer jocs amb polígons, com podem veure a la figura 1.1.

1.2 Objectiu principal

L'objectiu principal d'aquest projecte és aconseguir crear un motor de joc que funcioni com a alternativa a Pygame, la llibreria més popular per fer jocs a Python, però que utilitzi la GPU per generar gràfics poligonals 2D i 3D, tot mantenint un entorn amigable de programació i adquirint la capacitat de mostrar elements tridimensionals.

Figura 1.1: Un conegut joc (Shovel knight) d'estil pixelat que utilitza polígons en un espai tridimensional de manera constant. Extreta de https://www.youtube.com/watch?v=vjENktnbCaE

1.3 Objectius específics

- Estudi de les característiques dels motors gràfics actuals més significatius que estan oberts al públic: Unreal Engine, Unity, RPG Maker d'entre altres.
- Anàlisi dels requeriments del motor gràfic a desenvolupar: quines característiques es considerarien més importants, quins recursos faríem servir per donar-les...
- Disseny i desenvolupament del motor gràfic: Decisió de la llibreria amb que ens comunicarem amb la GPU, com configurar millor el procés de comunicació...
- Validació de la proposta: comparar els resultats que pot donar el motor amb PyGame.

1.4 Planning

Durant el transcurs del desenvolupament d'aquest projecte, es durà a terme un procés de desenvolupament incremental i iteratiu inspirat en SCRUM, on es definiran diferents sprints setmanals, en els quals es desenvoluparan i dissenyaran diferents parts del codi del motor, amb proves unitàries que aniran creixent a mesura que el projecte avanci, com es pot veure a la figura 1.2.

Figura 1.2: Diagrama de Gantt amb la planificació quinzenal del projecte.

A la figura 1.2 es pot veure el diagrama de Gantt del projecte on es es descriu la planificació temporal de les tasques que s'enumeren a continuació. Cal destacar que per a cada tasca s'han realitzat proves i tests específics per a validar el seu funcionament.

- Antecedents: Recopilació d'informació relacionada amb altres motors i els seus punts forts i febles
- Finestres: Desenvolupament del control de finestres del programa.
- Input: Gestió de la interacció entre el jugador i el programa.
- Tris: Comportament de polígons triangulars.
- Quads: Comportament de polígons quadriculars.
- Malles: Comportament de les malles de polígons.
- Textures: Control sobre les textures del programa.
- Col·lisions: Gestió, creació i control sobre els cossos de col·lisions.
- Matrius de transformació: Aplicació de les apropiades transformacions als cosos del programa.
- So: Inclusió de la prestació del so en els jocs.
- Memòria: Redacció d'aquest document.

1.5 Organització del document

A continuació, hi ha desglossats els diferents apartats d'aquest document, i el seu contingut

- **Introducció:** Es el present capítol on s'introdueixen alguns conceptes bàsics sobre el tema i es defineixen els objectius principals i la seva planificació.
- **Antecedents:** En aquest capítol s'introdueixen les característiques esperades en un motor gràfic i l'estudi dels motors gràfics actuals més relacionats amb el projecte, també es compararan aquests motors amb el que s'espera del resultat del projecte.
- **Anàlisi i Disseny central del motor gràfic**: En aquest capítol es descriuen els requeriments i el disseny del nucli central del joc, les principals classes implicades com és la classe del joc (Game), l'escena, les entitats de joc, etc. i la seva connexió amb la GPU. Així mateix es descriu com es pot definir la interacció amb els diferents elements gràfics i com es controla des del motor.
- **Detecció de Col·lisions:** En aquest capítol es descriu com s'integra en el motor el control de col·lisions tant en 2D com en 3D.
- **Implementació i desenvolupament:** En aquest capítol es detallen aspectes tècnics del desenvolupament del projecte.
- **Resultats i observacions:** En aquest capítol es mostren els resultats obtinguts del projecte desenvolupat i les comparacions realitzades amb el motor PyGame per a validar l'abast del projecte.
- **Conclusions i feina futura:** En aquest últim capítol es detallen les conclusions i les línies obertes per integrar en futures versions del motor.
- **Apèndix:** Està separat en dues parts, la primera és el manual tècnic, on es detallen els requeriments mínims i les instruccions d'us del programa per poder visualitzar les demostracions del projecte. La segona part explica com accedir a la documentació del projecte i navegar-hi.

Capítol 2

Antecedents

En aquest capítol, veurem el que és un motor de joc, els diferents components que el defineixen, les seves avantatges, la raó per què es fan servir a la industria, i el què el diferencia d'un motor gràfic

2.1 Motors de joc i les seves principals característiques

Comunament, s'entén com a "motor de joc"o "engine"qualsevol software no directament dependent d'un altre, dedicat al desenvolupament de videojocs. Definicions una mica més desenvolupades del concepte especifiquen que han de formar "arquitectures de dades reutilitzables sense contingut de joc", d'acord al contingut lliure de "Game Engine book"[1], una característica de la qual manquen alguns elements de software que volen ser descrites com a motors. Un exemple n'és PyGame Engine (ho diu el nom). A la figura 2.1 hi podem observar un exemple en forma d'esquema d'algunes de les capacitats que s'espera que tingui un motor, a més d'una proposta d'estructura funcional, tot i que on més destaca és a la part de motor gràfic. Cal entendre que un motor de joc no és el mateix que un motor gràfic, perquè el primer s'encarrega de més tasques que el segon. Per posar un exemple, tot i que ja hem vist que la figura 2.1 se centra a la part visual del que és un motor, també hi ha definits elements de so.

En resum, un motor de joc es el resultat format pel conjunt de sistemes funcionals que li donaran les diferents capacitats al joc. L'exemple més obvi és el motor gràfic, però a part també tenim el motor de so, el motor d'entrades del jugador, també podem arribar a tenir elements més específics com un motor de físiques o un sistema de comunicació en xarxa, per posar alguns exemples.

Val la pena fer èmfasi en que els components que formen un motor de joc poden ser tan independents entre si, que poden resultar en motors d'utilitat específica reutilitzables en altres motors de joc, com en seria d'exemple el motor de físiques

[1] https://www.gameenginebook.com/coursemat.html

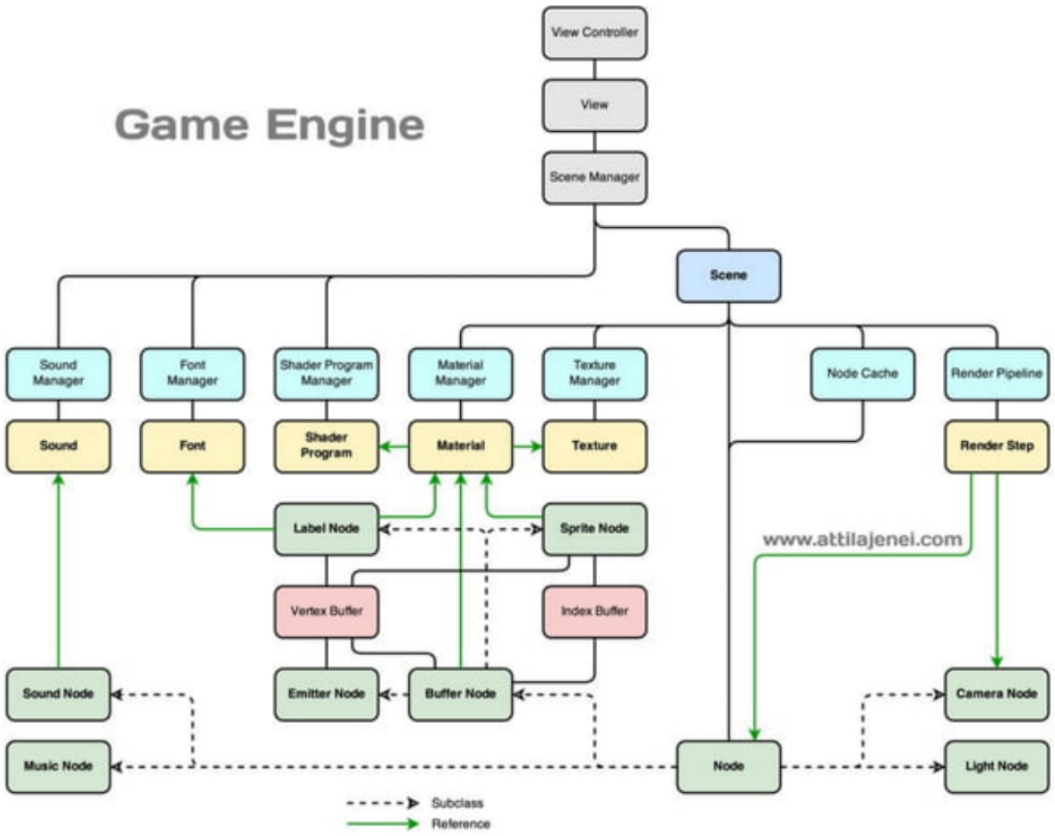


Figura 2.1: Un exemple esquemàtic de l'estructura d'un motor de joc. Extret de https://www.slideshare.net/AttilaJenei/game-engine-architecture-34006558

de havok[2], usat en una immensa varietat de motors i jocs.

2.2 Altres motors de joc actuals

Per començar, és important observar els punts forts de motors similars amb la intenció d'assolir els mateixos objectius o semblants, i de la mateixa manera, entendre les seves mancances intentar-les suplir, compensant les característiques que sigui inviable implementar, per aconseguir realitzar un producte amb valor comparat amb la resta.

A continuació s'estudien diferents motors actuals, que, tot i que no és una llista exhaustiva, s'han escollit els que per les seves característiques són més propers als objectius del projecte.

[2] https://www.havok.com/havok-powered/

2.2.1 Cry Engine[3]

Cry Engine és un motor desenvolupat pels desenvolupadors Crytek. Va néixer amb la saga de Far cry, i n'han anat creant noves versions, llençant la cinquena (Cry Engine V) al públic, per permetre'n l'ús gratuït. El motor està enfocat a jocs visualment complexes de disparar en 3D, això els permet enfocar-se a produir característiques específiques, diferents efectes de llum i ombra, d'entre altres efectes visuals, físiques realistes en temps real, un robust sistema d'àudio, a part de proporcionar un editor específic per desenvolupar mapes que poden arribar a ser enormes. Per altra banda, això fa que sigui complicat desenvolupar qualsevol concepte fora d'aquest concepte, i que el resultat no es pugui gaudir amb tot el seu potencial a ordinadors de característiques modestes, donant efectes visuals pitjors i/o rendiment.

2.2.2 Game Maker[4]

Anteriorment coneguda com a Animo, és una eina de desenvolupament que des de la seva concepció es va enfocar a persones sense gaire experiència amb la programació. És una eina específica per jocs 2D, i avui dia, s'ha arribat a usar tan grans companyies com per desenvolupadors independents, sent aquests últims els més comuns. És la eina que s'ha fet servir per crear jocs populars com Katana Zero o Undertale, que han arribat a tenir un fort impacte a les xarxes socials. Utilitza DirectX, està fet en C++, i es desenvolupa usant un llenguatge propi interpretat, les seves capacitats tècniques però, es limiten als jocs 2D.

2.2.3 Godot[5]

Godot és un motor de capacitats 2D i 3D gratuït i de codi obert multiplataforma, funciona sobre OpenGL, està programat sobre C++, permet programar en el seu propi llenguatge interpretat, a part de donar opció a usar altres eines per programar en C++ o C#. Destaca per incloure les seves pròpies eines d'animació, edició de nivells, i programació de shaders per via d'una eina de nodes.

2.2.4 Pygame[6]

És un mòdul de Python dissenyat pel desenvolupament de videojocs, inclou llibreries de desenvolupament gràfic, so i detecció de col·lisions, fet per ser usat a Python. Es va concebre com a substitut de PySDL[3], com a tal, esta basat en SDL (Simple Directmedia Layer[4]), la qual cosa causa que les seves capacitats es limitin a l'espai 2D, treballant només amb imatges i formes quadrades, i impedeix l'ús de gràfics accelerats per GPU, donat que SDL únicament utiliza la CPU.

[3] https://pypi.org/project/PySDL2/
[4] https://www.libsdl.org/

2.2.5 RenPy[7]

Aquesta és una eina específicament creada per crear jocs del gènere anomenat "novela visual", com a tal, està molt limitat a mostrar solament imatges i textos de diàleg per pantalla, tot i així, la gent no deixa de nombrar-la com a motor de joc, com que està desenvolupada en Python, és menys complicat que en altres motors crear alguna cosa fora de l'esquema normal de desenvolupament, tot i que està fet en PyGame, arrastrant també les seves limitacions. Tots aquests factors no són impediment per tenir resultats destacables, com és el cas de Doki Doki Literature Club [5], un joc desenvolupat per un petit grup que ha arribat a tenir tant èxit com per aconseguir un llançament físic.

2.2.6 RPG Maker[8]

RPG Maker té múltiples versions, basant-se les més noves en Javascript, i donant un entorn específic per desenvolupar jocs 2D de rol per torns, amb un editor de mapes, events, i de les dades del programa. Elements pels que el motor no estigui específicament preparat, es torna una feina dura en el millor dels casos.

2.2.7 Unity[9]

El motor gràfic Unity està desenvolupat en C# i des del principi va estar pensat per l'ús de petits desenvolupadors, està dissenyat per usar OpenGL[6], i per windows, Direct X[7]. És un motor de propòsit general, la qual cosa fa que tot el que sigui desenvolupar característiques noves pel joc, depèn molt de l'habilitat del propi programador a l'hora de dur-la a terme. La seva arquitectura es basa en "Game Objects", amb un disseny basat en components que fan que sigui molt modulable i flexible a l'hora d'incloure noves característiques als objectes.. Té sistema d'escenes en que cadascuna d'aquestes té un esquema d'objectes, cadascun dels quals té el seu propi grup de scripts (els components) que funcionen independentment (o no) dels altres També s'implementa d'aquesta manera els elements de so i de renderitzat, el que facilita al desenvolupador el flux de feina, gràcies a un format molt estable.

2.2.8 Unreal Engine[10]

El motor de joc Unreal Engine està desenvolupat en C++, i utilitza DirectX. Està orientat principalment a jocs d'acció en primera persona, però s'ha demostrat en múltiples ocasions què és prou flexible com perquè desenvolupadors experimentats aconsegueixin qualsevol resultat, i ha evolucionat per acabar donant suport a un desenvolupament genèric. Tot i així, és una eina molt poc amigable amb gent sense experiència en aquest entorn específic, a més de tenir niuades moltes eines diferents

[5] https://steamcommunity.com/app/698780/discussions/0/1483233503859849344/
[6] https://www.opengl.org/
[7] https://www.windowscentral.com/what-directx-why-does-matter-gaming

i dependents entre si pel desenvolupament. A més a més, tot i que dona grans prestacions gràfiques, és un programa que consumeix molts recursos de la màquina, la qual cosa resulta en una pitjor experiència de cara als consumidors amb uns recursos més modestos que un ordinador d'alta gamma.

2.3 Comparativa resum

Prenent de referència les dades obtingudes sobre els software mencionats les seccions anteriors, podem determinar fins a cert punt les capacitats que aporten a un programador a l'hora de desenvolupar un videojoc. Hi ha molts elements a l'hora de tenir en conte per determinar la potència d'un motor, tot i així, és irrealista intentar assolir els mateixos resultats que aquests exemples en tot. Per això s'han escollit he escollit les diferents característiques de la figura 2.2 poder analitzar més fàcilment els motors, comparar-los, i determinar un objectiu assequible.

Les diferents columnes de la taula 2.2 representen:

- **2D:** Suport nadiu per gràfics bidimensionals.
- **3D:** Suport nadiu per gràfics tridimensionals.
- **Tipus**: Si està preparat per fer algun gènere de joc concret.
- **Col·lisions:** Capacitat per determinar si dos elements comparteixen espai.
- **Físiques:** Generació de moviment a partir de gravetat i/o col·lisions.
- **Llums:** Suport a llums. les llums alteren de manera nativa les visuals del joc.
- **Llenguatge:** els diferents llenguatges en què es pot programar.
- **GPU:** Ús o no de la GPU.

Motor	2D	3D	Tipus	Col·lisions	Físiques	Llums	Llenguatge	GPU
Cry Engine	✗	✓	Acció	✓	✓	✓	C++,Lua,C#	✓
Game Maker	✓	✗	Genèric	✓	✓	✗	Propi	✓
Godot	✓	✓	Genèric	✓	✓	✓	Propi	✓
Pygame	✓	✗	Genèric	✓	✗	✗	Python	✗
Ren'Py	✓	✗	VN[8]	✗	✗	✗	Python	✗
RPG Maker	✓	✗	RPG	✓	✗	✗	Javascript	✓
Unity	✓	✓	Genèric	✓	✓	✓	C#	✓
Unreal	✓	✓	Genèric	✓	✓	✓	BP, C++	✓

Figura 2.2: Comparació de les característiques principals dels motors gràfics estudiats.

En resum, analitzant els exemples de motors gràfics més coneguts disponibles al públic, es pot concloure que la majoria tenen suport pel desenvolupament 2D, però

Motor	2D	3D	Tipus	Col·lisions	Físiques	Llums	Llenguatge	GPU
3D Python Engine	✓	✓	Genèric	✓	✗	✗	Python	✓

Figura 2.3: Característiques del motor gràfic a desenvolupar

només la meitat d'aquests exemples té suport per jocs 3D o il·luminació , la majoria donen suport a físiques (5 dels vuit), però pràcticament tots els motors esmentats gestionen col·lisions.

A partir d'aquesta anàlisi comparativa entre els motors gràfics mentats, ens plantegen cobrir algunes mancances amb el projecte actual. De fet, ens proposem desenvolupar un motor gràfic amb suport per gràfics 2D/3D, detecció de col·lisions, usable des de Python i que aprofiti els recursos de la targeta gràfica. A la figura 2.3 es mostren les característiques que es pretenen cobrir.

Capítol 3

Anàlisi i Disseny central del motor

En aquest capítol veurem una proposta per l'arquitectura general del motor, la que s'ha usat per desenvolupar aquest projecte. Molts motors de joc tenen estructures similars, majorment heretades de l'època en que tots els jocs eren conjunts de diferents nivells. Podem veure això a motors com Unreal Engine, on tot succeeix a un "nivell"o un "mapa". Els noms canviaran entre motors, però no costa gaire trobar equivalències.

Havent observat les diferents arquitectures de motors de joc, la usada aquí està majorment inspirada en la de Unity, donat que permet bastanta versatilitat i simplifica molt on ha d'escriure codi el desenvolupador.

3.1 Arquitectura general d'ArcaDev

Cal entendre què és el que es vol proporcionar a la persona que desenvolupa, i decidir a quin element de l'estructura li delegarem la feina de dur a terme aquestes tasques.

Pensant en el què se li vol oferir al desenvolupador, volem que pugui mostrar elements per pantalla, i, a banda d'això, li volem permetre tenir ple control sobre la finestra del programa, els elements que hi ha a l'execució del programa, i el seu comportament.

Tenint això clar, podem dedicar un objecte a cada feina(veure figura 3.1); hi ha una classe Game que gestiona l'execució del programa, i la interacció amb l'usuari, també conté els altres elements amb tasques més rellevants que poguessin fer de manera independent. Aquestes són la comunicació amb la GPU, i un objecte de escena que seria on el desenvolupador pot prendre control del que passa. Dins una escena, hi ha dos tipus d'elements diferents, un és l'entitat, que contindrà tot el que representa cada objecte i l'altre tipus d'elements, que són els Components, on realment el desenvolupador pot programar ell mateix el comportament específic per les entitats. Separar els objectes del comportament, permet programar un component que usin múltiples objectes diferents sense haver de repetir codi. Un exemple d'això és l'obtenció d'informació dels elements que es mostrin en pantalla:

no tots els objectes es mostraran en pantalla, i no tots els elements que ho facin tindran el mateix comportament.

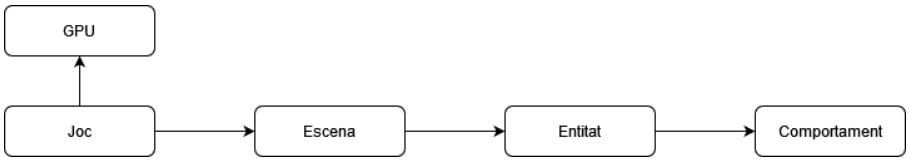


Figura 3.1: Arquitectura general

3.2 Disseny del Nucli del joc

A la figura 3.2 es mostra una primera representació en diagrama de les classes que han resultat de la primera iteració del disseny. Tot i que en el futur poden arribar a canviar certs elements, l'esquema resultant hauria de mantenir-se similar aquest. A més a més, és molt similar al sistema de GameObject i Monobehaviour que utilitza Unity, o sigui que si algú familiaritzat amb aquest motor intenta començar a fer servir aquest altre nou entorn, la corba d'aprenentatge serà suau.

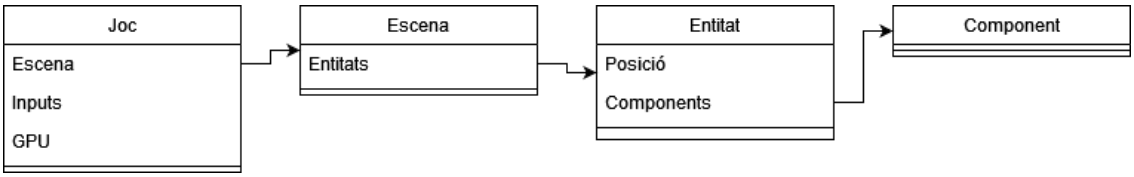


Figura 3.2: Diagrama de classes de la proposta inicial

Joc

És una classe que té tres atributs privats crucials pel cicle de joc: **Escene**, que serà l'escena activa pel joc, **inputs**, que és una llista de tuples que indiquen els canvis d'estats del teclat, **gpu**, que conté la instancia de la API per comunicar-se amb la gpu.

Llavors, de cara al comportament, també li hem de donar capacitat per, enregistrar entrades de teclat mencionades, i per actualitzar cada frame el joc.

Escena

Aquesta classe està dedicada a generar els diferents entorns en els què s'executarà cada joc. Té un llistat d'entitats, que és el llistat d'entitats a processar i renderitzar.

Els seus mètodes li han de donar capacitat per actualitzar l'estat de les seves entitats, per afegir-ne, i per enviar les dades d'aquestes a la GPU, per que així puguin ser renderitzades a pantalla.

Entitat

La entitat que ocuparà les escenes, té un llistat de components i una posició. Té un mètode per afegir components a la entitat, i un altre per actualitzar l'estat de les components que tingui.

Component

Aquesta és una classe que en altres llenguatges ben bé podria haver estat definida com a abstracta, donat que per si mateixa no executa cap funció. Està dissenyada per crear-ne classes filles per definir diferents comportaments de cara a les entitats que les usaran, no té atributs, només un mètode mencionat anteriorment a actualitzar el seu propi estat, candidat a ser sobrecarregat per les classes filles.

Renderable

Aquesta és una classe filla de Component que està dissenyada per contenir la informació per renderitzar les entitats a pantalla. Ara per ara, només conté la informació dels vèrtexs i una propietat per poder llegir la seva informació i enviar-la a la GPU.

3.3 Control sobre les entrades

Aquesta tasca l'ha de dur a terme la llibreria que també gestiona el control de la finestra del programa, donat que és la que estarà més preparada per fer-ho, això vol dir que detectar les entrades dins aquest disseny ho ha de fer la classe "Game". El control però sobre la reacció a aquestes entrades no és aquí, la resposta s'implementara a la classe de la escena, i la intenció és que per cada tipus d'escena diferent s'implementi una classe filla diferent.

3.4 Connexió amb la GPU

Molts exemples online deleguen als diferents elements de l'escena la tasca de comandar a la GPU ser renderitzats. Aquí, però, podem separar els objectes del joc de la comanda de renderitzat, donat que així podem aconseguir un codi més solid, robust, i extensible, donat que si en el futur es necessita canviar l'API de comunicació amb la GPU, serà molt més senzill.

Aplicar aquest concepte a la nostra arquitectura consistirà en, a cada frame generat des de Game, iterar sobre tots els elements renderitzables de l'escena activa, i enviar per cadascun, la seva informació a la GPU, i la ordre de dibuixar l'element.

Capítol 4

Detecció de col·lisions

[1]Si volem que aquest motor pugui ser d'us general, cal tenir un mínim control sobre la detecció de col·lisions. No és implícitament necessaria per absolutament tots els jocs. Per exemple, en el cas dels jocs de simulació, jocs per caselles, on es poden muntar múltiples implenetacions, com un array bidimencional, un diccionari, o comprovar objecte per objecte a quina casella correspon la seva posició, tot i que aquest últim no és gaire eficient.

Donat que volem aconseguir que el codi del motor tingui tanta escalabilitat com sigui possible, el millor és començar per donar un punt de localització a l'espai en tres dimensions per cada element a l'escena. Un cop es tenen aquestes dades, es poden començar a implementar tan la visualització d'elements per pantalla, com la detecció de col·lisions d'elements a l'espai.

Cal tenir clar les dades que es necessiten per identificar cada element i calcular les col·lisions amb aquests, a més a més, és necessari entendre les dades que volem aconseguir d'una col·lisió, i que hi hauran diferents tipus d'elements amb que voldrem detectar una col·lisió, i també s'ha de tenir alguna manera uniforma de descomposar els elements a un mateix tipus d'objecte per no haver d'implementar per cada nou objecte la detecció de col·lisions amb els altres tipus.

Observant aquestes necessitats, la manera més senzilla per contenir les dades necessàries per cada element són: l'element uniforme mentat al paràgraf anterior, la informació del cos amb que es vol detectar la col·lisió, i el mètode particular que cada element haurà de fer servir per detectar la col·lisió.

Potser el més important per saber les dades que podem extreure de les col·lisions és saber què és el que farem servir per detectar-la. El més viable per fer-ho és un dels elements més usat als gràfics generats per ordinador, donat que ens permet extreure molta informació de la seva col·lisió amb qualsevol cos tridimensional primitiu, el raig.

[1]Informació dels calculs dels rajos extreta de la font[11]

4.1 El raig

Prenent de referència la fórmula paramètrica de la recta $O + tD$, un raig està caracteritzat per tres elements, punt d'origen, direcció i distància és a dir, dos vectors tridimensionals i un valor decimal. De manera similar, un raig col·lisionant amb un objecte pot retornar tres elements informatius, la distància des de l'origen, la normal i el punt d'impacte (raó per la qual anomenarem a aquest conjunt de dades impacte), que segueixen sent dos vectors i un decimal.

Per entendre les avantatges d'usar els raigs sobre altres alternatives, podem veure altres exemples, com el de la col·lisió entre dues esferes, on considerem que estan col·lisionant quan la equació de la figura 4.1 dona un resultat verdader.

$$f(x) = \begin{cases} 1 & if \quad 0 \leq (r1 + r2) - distance(c1, c2) \\ \\ 0 & if \quad 0 > (r1 + r2) - distance(c1, c2) \end{cases}$$

On prentem les r com els radis de les esferes, i les c com els centres.

Aquesta alternativa no ens dona tanta informació com la dels raigs, i només la podem fer servir si ambdos cossos son esferes. Tot i així, també és molt menys costosa tant en recursos com en complexitat, raó per la qual es pot arribar a fer servir de manera complementaria.

Entenent aquestes capacitats, ara caldrà implementar altres cossos primitius per aprofitar-les, es poden fer servir esferes, plans, polígons tridimensionals, i polígons 2D

4.2 Esferes

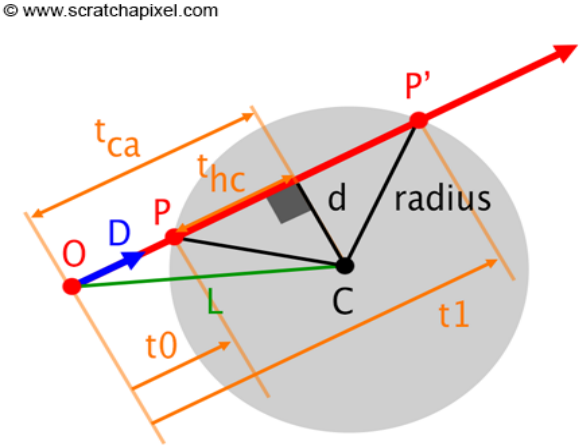
Les dades que identifiquen una esfera són el centre i el radi. Entenent això, cal tenir en conta un concepte molt important al moment de trobar l'impacte entre un raig i una esfera (quan hi ha), i és que un raig no deixa de ser una linia, i quan una en creua una esfera, la creua dues vegades.

Entenent això, el nostre raig generarà dos punts d'impacte diferents, continguts a la mateixa recta, aquesta recta forma un triangle rectangle on (en referència a la figura anterior) la hipotenusa és L, on podem obtenir la distància t_{ca} projectant D sobre L que ens permetrà obtenir el centre del triangle que conté els punts d'impacte (P i P').

Arribats a aquest punt, podem fer la primera comprovació per saber si el raig no creua l'esfera, si $t_{ca} < 0$, significarà que l'esfera esta per darrere de l'origen.

Ara s'ha de construir un segon triangle rectangle com es veu a la figura 4.1, format per d, t_{hc} i el radi de l'esfera, primerament calcularem d, que la podem aconseguir amb $d = \sqrt{L^2 - t_{ca}^2}$ on podrém determinar que no col·lisiona en cas que $d < 0$ o $d < radius$. Tenint totes aquestes dades, ja es pot calcular t_{hc} usant pitàgores, i a partir d'aqui, les distancies del punt origen amb els punts d'impacte

Figura 4.1: Raig interseccionant una esfera.
Extreta de https://scratchapixel.com/



a l'esfera.

$$t_{hc} = \sqrt{radius^2 - d^2}$$
$$t_0 = t_{ca} - t_{hc}$$
$$t_1 = t_{ca} + t_{hc}$$

Tenint aquestes distàncies, ara s'ha de determinar quin dels dos valors és la distància del punt de col·lisió. Podem considerar que la t correcta és la més petita, però també hem de descartar les distàncies menors a 0, donat que això vol dir que estan per darrere del punt d'origen del raig, a més a més, ambdos valors poden estar per darrere del punt origen, per tan, considerarem que no hi ha col·lisió.

$$t(x) = \begin{cases} t_1 \; if \; t_0 < t_1 \; or \; t_1 < 0 \\ t_0 \; if \; t_1 < t_0 \; or \; t_0 < 0 \end{cases}$$

4.3 Plans

De cara als següents elements, es pendran com a base els plans. Els plans estan representats per dues dades, una coordenada d'origen i un vector normal (com mostra la figura 4.2), és a dir, dos vectors tridimensionals.

Donat que D simbolitza la distància entre el pla, i el punt origen (0,0,0), per evitar confusions entre les formules del pla, i la fórmula del raig, tractarem la fórmula del raig com a $P = O + tR$.

Per començar, cal definir que el producte escalar entre dos vectors perpendiculars sempre és 0. Llavors, donats un vector A i B perpendiculars entre si:

$$A \cdot B = 0$$

Ara, sabent això, gràcies al vector generat entre les coordenades d'origen i un punt de referencia del pla, podem escriure l'equació del pla de la següent manera:

$$(P - P_0) \cdot n = 0$$

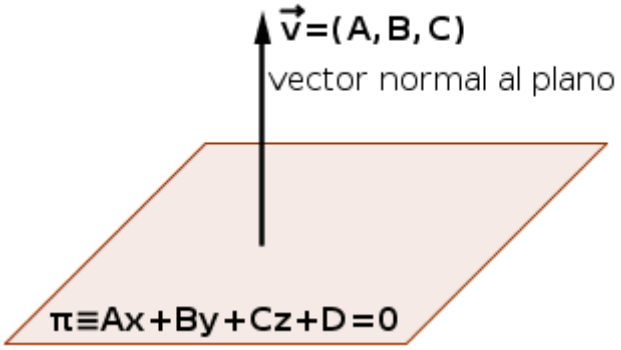


Figura 4.2: Representació del pla amb el vector normal.
Extreta de https://matematicasies.com/05-Vector-normal-a-un-plano

Ara, podent calcular la posició de qualsevol punt al llarg del raig des de l'origen del raig, la seva direcció i el terme, que és un nombre real positiu (que, com és habitual, és la distància paramètrica des de l'origen del raig fins al punt d'interès), si el raig i el pla es tallen, comparteixen un punt on la recta talla el pla. Si aquest punt és la P del raig, es pot combinar l'equació anterior amb la del raig de la següent manera:

$$(O + tR - P_0) \cdot n = 0$$

Ara, com que el valor que interessa trobar és el de t, resolent l'equació s'acaba obtenint el següent càlcul:

$$t = \frac{(P_0 - O) \cdot n}{R \cdot n}$$

Coneixent el valor de t, si és positiu, indicarà impacte. Es pot usar per trobar el punt de col·lisió amb l'equació del raig, i la el valor del vector normal de l'impacte que és el mateix que el vector normal del pla. Val la pena destacar, que quan el pla i el raig són paral·lels, el resultat de $R \cdot n$ serà molt proper a 0, situació on hi ha infinites sol·lucions, i com que només podem acceptar-ne una de vàlida, al codi s'assigna un marge de tolerància, segons el qual es determina que si el resultat de $R \cdot n$ és massa petit, s'assumeix que R és paral·lel a n, retornant que no hi ha col·lisió.

$$-tolerancia < R \cdot n < tolerancia \rightarrow parallel$$

4.4 Triangles i quadrilàters

Els quadrilaters són una forma que serà molt útil de cara al joc, però més encara ho seran els triangles, també anomenats "tris". Donat que són la forma poligonal més simple, serviran de fonament per construir-ne de més complexes. Començarem per parlar del triangle.

De cara a la computació, el primer pas per a calcular l'impacte d'un raig a un triangle és el mateix, sols que s'ha de comprovar que el punt sigui dins d'una àrea específica formada pel triangle.

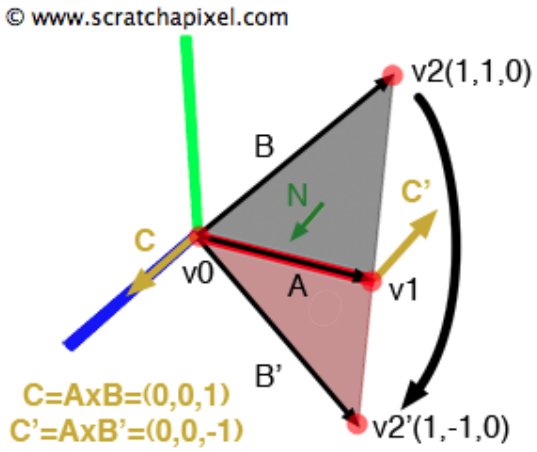


Figura 4.3: Raig interseccionant un triangle.
Extreta de https://scratchapixel.com/

Aquest problema s'ha de resoldre seguint la solució que en anglés es coneix com ïnside-outside test", el proces és el següent, s'ha de tenir en conte els catets del triangle en forma de vector, això ho aconseguim operant amb la posició dels vertex com a l'exemple de la figura 4.3, obtenim el vector A a partir de $A = V1 - V0$, i B a partir de $B = V2 - V0$. A partir d'aquestes dades, podem calcular el producte vectorial entre A i B, que donarà com a resultat un vector C en la mateixa direcció a la normal del pla.

$$
\begin{aligned}
&A = V1 - V0 = (1, 0, 0)\\
&B = V2 - V0 = (1, 1, 0)\\
&C_x = A_y B_z - A_z B_Y = 0\\
&C_y = A_z B_x - A_x B_z = 0\\
&C_z = A_x B_y - A_y B_x = 1 * 1 - 0 * 1 = 1\\
&C = (0, 0, -1)
\end{aligned}
$$

Ara bé, per entendre el valor d'aquesta informació, posem un cas oposat. Tractarem les mateixes dades, però en comptes de $V2$ usarem la $V2'$ que es veu a la figura 4.3, al costat oposat del vector A, donarà com a resultat una C' amb el sentit oposat a C

$$
\begin{aligned}
&A = V1 - V0 = (1, 0, 0)\\
&B = V2 - V0 = (1, -1, 0)\\
&C_x = A_y B_z - A_z B_Y = 0\\
&C_y = A_z B_x - A_x B_z = 0\\
&C_z = A_x B_y - A_y B_x = 1 * -1 - 0 * 1 = -1\\
&C = (0, 0, -1)
\end{aligned}
$$

Ara cal entendre aquestes dades d'una manera més simple. Com que hem d'identificar numèricament si el sentit de C és el mateix o l'oposat al vector normal

del triangle, només cal calcular el producte escalar.

Tenim tres resultats diferents a interpretar. El primer, si el resultat és positiu, el punt és al costat esquerra de A (el que ens interesa, i identificarem com "dins"), si el resultat és negatiu però, vol dir que el punt és a la dreta de A (fora de l'area d'interés).

El resultat de producte escalar 0 és un cas especial, donat que té dues interpretacions diferents, com que el significat és que està alineat amb A és igual de valid entendre que és dins com fora de l'àrea d'interès (Al codi d'aquest projecte s'ha interpretat que és dins l'àrea d'interès).

Com que aquests càlculs ens permeten entendre si un punt és a un costat o altre d'un vector a un pla, la idea és comprovar a quina banda de cada catet és el punt d'impacte, i només serà un impacte vàlid si a tots els casos el punt és a l'esquerra del catet, llavors, podem fer servir aquesta solució tan pel cas del triangle, com pel cas del quadrilàter.

Pel que fa a la representació d'aquests elements en forma de dades, podem precalcular els vectors que representen els catets, i aplicar posteriorment les transformacions que siguin necessàries, per tan, un triangle pot estar representat pels vèrtex A, B, C i els vectors AB, BC, CA, en canvi, el quadrilàter estaria composat pels vèrtex A, B, C, D i els vectors AB, BC, CD, DA.

4.5 Les malles

En el context dels gràfics en 3D, una malla (també coneguda com a malla poligonal o malla de polígons) es refereix a la representació estructural d'un objecte tridimensional mitjançant una col·lecció de polígons interconnectats. Aquests polígons generalment són triangles, quadrilàters o, en alguns casos, polígons amb un major nombre de costats, pel correcte funcionament del nostre programa però, es convertiran tots els polígons a triangles.

La malla està composta per vèrtexs, arestes i cares. Els vèrtexs són punts a l'espai tridimensional que defineixen la posició dels punts d'intersecció dels polígons. Les arestes són els segments que connecten els vèrtexs i defineixen les vores dels polígons. Les cares són els mateixos polígons i representen les superfícies visibles de l'objecte. L'ordre en el què es llisten els vèrtex defineix la cara visible del polígon (la diferenciació entre cara visible i no visible s'anomena Face culling). En el cas concret de OpenGL, la configuració per defecte fa servir la orientació inversa a les agulles del rellotge figura 4.4, però també es pot configurar per fer servir l'ordre oposat, fins i tot per dibuixar independentment de la orientació de la cara, altres llibreries de comunicació amb la GPU actuen de la mateixa manera.

La malla permet descriure la forma i la topologia d'un objecte 3D de manera discreta i finita. Cada polígon de la malla té un conjunt de vèrtexs que especifiquen la seva forma i posició a l'espai tridimensional. En unir els vèrtexs dels polígons adjacents, es forma una xarxa que representa la superfície de l'objecte.

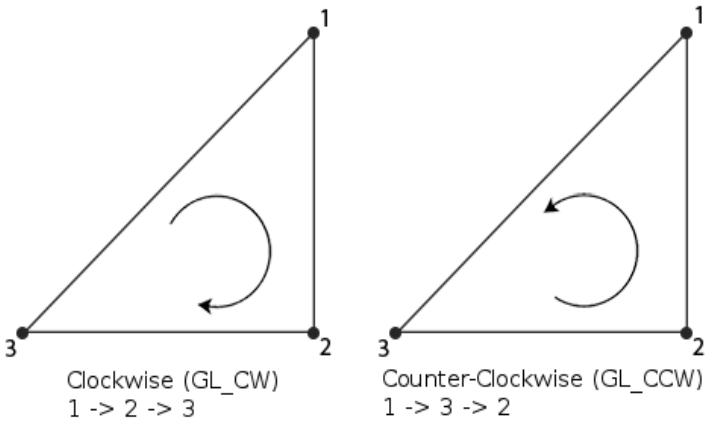


Figura 4.4: Les dues configuracions pel Face culling.
Extreta de https://learnopengl.com/Advanced-OpenGL/Face-culling

La elecció de l'estructura de la malla depèn de diversos factors, com ara l'aplicació, la complexitat de l'objecte i els requisits de rendiment. Els triangles són el tipus de polígon més comúment utilitzat a les malles, a causa de les propietats matemàtiques favorables que posseeixen i la seva simplicitat, que permeten construir les altres formes més complexes.

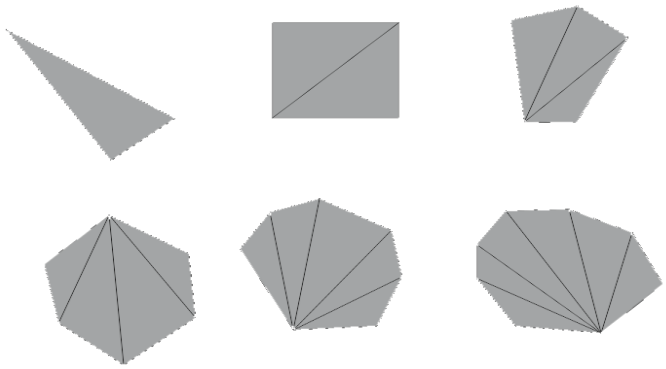


Figura 4.5: Es poden formar tots els poligons usant triangles

Un cop s'ha definit la malla, es pot aplicar informació addicional als vèrtexs i les cares per aconseguir un major nivell de detall visual. Aquestes dades addicionals poden incloure coordenades de textura, que assignen una imatge a cada polígon, permetent l'aplicació de textures i materials realistes. També es poden especificar normals, que són vectors perpendiculars a les cares i que determinen la direcció de la llum i els efectes d'ombres, la majoria d'aquestes dades són calculades pels programes de disseny 3D, i guardades als arxius amb la resta de dades.

Les malles tenen un us molt ampli als videojocs, especialment en 3D, donat que proporcionen una representació eficient i flexible d'objectes tridimensionals, sense un

gran compromís de temps o recursos (la complexitat també depèn de la quantitat de polígons que conté una malla), permetent la seva manipulació, deformació i visualització en temps real. A més, les malles són compatibles amb tècniques avançades de renderitzat, com ara l'ombreament suau, les ombres i els efectes d'il·luminació, que contribueixen a crear imatges i escenes més realistes, en el context de les col·lisions, podem prendre l'avantatge de poder deformar-les en temps real per si hi ha cap cas en que es necessiti un cos que varïi la seva mida de manera constant.

En resum, una malla en el context dels gràfics en 3D és una representació estructural d'un objecte tridimensional mitjançant una col·lecció de polígons interconnectats (que seran convertits a triangles).

4.6 Cossos de col·lisió 2D

Els jocs 2D són massivament més simples a l'hora de desenvolupar, haver de prestar atenció a una dimenció menys lliura de moltíssima feina i acomoda el desenvolupament, i més tenint en conte que al nostre projecte no tenim previst implementar cap mena d'interfície visual per donar suport al desenvolupament. És complicat posicionar correctament elements a un espai 3D.

Ara bé, en previsió de que el més important serà donar facilitat per desenvolupar jocs bidimensionals, s'haurà d'implementar sistemes que facilitin el desenvolupament d'aquest tipus de joc, però encara no en tenim, a arrel de que una de les característiques principals dels cossos plans que s'han implementat fins ara és que ignoren col·lisió amb els cossos coplanars.

Aqui hi han dues opcions, implementar un sistema de colisions específic pel 2D, o bé implementar cossos dins el sistema que ja tenim per acceptar col·lisions coplanars, que és el que s'ha fet.

La solució que s'ha implementat és d'allò més simple, s'ha pres de base els triangles i els quadrilàters, tot i que ignoren les col·lisions coplanars. Dit això, ho podem aprofitar per comprovar si s'ha de buscar cap mena d'impacte 2D, és a dir, es pot fer servir com a guia per determinar si els rajos són al mateix pla que el cos.

Ara el problema és a detectar les col·lisions coplanars, cosa que podem aconseguir reutilitzant codi dels mateixos objectes que prenem de base. Primer, cada catet del cos tindrà associat un pla, després, al igual que el triangle o el quadrilàter, hi haurà catets a partir dels quals comprovarem la posició relativa de l'impacte, però no estaran conec-tats entre si, simplement seran línies perpendiculars al pla 2D, que sortiran de cadascun dels vèrtex de la figura que ens interessa. Llavors, ja es pot usar la mateixa operació que en els altres cossos amb catets, determinant si el punt d'impacte cau a l'àrea d'interès.

4.7 Classes implicades

Ara s'haurà de donar utilitat a aquest desenvolupament al motor, per aconseguir-ho, implementarem un component, que podem anomenar **ColiderComponent**, i aquest component contindrà les següents dades: el mètode per detectar col·lisió amb l'element, les dades que representen l'element, i els rajos que representen l'element.

Fet això, per donar utilitat a aquest component, s'ha de delegar a algun objecte la tasca de detectar les col·lisions entre tots els cossos. L'element que ho fa en el cas del motor d'aquest projecte és la escena, donat que és la que executa les updates a tots els objectes cada frame, i té l'autoritat directa sobre les entitats. Ara, hi ha múltiples opcions, però el més senzill és comprovar un per un cada objecte si col·lisiona amb cap altre, una manera de fer això de manera prou ràpida és, per cada entitat amb col·lisió, comprovar si col·lisiona amb algun dels elements que encara no s'ha comprovat, i, en cas que se'n detecti, enllistar als dos components les dades de la col·lisió. Així, ambdues entitats tindran enregistrada la col·lisió sense necessitar comprovar-la dues vegades. Ara bé, hi ha diverses maneres d'implementar això, però en aquest motor, la classe "Game" passa l'objecte de GPU a la escena, i l'escena s'encarregarà de passar a la GPU les dades de totes les entitats que conté.

Capítol 5

Implementació i Desenvolupament

El desenvolupament d'aquest projecte s'ha estructurat en diferents cicles o iteracions. La primera iteració ha consistit principalment en la implementació de l'arquitectura dissenyada, i la segona en el desenvolupament d'exemple de demostració de les capacitats del motor, llevat d'algunes petites millores.

5.1 Decisions de desenvolupament i implementació

De cara al processament visual, és a dir, la comunicació amb la targeta gràfica, les opcions més viables disponibles a Python eren OpenGL i Vulkan. He decidit usar OpenGL, tot i que no és la API més eficient. De cara a escollir una api més moderna, com seria Vulkan[1], la API destinada a succeir a OpenGL, i també amb suport per més dispositius que Direct X, tot i tenir-ne similituds. Algunes de les característiques per les que destaca Vulkan són una menor càrrega sobre la CPU, una millor gestió dels processadors multi-nucli i un processament dels shaders[2], però a la figura 5.1 podem contemplar una taula comparativa de les diferències tècniques més importants entre OpenGL i Vulkan.

Entenent tot això, podem veure que Vulkan realment és més potent i eficient, però a la vegada és més complexe, és a dir, més difícil de desenvolupar. A banda

[1] https://www.vulkan.org/

[2] Els shaders son el codi que indica com es mostren els elements per pantalla

OpenGL	Vulkan
Una sola màquina d'estat global	Basat en objectes sense estat global
L'estat està lligat a un context únic	Tots els conceptes d'estat estan localitzats en un buffer d'ordres
Les operacions només es poden executar seqüencialment	La programació amb múltiples subprocessos és possible
La memòria i la sincronització de la GPU solen estar ocultes	Control explícit sobre l'administració i la sincronització de la memòria
Àmplia comprovació d'errors	Els controladors de Vulkan no verifiquen errors en el temps d'execució, hi ha una capa de validació per als desenvolupadors

Figura 5.1: Majors diferències entre OpenGL i Vulkan

d'això, la llibreria de Python està desactualitzada, la última actualització[3] va ser el 2019.

5.2 Primera iteració: Construint una finestra interactiva

Aquesta secció descriu el primer cicle de desenvolupament del motor, on l'objectiu principal era arribar a poder obrir una finestra al sistema, on s'executaria el programa del joc, i hi apareixeria alguna mena d'element, que pogués alterar la seva posició relativa a la càmara, amb l'ús d'inputs per part del jugador.

5.2.1 Recursos

En aquesta primera versió, les principals llibreries a usar seran glfw(2.5.6)[4] per la creació i control sobre la finestra del programa, així com també la recepció de inputs per part del jugador. Pel que fa a la part de renderitzat a la pantalla, ara per ara es treballarà amb PyOpenGL(3.1.6)[5], i les implementacions millorades de PyOpenGL-accelerate(3.1.6)[6].

5.2.2 Disseny de classes

A continuació, una explicació de les dades més importants de la implementació de cada classe, els seus mètodes, i els atributs, i els atributs privats:

- **Game:** Amb els atributs **escena** i **gpu**, té un mètode **set_scene** per canviar de manera apropiada la escena a l'execució. També té el mètode **key_callback** per detectar entrades per consola, i una propietat per poder accedir a l'escena des de fora de Game. El mètode **run** és el que s'encarrega de la gestió dels inputs i actualitzacions de pantalla, sent el mètode que s'usa per executar el joc
- **GameScene:** Conté llistes privades per **entitats**, **renderitzables** i els **col·lisionables**. També conté la implementació del mètode **update** que s'encarrega de cridar el mètode homònim de les entitats dins l'escena, **detect_colisions** per comunicar a cada entitat les seves col·lisions, **spawn_entity** per inserir entitats a l'escena, **render** per dibuixar els elements de la escena, **set_camera** per determinar quina entitat es prendrà com a referència per renderitzar l'escena des de la seva perspectiva, i el mètode **input** que determina a quins inputs respondrà el joc.

[3] https://pypi.org/project/vulkan/#history
[4] https://www.glfw.org/
[5] https://pypi.org/project/PyOpenGL/
[6] https://pypi.org/project/PyOpenGL-accelerate/

- **GameEntity**: Tindrà múltiples atributs privats, com **nom**, **posició**, **rotació** i un diccionari de **components**. A banda d'aquests, també contindrà referències a components freqüentment utilitzats com **shader**, **render**, **colider** que, en cas de no ser usats, contindran un valor nul (que al codi Python figura com a None). Pel que fa als mètodes, n'hi haurà per accedir a la posició, la rotació, **add_component** per afegir components de manera correcta i el mètode **update** que és el que crida als mètodes amb el mateix nom dins els components.

- **EntityComponent:** És una classe que no té atributs, està feta per fer-ne classes filles. té un mètode **update** buit que és el que el desenvolupador principalment alterarà. També conté unes propietats per determinar si s'ha de cridar en renderitzat, detecció de col·lisions, o si s'ha de cridar en update. Això està fet així per afegir a les llistes adients només els elements apropiats, per exemple, si el component no conté comportament, i només conté dades, no cal afegirlo a la llista de components a cridar a les updates.

5.2.3 Inputs i Control sobre el Joc

Donat que la intenció del motor és estar preparat per programar un videojoc, una característica que no podem ignorar es l'input per part del jugador, aquesta tasca se li dona a la mateixa llibreria que s'ha de fer càrrec del control sobre el programa i la finestra.

Una de les opcions més interessants per fer servir és SDL, donat que porta un munt de funcionalitats, com un ampli suport per controls de consola, o la capacitat de reproduir sons de manera integrada. Tot i així, però, com que el wrapper més actualitzat d'aquesta llibreria a Python és Pygame, he decidit fer servir una alternativa, donat que prendre de base Pygame per intentar donar un producte millor, li treu valor al resultat.

Això és el que ha portat com a elecció de la llibreri **glfw** pel control de la finestra i els inputs, i pel que fa als sons, es farà servir la llibreria de **sounddevice**, que permet executar sons d'arxius **wav**.

Tal com treballa **glfw**, a l'objecte **Game** es guardarà un apuntador a l'objecte de la finestra, i també s'encarregarà de controlar els events de input, tan de teclat com de ratolí. Els ha de controlar per separat, pero de manera similar. S'ha de crear un mètode amb els paràmetres adients, i després assignar-los com a escoltadors per la finestra amb una funció de **glfw**. El cas del ratolí és senzill, es guarda la posició del cursor, i l'estat dels botons (sent polsats, mantenint-los polsats, alliberats o sense polsar), pel que fa el teclat però, a part de la tecla en concret i de l'estat, també hi ha retroalimentació d'atenció a tecles especials, indicant si, per exemple, es polsa el shift, el control, o la tecla alt. Una vegada entenem això, no és estrany el com funcionarà el sistema de inputs, des del loop de l'objecte tipus Game, el primer que es fa es cridar al mètode **poll_events**, que és el que farà que la finestra escolti les actualitzacions dels inputs, després, dins la funció d'escoltador del ratolí, es guardara en una variable les dades mencionades anteriorment, i pel que fa al

teclat, com que l'escoltador s'hauria d'executar per cada tecla actualitzada, el més adient és guardar les dades en una llista o un diccionari, amb l'estat i la resta d'elements referents a l'input de la tecla. Per últim, totes aquestes dades podran ser passades per paràmetre a l'escena, i des del mètode **update** de escena cap al mètode equivalent de les entitats, i d'aquestes als components en cas de ser necessari.

Dit això, una situació que es repeteix en motors com Unreal Engine o Unity, és el fet d'haver de preparar un controlador d'inputs diferent cada vegada, si es fa bé, només s'haurà de generar un a cada escena. Prenent aquesta tendència com a referència, la classe escena té un mètode anomenat input, que s'executara al principi de cada update de la escena. La intenció de fer això així és encaminar al programador a configurar aquest mateix mètode com al controlador que en altres motors hauria estat a algun objecte de l'escena, la qual cosa pot generar confusió, perquè algú sense experiència, probablement no sabrà si la funció del control de inputs ha de ser el propi personatge, o un element diferent de l'escena. Deixant això preparat així, forçarem que la responsabilitat de rebre els inputs del jugador es centralitzi, donant lloc a un codi més solid.

5.2.4 Control de referències

A Python, el programador té poc control sobre la memòria, i s'ha de tenir present el sistema intern de control sobre aquesta que usa Python per donar una bona experiència al programador, donat que s'ha de preparar el sistema per que s'alliberi l'espai en memòria ocupat per elements que estan en desús, i la intenció és aconseguir-ho sense que el desenvolupador hagi de cridar manualment la comanda "del"de les variables no usades. Si l'esquema d'elements es construeix correctament, podem delegar amb confiança el control de la memòria a Python, per així poder esborrar correctament elements en quan es canvi d'escena.

Internament, els elements de Python no deixen de ser estructures de C que tenen un apuntador a un objecte, i un comptador de quantitat de referències. Això funciona així per què el "Garbage colector"de Python elimina els elements quan el seu comptador arriba a zero, donat que, lògicament, el programa no n'estarà fent cap ús, a la figura 5.2 podem veure un exemple.

El més important per poder assolir un bon comportament per part del "Garbage colector"és evitar referències cícliques per part del codi del motor.

Diem que tenim una "referència cíclica"quan hi ha un conjunt d'objectes desconnectat del programa, però amb referències entre ells, evitant així ser eliminats per Python.

Un exemple que mostra molt fàcilment aquesta situació és la relació entre elements pares i fills en objectes de la jerarquia d'una escena, imaginant una escena com la de la figura 5.3, on la entitat 1 vol interactuar amb la entitat 2.

És natural que un element fill pugui requerir tenir capacitat per accedir als seus elements immediatament superiors dins la jerarquia (un exemple, podria ser un objecte dins d'una escena demanant a l'escena on habita, l'accés a un altre

```c
static PyObject* _hello(PyObject* self, PyObject* args){
    PyObject* name;
    if(!PyArg_ParseTuple(args, "U", &name)){
        return NULL;
    }
    PyObject* up = PyObject_CallMethod(name,"upper", NULL);
    if(!up){
        return NULL;
    }
    PyObject* result = PyUnicode_FromFormat("hello %U", up);
    Py_DECREF(up);
    return result;
}
```

Figura 5.2: Exemple de decreixement de referències a objectes de Python des de codi de C

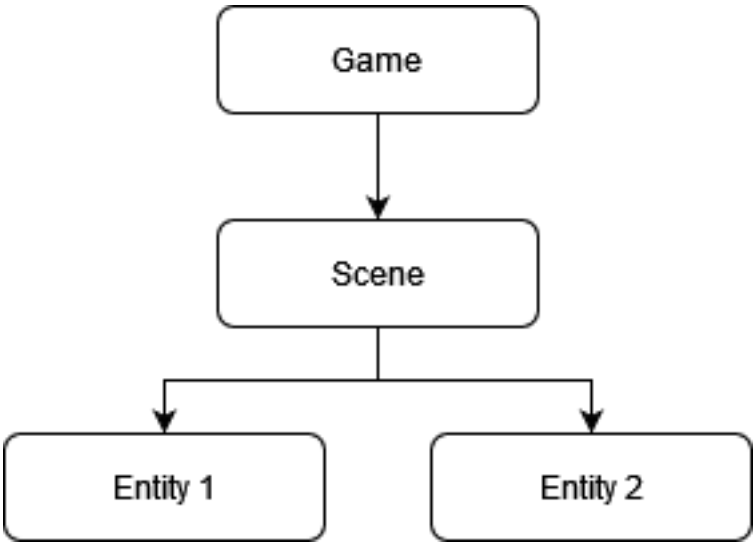


Figura 5.3: Exemple jerarquia d'elements d'una escena

element), però s'ha d'evitar que guardin referències a aquests en forma de variables internes, donat que donaran peu a una referència cíclica. Una manera d'evitar-ho, donant però accés als elements superiors, és passar-los per paràmetre en els mètodes de bucles, és a dir, podem passar l'escena pel mètode update de les entitats.

També val la pena comentar el cas específic de l'objecte de tipus Game, com que és un objecte que es destruirà quan s'aturi l'execució del programa. Aquest element es podria assignar a una variable de l'escena i així ens estalviaríem un paràmetre a passar al mètode update.

Seguint aquestes idees per evitar problemes de referències cícliques, des de tots els

components es podrà accedir a l'escena, i des de l'escena es podrà accedir a Game i a totes les entitats de l'escena. Les entitats donaran accés a tots els seus components, mantenint una via per comunicar-se amb tots els elements del programa en tot moment.

5.3 Segona iteració: Creant un joc

A la segona part del projecte es va polir el codi intern d'alguns mètodes del motor, revisant el funcionament de les col·lisions, corregint les aplicacions de les matrius de transformació, i afegint so. També es van afegir algunes característiques menors, com per exemple, renderitzat d'elements transparents, tot i que, donada la complexitat de moltes d'aquestes característiques, s'han implementat d'una manera simple per obtenir un resultat visible, encara que no tingui la millor implementació.

Dit això, l'altra codi important que es va desenvolupar en aquesta segona part va ser una demo de un joc usant el codi desenvolupat fins ara. El joc escollit per fer de prova és Pong, es considera un dels primers videojocs, i és prou senzill com per desenvolupar-lo sense gaires problemes.

Un cop tinguem un joc fet amb el nostre codi, la intenció és comparar el resultat amb un altre joc de Pong, però fet en una altra llibreria, buscant un exemple[7] a internet, per comparar un resultat amb l'altre.

5.3.1 Disseny del joc

A la figura 5.4 podem contemplar un exemple senzill de la implementació del joc de Pong, a simple vista, hi ha tres elements obvis que s'hauran d'implementar, els dos jugadors, i la pilota. Per deixar clar els elements a desenvolupar, ara allistaré les funcions que necessita el joc per funcionar, i després mostraré maneres d'implementar-les conceptualment.

- **Jugadors:** Necessitem que els jugadors es puguin moure a dalt i a baix, i fagin rebotar la pilota, és a dir, com a mínim, cada jugador tindrà un component de col·lisió
- **Parets:** Els màrges superior i inferior han de fer rebotar la pilota, raó per la qual també contindran com a mínim una component de col·lisió.
- **Porteries**: Han de poder detectar que toquen la pilota per anotar els punts, son diferents a les parets perquè hem de contar els punts.
- **El marcador:** Necessitem algun sistema per contar els punts anotats per cada jugador, i mostrar el resultat als jugadors.
- **La pilota:** Ha de tenir moviment, rebotar, hem de detectar quan toca una porteria.

[7] https://github.com/techwithtim/Pong-Python

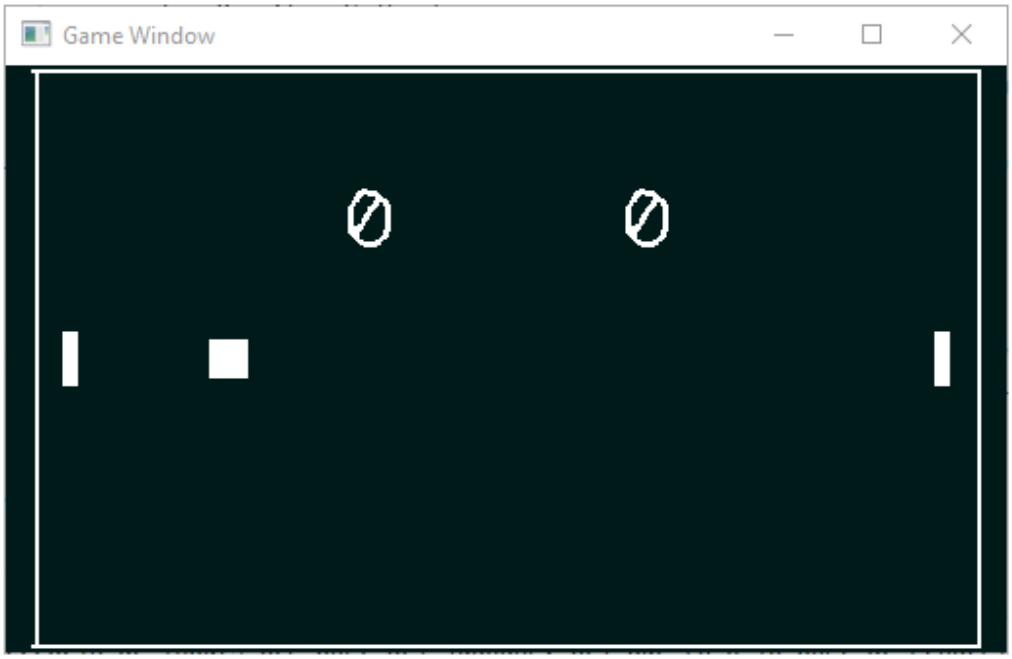


Figura 5.4: Un exemple del joc de pong (aquest en concret està desenvolupad dins aquest projecte).

```
dump_data( create_data_quad2D((-1.2, 0.71, 0), (1.2, 0.71, 0), (1.2, 0.72, 0)),
           impact_quad2D, translocate_quad2D)
```

Figura 5.5: Exemple d'ús dels components de col·lisió

5.3.2 Com construir l'escena

De cara al control dels jugadors, es crea una classe filla de escena on s'esperin quatre inputs, per moure cada jugador amunt o avall, i farem servir aquesta escena per afegir els elements del joc. Per afegir els murs, s'han de generar dues entitats iguals, el procés d'afegir una entitat que col·lisioni i sigui renderitzada es repeteix molt, per això només el descriuré una vegada: per donar la capacitat de col·lisionar, se li afegeix un component de **Colider**, i se li dona al component la informació pertinent amb el mètode **dump_data** seguint l'exemple de la figura 5.5.

Ara, per afegir visuals a l'element, se li ha de donar una malla de vèrtex, i després afegir una textura, els dos components s'afegeixen quan s'afegeix un component **Renderable**, que generarà i afegira components de malla (MeshComponent al codi) i de textura (TextureComponent). Llavors, per afegir una malla específica, s'han de generar els punts, el mètode **make_sprite** de la figura 5.3.2 genera una malla quadrada, i se li pot donar una textura seguint la guia de la figura 5.3.2.

Ara, per afegir la entitat a la escena la passarem amb el mètode **spawn_entity** per afegir-la amb un nom i posició concrets (figura 5.3.2)

```
obj.components['MeshComponent'].set_model(make_sprite(points=((-1.2, 0.71, 0), (1.2, 0.71, 0), (-1.2, 0.72, 0), (1.2, 0.72, 0))))
```

Figura 5.6: Exemple per afegir una malla a una entitat (obj)

```
obj.components['TextureComponent'].texture = load_texture(r"ruta de la imatge")
```

Figura 5.7: Exemple per carregar i especificar una textura pel component de textura de una entitat (obj)

Ara, les porteries, les entitats de jugadors i la pilota seguiran el mateix procediment (sols que amb les coordenades apropiades), i podem crear un component específic per la pilota, que detectarà quan hi ha una col·lisió, si és qualsevol objecte, simplement rebotarà, però com que la informació que dona la escena sobre l'impacte conte la referència a l'entitat amb que s'ha impactat, podem fer que si es detecta que ha col·lisionat amb una porteria, torni a la zona d'origen, i afegeixi un punt al jugador adient, això ho gestionarà una altra entitat, a la qual donarem un altre component que emmagatzemarà la puntuació, i actualitzarà unes altres entitats, que seran les que mostren als jugadors els punts anotats.

Hi ha un component que hereta de Renderable fet per gestionar un element que mostri el que en el mon dels videojocs s'anomena Sprite[8], al qual se li ha de donar les coordenades de cada frame diferent del sprite (es pot fer servir el mètode **sprite_frames_by_grid** del paquet d'utilitats per generar-les) amb el mètode **set_frames** de la figura 5.3.2, i una llista de animacions, on les diferents animacions estaran representades pels conjunts de frames de la animació, com a la figura 5.3.2

5.3.3 Herència d'escena

Per aportar utilitats al motor, s'han afegit una serie de classes filles a la classe component, les de la figura 5.11.

- **MeshComponent:** Conte les dades corresponents a una malla a renderitzar.
- **TextureComponent:** Conte una imatge en forma de numpy array.
- **ShaderComponent**: Conte el programa de shader.
- **RenderableComponent:** Processa les dades dels components que s'han d'usar pel renderitzat.

[8] https://www.educative.io/answers/definition-sprite

```
escena.spawn_entity(entitat, (0, 0, 0), 'nom')
```

Figura 5.8: Exemple per afegir una entitat a la escena

renderable.set_frames(sprite_frames_by_grid((10, 1)))

Figura 5.9: Exemple per afegir una entitat a la escena

renderable.set_states(((0,),(1,),(2,),(3,),(4,),(5,),(6,),(7,),(8,),(9,),))

Figura 5.10: Exemple per afegir una entitat a la escena

- **SpriteAnimator:** Processa les dades dels components que s'han d'usar pel renderitzat de tal manera que processa una textura animada.
- **ColiderComponent:** Conte les dades corresponents a un cos de col·lisió.

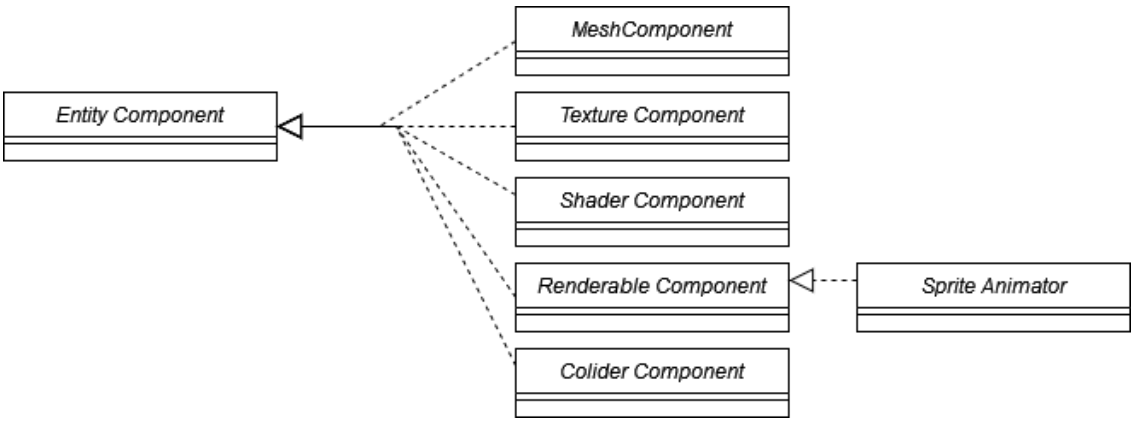


Figura 5.11: Classes filles de component

Capítol 6

Resultats i Validació

6.1 Resultats

D'aquest projecte, ha resultat un codi del què en podem destacar tres característiques provat per les "Demos"desenvolupades: el suport per col·lisions (figura 6.1), on la pilota rebota quan detecta la col·lisió contra tots els elements, i quan l'element en qüestió és una porteria, dona a un element que no és visible l'ordre d'actualitzar la puntuació i tornar la pilota al centre, les animacions (figura 6.2 i figura 6.3), on la demo mostra un element que canvia la seva animació en polsar la barra d'espai, i el processat de perspectives en 3D (figura 6.4), a la demo es pot controlar la camara amb el ratolí i WASD.

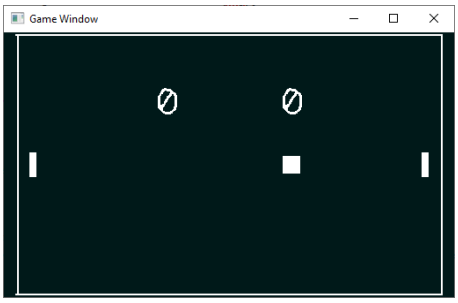


Figura 6.1: Mostra de col·lisions

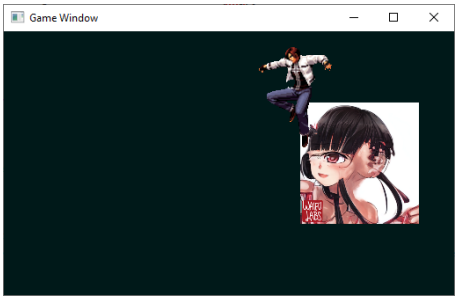


Figura 6.2: Mostra d'animacions en un estat

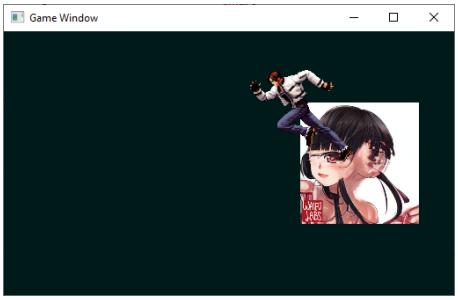


Figura 6.3: Mostra de la mateixa escena, però s'ha canviat l'estat per codi

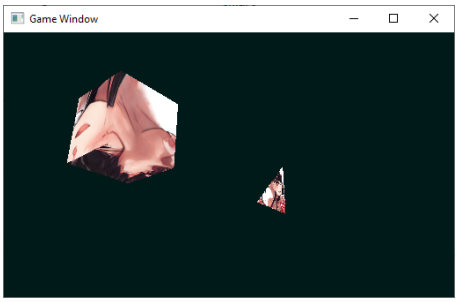


Figura 6.4: Mostra de les capacitats 3D

6.2 Validació

Ara que ja tenim una demostració de les capacitats del motor, cal comparar els resultats amb l'exemple de pygame mencionat anteriorment , compararem els recursos usats per la màquina, i, tot i que pot resultar una mètrica subjectiva, la netedat del codi.

El codi de la figura 6.5 representa només la gestió de la col·lisió de la bola a l'exemple de PyGame, en canvi, la figura 6.6 conte tot el codi relacionat amb la pilota al motor implementat en aquest projecte. Podem dir que codi és més net i estable, donat que l'altre exemple requereix molt més codi per aconseguir el mateix que a la implementació d'aquest projecte, i a més, el codi és molt més similars a les altres entitats del programa, aportant un desenvolupament còmode i estructurat.

De cara al consum de recursos, podem veure a la figura 6.8 un major fluxe de comunicació amb la GPU, però a la figura 6.7 podem veure un consum major de CPU, i un major us de memoria

```
    for i in range(10, HEIGHT, HEIGHT // 20):
        if i % 2 == 1:
            continue
        pygame.draw.rect(win, WHITE, (WIDTH // 2 - 5, i, 10, HEIGHT // 20))

    ball.draw(win)
    pygame.display.update()


def handle_collision(ball, left_paddle, right_paddle):
    if ball.y + ball.radius >= HEIGHT:
        ball.y_vel *= -1
    elif ball.y - ball.radius <= 0:
        ball.y_vel *= -1

    if ball.x_vel < 0:
        if ball.y >= left_paddle.y and ball.y <= left_paddle.y + left_paddle.height:
            if ball.x - ball.radius <= left_paddle.x + left_paddle.width:
                ball.x_vel *= -1

                middle_y = left_paddle.y + left_paddle.height / 2
                difference_in_y = middle_y - ball.y
                reduction_factor = (left_paddle.height / 2) / ball.MAX_VEL
                y_vel = difference_in_y / reduction_factor
                ball.y_vel = -1 * y_vel

    else:
        if ball.y >= right_paddle.y and ball.y <= right_paddle.y + right_paddle.height:
            if ball.x + ball.radius >= right_paddle.x:
                ball.x_vel *= -1

                middle_y = right_paddle.y + right_paddle.height / 2
                difference_in_y = middle_y - ball.y
                reduction_factor = (right_paddle.height / 2) / ball.MAX_VEL
                y_vel = difference_in_y / reduction_factor
                ball.y_vel = -1 * y_vel
```

Figura 6.5: Codi de col·lisio de la bola pel pong de PyGame

```
146
147 class ballComponent(EntityComponent):
148
149     def __init__(self):
150         self.movement = np.array((0.01, 0, 0), dtype = np.float32)
151
152     def update(self, entity):
153         if entity.impacts:
154             si, distance, point, normal, other = entity.impacts[0]
155             if other.name in ('Rigborder', 'leftborder'):
156                 entity.score(entity.position)
157                 entity.set_position(Vector3D((0, 0, 0)))
158                 self.movement = -self.movement
159             else:
160                 vp = Vector3D((0, 0, 0))
161                 for s, t, p, n, o in entity.impacts:
162                     vp += p
163                 s = -normalize(vp/len(entity.impacts)-entity.position)*0.01
164                 if abs(s[1]) > 0.009:
165                     s[0] = self.movement[0]
166                 self.movement = s
167         entity.move_position(self.movement)
168         pass
169 obj.add_component(ballComponent())
170 c = Colider()
171 c.dump_data(create_data_quad2D((-0.05, -0.05, 0), (0.05, -0.05, 0), (0.05, 0.05, 0)), impact_quad2D, translocate_quad2D)
172 obj.add_component(c)
173 obj.components['MeshComponent'].vertices = v
174 obj.components['MeshComponent'].indices = i
175 obj.components['TextureComponent'].texture = im
176 obj.score = sc.components['SCComponent'].score
177 s.spawn_entity(obj, (0, 0, 0), 'ball')
```

Figura 6.6: Codi de la bola al motor d'aquest projecte

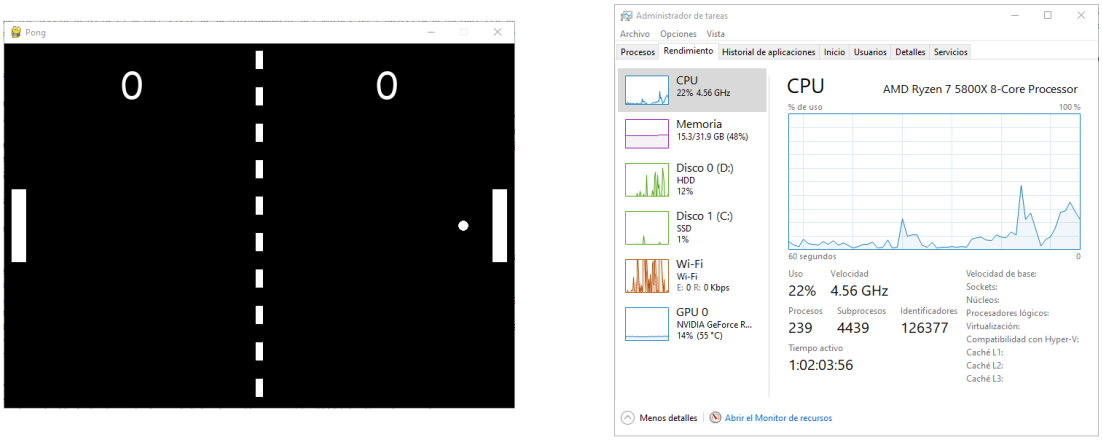


Figura 6.7: Codi d'exemple de PyGame

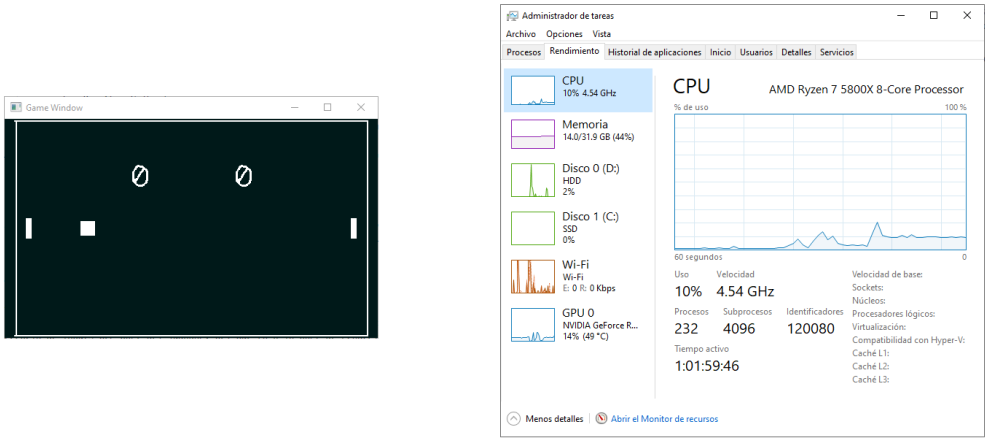


Figura 6.8: Codi pong al motor d'aquest projecte

Capítol 7

Conclusions i Feina Futura

7.1 Conclusions

Pygame és una llibreria robusta, tot i les mancances que pugui tenir, però li falta un element molt important per afavorir al desenvolupament, una estructura, donat que hi ha exemples de jocs que l'usen i realment no tenen un mal rendiment (dins de usar python), però delegar la tasca de construir una arquitectura al desenvolupador fa que sigui fàcil generar problemes al codi, que és el més comú als programadors novells, els que és més fàcil que escolleixin aquesta llibreria. A més a més, tot la manca d'estructura fa que a les definicions més complexes de "motor de joc"no hi encaixi.

La implementació feta en aquest projecte aporta una estructura que afavoreix un desenvolupament estable per entitats simples o que es repeteixin molt, a més de suport per renderitzat de polígons tridimensional, i col·lisions més complexes i utilitza un sistema amb components, que és similar a altres motors de joc, donant peu a que si un desenvolupador d'aquests altres motors el prova, estigui una mica familiaritzat.

De cara al motor gràfic, separa totalment les dades dels elements a renderitzar dels mètodes que les processen, permeten així que de cara a futur es pugui establir comunicació amb la gpu usant altres llibreries. Aquest requeriment pot ser útil per casos en que certs dispositius com les consoles usen llibreries específiques pel processat gràfic.

7.2 Feina futura

Hi ha elements que requereixen una implementació més complexa, com per exemple, la renderització d'elements transparents, perquè la que hi ha no funcionarà correctament a tots els casos.

Es pot implementar un sistema de col·lisions "per capes"similar al que tenen Unity o Unreal per que no es comprovin col·lisions amb elements no pertinents,

cosa que ajudaria al desenvolupament de qualsevol joc.

Es podrien arribar a desenvolupar sistemes per guardar components i entitats prefetes en arxius per poder generar entitats de manera repetida i a diferents escenes, i també un sistema amb el patró de disseny Factory per ajudar a generar entitats iguals amb codi més simple

Es pot alterar la pipeline per que els shaders puguin processar llums.

Appendices

Apèndix A

Manual Tècnic

Aquí es documenta com interactuar amb el projecte, es pot descarregar al repositori de Git Hub https://github.com/ArcaDevProject/3DPythonEngine

A.1 Requeriments mínims

El projecte s'ha desenvolupat usant Python 3.10, però es pot executar usant qualsevol interpretador de Python compatible amb els paquets dels requeriments, les llibreries en qüestió són les següents: cffi==1.15.1
glfw==2.5.6
multipledispatch==0.6.0
numpy==1.24.2
Pillow==9.4.0
pycparser==2.21
PyOpenGL==3.1.6
PyOpenGL-accelerate==3.1.6
pyrr==0.10.3
six==1.16.0
sounddevice==0.4.6
soundfile==0.12.1
es poden instalar a l'interpretador de Python usant la comanda *pip install* (*nom del paquet*).

Una alternativa més senzilla és usar entorns de desenvolupament com Pycharm. IDEs com aquest ofereixen la instal·lació de les dependències necessàries quan les detecten a l'arxiu *requirements.txt* dins el projecte.

A.2 Com utilitzar 3D Python Engine?

Per executar les demostracions del projecte, només cal executar amb l'interpretador de Python preparat els arxius amb el prefix DEMO al repositori del projecte. L'execució dels arxius de demostració es torna més senzilla usant entorns de desen-

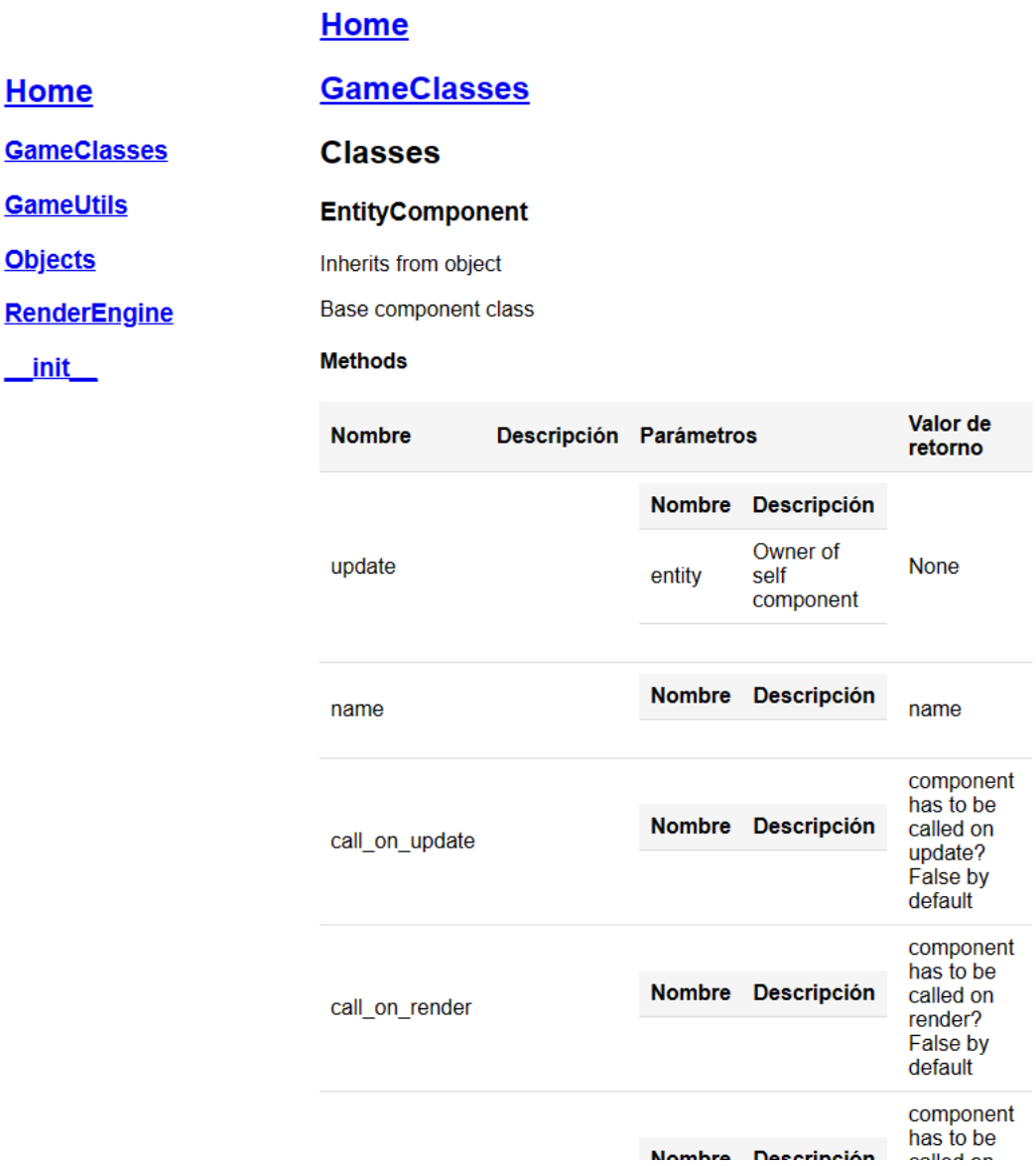
volupament com PyCharm on pots indicar un arxiu especific a executar.

Apèndix B

Documentació de l'API del motor 3D Python Engine

L'aspecte visual de la documentació del codi es pot veure a la figura B.1 i a la figura B.2. Dins es poden veure les classes que hi ha a a cada arxiu, una descripció de la seva funció, i la informació pertinent dels seus mètodes. En els arxius on només hi han funcions tan sols es mostren aquestes. Es pot navegar entre cada arxiu usant els vincles que hi han

Es pot accedir a la documentació del projecte extraient els arxius de l'arxiu comprimit *Doc.zip* al repositori del projecte.



Home

GameClasses

GameUtils

Objects

RenderEngine

__init__

Home

GameClasses

Classes

EntityComponent

Inherits from object

Base component class

Methods

Nombre	Descripción	Parámetros		Valor de retorno
update		Nombre	Descripción	None
		entity	Owner of self component	
name		Nombre	Descripción	name
call_on_update		Nombre	Descripción	component has to be called on update? False by default
call_on_render		Nombre	Descripción	component has to be called on render? False by default
		Nombre	Descripción	component has to be

Figura B.1: Visual de la documentació d'aquest projecte

Home

GameClasses

GameUtils

Objects

RenderEngine

__init__

Home

GameUtils

Functions

Nombre	Descripción	Parámetros		Valor de retorno
make_mesh		Nombre	Descripción	An array with the vertices from a model, and an array containing the faces
		points	vertices	
		normals	normals for each vertex	
		uvs	uv for each vertex	
		faces	list of vertices forming faces	
load_mesh		Nombre	Descripción	An array with the vertices from a model, and an array containing the faces
		filename	File to load	
make_colision_mesh		Nombre	Descripción	vertices, center and max distance from any vertex to the center
		vertices	vertices	
		indices	list of vertices for the faces	
		Nombre	Descripción	
		points	(optional) points of the	

Figura B.2: Visual de la documentació d'aquest projecte

Bibliografia

[1] Overvoorde, A. (n.d.). OpenGL - Introduction, from
https://open.gl/introduction

[2] Batut, C.; Belabas, K.; Bernardi, D.; Cohen, H.; Olivier, M.: User's guide to *PARI-GP*,
pari.math.u-bordeaux.fr/pub/pari/manuals/2.3.3/users.pdf, 2000.

[3] CRYENGINE | The complete solution for next generation game development by Crytek. (n.d.). CRYENGINE. Retrieved February 19, 2023, from
https://www.cryengine.com/

[4] YoYo Games. (n.d.). Make 2D Games With GameMaker | Free Video Game Maker. GameMaker. Retrieved February 19, 2023, from
https://gamemaker.io/es

[5] Engine, G. (n.d.). Godot Engine - Free and open source 2D and 3D game engine. Godot Engine. Retrieved February 19, 2023, from ,
https://godotengine.org/

[6] Pygame. (n.d.). Pygame Official Website. Retrieved February 19, 2023, from
https://www.pygame.org/news

[7] The Ren'Py Visual Novel Engine. (n.d.). Retrieved February 19, 2023, from
https://www.renpy.org/

[8] R., R., & R. (n.d.). Make Your Own Game with RPG Maker. Retrieved February 19, 2023, from
https://www.rpgmakerweb.com/

[9] Unity. (n.d.). Unity Official Website. Retrieved February 19, 2023, from
https://unity.com/

[10] Unreal Engine | The most powerful real-time 3D creation tool. (n.d.). Unreal Engine. Retrieved February 19, 2023, from
https://www.unrealengine.com/en-US/

[11] Scratchapixel. Retrieved February 19, 2023, from
https://scratchapixel.com/