
This is the **published version** of the bachelor thesis:

Sanchez Amat, Isaac. *Simulador gràfic i interactiu de cossos celestes en Vulkan*.
Treball de Final de Grau (Universitat Autònoma de Barcelona), 2026
(Enginyeria Informàtica)

This version is available at <https://ddd.uab.cat/record/334360>

under the terms of the  license.

Simulador gràfic i interactiu de cossos celestes en Vulkan

Isaac Sánchez Amat

30 de juny de 2026

Resum— En aquest treball es desenvolupa un entorn de simulació gràfic i interactiu de cossos celestes, per a visualitzar i configurar escenaris reals o ficticis. Es detallen els objectius del desenvolupament, així com el marc teòric basat en la formulació física del moviment d'aquests cossos i s'esmenten els aspectes bàsics per un entorn gràfic, fent servir la API gràfica de Vulkan, que té una corba d'aprenentatge més pronunciada que d'altres APIs com OpenGL, però cada cop està més estandaritzada en l'àmbit dels gràfics per ordinador, ja que ofereix control dels recursos de la placa gràfica a baix nivell. A més, Vulkan permet l'ús dels compute shaders, cada cop més importants en un context on es busca l'optimització a partir del càlcul paral·lel en GPU. També, s'exposen els resultats d'aquest treball, que validen les possibilitats de visualització i simulació de les escenes. Finalment es destaquen les conclusions del treball i possibles millores.

Paraules clau— Vulkan, Simulació, Cossos celestes, Newton, Compute shaders.

Abstract— In this work we develop a graphical and interactive simulation environment of celestial bodies, to visualise and set up real or fictional scenes. We define the development objectives, as well as the theoretical framework based on physics formulas of the movement of these bodies and we establish the basis of a graphical environment, using the graphics API of Vulkan, that has a more pronounced learning curve than others like OpenGL, but nowadays it tends to be more standardised in the computer graphics field, because it offers more low-level settings. In addition, Vulkan allow the use of compute shaders, that now are more important in a context where we search for the optimisation by parallelizing tasks in the GPU. Then, we expose the results of this work, to demonstrate the possibilities of visualisation and simulation of the scenes. Finally we stand out the conclusions and possible improvements of the work.

Keywords— Vulkan, Simulation, Celestial bodies, Newton, Compute shaders.

1 INTRODUCCIÓ

FA anys que els gràfics per computador permeten entendre fenòmens científics a partir de simuladors gràfics i interactius, doncs tenen un gran potencial per tal de representar escenes realistes o per simular esdeveniments que encara no s'han explorat. Un dels grans avantatges dels entorns gràfics és que es pot representar l'evolució d'escenes a partir d'equacions físiques força realistes, el que genera simulacions més acurades que amb animació

convencional.

Una simulació de les òrbites de planetes, llunes i estrelles pot ser útil a l'hora d'entendre el moviment dels astres a una escala planetària, així com visualitzar com un satèl·lit aconsegueix una òrbita al voltant de la Terra, preveure òrbites d'objectes a l'espai o bé simular quin efecte tenen cossos massius com Júpiter vora la Terra. Una aplicació d'aquestes característiques pot ser útil en entorns acadèmics per a explicar fenòmens físics de l'espai o resoldre exercicis de manera visual. Els sistemes de simulacions en temps real permeten generar moviments i imatges realistes i entenedores per l'ull humà. Tot i això, aquest tipus de sistemes requereixen una alta potència de càlcul per resoldre equacions matemàtiques complexes aplicades a una gran quantitat de cossos en el menor temps possible.

Els gràfics per computador segueixen avançant en aspectes de visualització gràfica 3D, sobretot en optimitza-

• E-mail de contacte: 1667262@uab.cat

• Menció: Computació

• Treball tutoritzat per: Enric Martí Gòdia (Departament Ciències de la Computació)

• Curs 2025/26

ció. Actualment l'API gràfica Vulkan [1] és la millor opció, doncs permet optimitzar la representació i càlcul de visualitzacions complexes fent ús dels compute shaders per tractar grans quantitats de dades en la GPU. A més, Vulkan tracta processos a més baix nivell que altres APIs gràfiques més antigues, com OpenGL [2], i això permet als programadors crear entorns gràfics que optimitzen l'ús dels recursos hardware concrets que es disposin (targetes gràfiques).

Un exemple d'un sistema de simulacions planetàries és el videojoc Outer Wilds [3] desenvolupat en Unity [4], el qual aconsegueix una representació i simulació d'un sistema planetari fictici, on l'ambientació està fortament suportada per fenòmens químics i físics. En aquest cas, es demostra que la representació de planetes no està limitada a l'àmbit acadèmic, sinó que es poden utilitzar en àmbits d'entreteniment.

Tot i així, el principal referent d'aquest treball no és un videojoc, sinó un entorn de simulacions de sistemes planetaris dissenyat per un estudiant de la Universitat de Plovdiv, Bulgària [5]. Aquest treball consta d'un entorn propi desenvolupat en OpenGL, amb l'equació de la gravitació universal de Newton [6] com a base de la simulació. A la figura 1 s'observa una captura de pantalla de l'entorn 3D amb planetes i òrbites. Aquest treball ha sigut font d'inspiració pel nostre treball.

El que es busca amb el nostre treball és definir un disseny òptim per la implementació d'un entorn de simulacions gràfiques per sistemes planetaris, que sigui prou parametrizable com per crear escenes realistes o fictícies, però que igualment compleixin les lleis físiques.

Per tot això el principal objectiu d'aquest projecte és desenvolupar una aplicació d'aquestes característiques, utilitzant la llibreria gràfica Vulkan.

Com a funcionalitats o objectius més concrets destaquem:

- Creació i configuració d'objectes esfèrics (cossos celestes), tant de les seves dimensions com de les seves característiques físiques (massa, posició i velocitat).
- Simulació de gravitació universal òptima i viable per la visualització d'òrbites.
- Navegació per l'espai a partir d'inputs del teclat i ratolí, amb diferents modes de càmera.
- Creació aleatòria de textures dels planetes, a partir de funcions de soroll, per poder generar entorns ficticis.

En el nostre treball no pretenem aprofundir en conceptes bàsics de gràfics 3D, però sí que expliquem aspectes interessants sobre el funcionament de Vulkan i les llibreries i algorismes que s'utilitzen. Ens centrem més en explicar i documentar cada part del desenvolupament i implementació, així com justificar l'ús dels algorismes utilitzats.

Una part important del projecte és l'elecció de les eines de desenvolupament i llibreries externes.

A continuació es mostra una llista de les eines concretes que s'han fet servir pel control i desenvolupament del treball:

- Visual Studio [7] pel desenvolupament en C++ i test de l'aplicació.

- Github [8] per la organització del treball i gestió de versions.
- Premake [9] per la generació de projectes de Visual Studio.
- ShaderToy [10] per facilitar la visualització de fragment shaders.
- VS Code [11] per la implementació d'arxius addicionals, com arxius de premake o shaders.
- Blender [12] per la creació de models 3D.
- Google sheets [13] pel seguiment del treball dividit en tasques i subtasques.
- Canva [14] per la realització de diagrames i de representacions esquemàtiques del software.

A continuació es concreten les llibreries externes utilitzades per ajudar al desenvolupament de les funcionalitats:

- Vulkan [1] amb l'API gràfica i el seu kit de desenvolupament (SDK) utilitzada per la comunicació entre CPU i GPU.
- GLFW [15] per la gestió de finestres de l'aplicació.
- GLM [16] per a les operacions matemàtiques per gràfics 3D.
- ImGUI [17] per la creació d'una interfície gràfica intuïtiva, des de la qual accedir als diferents paràmetres de l'aplicació.
- ASSIMP [18] per la càrrega de models 3D.

La metodologia que s'ha utilitzat per la realització del treball ha sigut Kanban [19] amb una cua de prioritats, per tal d'aconseguir un MVP (Minimum Viable Product). Aquesta metodologia divideix les tasques del treball en tres estats diferents: 'To do' que són les tasques a realitzar, 'In progress' que són tasques que estan en fase de desenvolupament o validació i 'Done' que són les tasques que ja s'han validat i queden completades. Això ajuda a gestionar la feina de manera gràfica, raó per la qual es fàcil reconèixer les tasques més importants per arribar a punts clau del desenvolupament del treball. S'ha escollit aquesta metodologia per la facilitat de visualitzar quines tasques s'han completat i quines queden pendents. Es pot mantenir un flux de progressió formalitzant els avenços i és perfectament aplicable en treballs d'una sola persona. A la figura 2 es poden veure algunes tasques utilitzant Kanban i Google Sheets.

Aquest informe consta de les següents seccions: en la secció 2 expliquem el marc teòric sobre el que hem treballat, en la secció 3 s'explica el desenvolupament de l'aplicació, en la secció 4 es mostren els resultats i demostracions i finalment en la secció 5 les conclusions i millores del treball.

2 MARC TEÒRIC

En aquesta secció explicarem la física de cossos celestes, la principal eina gràfica per la creació de simulacions (Vulkan) i la interfície d'usuari utilitzada.

2.1 Física de cossos celestes

La física ofereix múltiples models per representar l'atracció de cossos a nivell planetari. En aquest treball ens basem en l'aplicació de la llei de gravitació universal de Newton [6], doncs segons alguns estudis, és un estàndard en simulacions senzilles i força realistes [20]. L'equació (1) descriu la relació de la força d'atracció exercida per un cos sobre els altres (F) respecte les masses actants ($m1$ i $m2$) i la distància entre aquests cossos (d). Existeixen altres models més complexos, com la llei de la relativitat d'Einstein [21], però és justament aquesta complexitat que fa que sigui més difícil de representar i visualitzar.

$$F = G \frac{m1 \times m2}{d^2} \quad (1)$$

Per definir el moviment d'un cos necessitem la velocitat. Mentre la força atrau als cossos entre si, la velocitat corba aquesta atracció, creant les òrbites que coneixem. Podem obtenir l'acceleració (a) a partir de la força exercida entre la massa del cos atret (eq. 2). Després, fem servir la integració d'Euler [22] per calcular la velocitat (v) (eq. 3) i la posició (p) (eq. 4), utilitzant un increment de temps fixe.

$$a = \frac{F}{m} \quad (2)$$

$$v = v + a \times \Delta t \quad (3)$$

$$p = p + v \times \Delta t \quad (4)$$

2.2 Perquè Vulkan?

Els creadors de Vulkan [1], el KhronosGroup, defineixen aquesta API com l'evolució d'OpenGL [2], doncs expandeix les seves característiques, com la capacitat d'utilitzar 'Instancing' [23] o 'Compute shaders' [24]. Vulkan, però, tracta aspectes a més baix nivell que OpenGL, el que permet aprofitar els recursos gràfics hardware disponibles. A més, és compatible amb C++, un llenguatge de programació força llegible i en el que es poden separar molt bé els diferents mòduls del disseny fent servir la lògica del 'Model-Vista-Controlador' (diferents classes per gestió de les dades, visualització i execució).

Com a base de l'aplicació s'ha utilitzat la *VulkanBase* desenvolupada pel Raúl Velázquez Gómez [25].

Entenem per entorn gràfic una aplicació que permet renderitzar figures i controlar el seu comportament segons unes funcionalitats definides en l'entorn. La base d'un entorn gràfic consta d'una API gràfica, que és una llibreria que conté tot el codi necessari per representar i interpretar geometria. En el cas concret de Vulkan, la API que fem servir, també conté codi per configuracions de la GPU a més baix nivell.

En aquest treball veurem la base de Vulkan com una caixa negra parcial doncs, com es pot veure en un codi típic d'una base de Vulkan [26], aquest conté una gran quantitat de definicions i inicialitzacions de conceptes. A la pràctica Vulkan és una API gràfica molt més complexa que OpenGL des de l'inici. Per tant, obviarem una gran quantitat de conceptes per centrar-nos en els més importants per entendre el funcionament de l'entorn gràfic i de l'aplicació.

2.3 Interfície d'usuari amb ImGUI

És important que l'usuari pugui interactuar amb l'entorn, doncs el punt fort d'aquesta aplicació gràfica és la seva lliure visualització i parametrització. Per això hem escollit la llibreria ImGUI [17] com l'encarregada de gestionar les finestres de la interfície. Aquesta llibreria ofereix una gran quantitat de 'widgets' que permeten modificar paràmetres, presentant-ho d'una manera minimalista, tant en codi com en disseny, com es veu a la figura 3, on podem veure dues finestres: a l'esquerra una amb seccions desplegable i a la dreta un petit menú amb barres desplaçables, *checkboxes* o botons.

Aquesta llibreria es molt estàndard en el desenvolupament d'aplicacions gràfiques en C++ i GLFW.

3 DESENVOLUPAMENT

A continuació expliquem cada mòdul present en l'esquema general de l'aplicació, a la figura 4.

La *VulkanBase* funciona com el 'cervell' de l'aplicació. Consta de tot el codi necessari per poder representar vèrtexs per pantalla, fent servir comandes Vulkan. Com que aquest codi es molt extens i conté una gran quantitat d'estructures, hem decidit dividir la *VulkanBase* en tres sub-mòduls.

- **Pipelines:** en essència defineixen quins shaders es faran servir i quines dades d'entrada s'esperen.
- **Buffers:** contenen les dades d'entrada pels shaders. Vénen a ser uns arrays de tipus concrets. A més, Vulkan ofereix una gran quantitat de flags per definir l'ús dels buffers. Al igual que amb OpenGL, podem fer servir els buffers per guardar geometria o índexs. Tant la GPU com la CPU podrà accedir a aquests buffers.
- **Other Vulkan Structs:** aquest sub-mòdul conté tota la resta del codi necessari per fer funcionar Vulkan. Conté estructures com els DescriptorSets o les Images, necessàries per definir textures.

El mòdul de *ShaderPrograms* és igual de fonamental que la *VulkanBase*, doncs els shaders són la manera de dir-li a la GPU com i què volem visualitzar per pantalla. Principalment les pipelines consten de dos shaders: el VertexShader (què volem visualitzar: vèrtex) i el FragmentShader (com volem visualitzar-ho: píxels). Però Vulkan ens permet definir altres shaders, que normalment estan definits per defecte (quadres vermells de la columna esquerra de la figura 5, que mostra tots els elements bàsics de Vulkan que poden participar en una pipeline). També, aquesta API ens permet utilitzar compute shaders, que no tenen cap ús concret, més enllà del de paral·lelitzar tasques de càlcul no gràfic. Per a aquesta aplicació aquests shaders es faran servir per calcular el moviment dels planetes, lliurant així a la CPU de la càrrega de treball i enfocant l'ús d'aquesta unitat de còmput en l'execució del bucle principal. Podem observar a l'esquema general (figura 4) que aquests shaders es criden (fletxa call) des de la base. A més, poden enviar (fletxa send) o rebre (fletxa receive) dades dels buffers.

El mòdul *Scene* de l'esquema general (figura 4) conté totes les dades necessàries per representar un sistema planetari per pantalla. Aquestes dades no estan en els buffers, sinó

en estructures de dades o classes, com podrien ser les classes 'Planetes' o 'Camera'. De fet, els buffers serveixen per comunicar les dades de l'escena als shaders. Així doncs, l'escena podrà actualitzar els Buffers (fletxa update a l'esquema) amb noves dades, o bé sobreescriure les dades amb les dels buffers (fletxa read a l'esquema).

Per últim, l'entorn ha de ser parametrizable, és a dir, l'usuari ha de poder accedir a aquestes dades de l'escena i modificar-les. Això es fa a través del mòdul d'*ImGUI UI*. L'usuari modifica els 'widgets' de la interfície d'*ImGUI* associats a un paràmetre de l'escena (fletxa change d'*Usuari* a *ImGUI UI*). Després, aquests nous valors es transfereixen directament a les dades de l'escena (fletxa change d'*ImGUI UI* a *Scene*).

Per veure un codi d'exemple complet de Vulkan es pot accedir a la pàgina de VulkanTutorial [26], el qual explica en detall totes les parts de la base que s'ha fet servir.

En els propers punts s'expliquen les funcionalitats principals i més importants del treball, relacionant-ho amb cada mòdul de l'esquema general.

3.1 Representació de l'escena

Per entendre com es representa una escena concreta 3D per pantalla primer hem de saber quines dades i estructures de Vulkan són essencials:

1. Model i matriu MVP
2. Pipeline gràfica
3. Selecció de recursos i dibuixat
4. Bucle principal de l'aplicació

3.1.1 Model i matriu MVP

Primerament, hem de definir quines són les dades mínimes per representar un objecte per pantalla, com un poliedre. Aquesta unitat mínima són els vèrtexs, que consisteixen en una posició en l'espai expressat com un vector de tres dimensions (x, y, z). Unint tres vèrtex obtenim la figura primitiva de la majoria de APIs gràfiques: el triangle. Qualsevol poliedre es pot representar a partir de triangles.

Un cop posicionada la figura a l'espai 3D, volem que el nostre punt de vista (càmera) alteri la percepció de la figura. En gràfics 3D això es fa variant la posició de la figura depenent d'uns paràmetres de la càmera. La forma d'alterar aquestes posicions és aplicant la multiplicació de matrius MVP (Model-View-Projection) [27].

En el nostre entorn 3D, els poliedres són els planetes. El model dels planetes és l'esfera definida per la funció *GLU.Sphere()*, present en l'entorn de l'assignatura VGI [28] que ha sigut modificada per poder fer-se servir en la *VulkanBase*. Els vèrtexs d'aquest model es guarden en buffers de Vulkan amb la flag *VK_BUFFER_USAGE_VERTEX_BUFFER_BIT* que permet que aquest buffer es pugui fer servir com a entrada d'un vertex shader. Per poder moure aquest objecte per l'espai es pot aplicar una translació a la matriu *Model*, però per qüestions de disseny hem decidit crear una estructura de dades de *BodyPhysics* que conté les variables d'un objecte per la simulació física. Més endavant s'explica en detall aquesta estructura de dades. Però ara només cal entendre que conté

un camp *position*, el qual fem servir per variar la posició del nostre cos celeste. Aquesta posició pot estar en un altre buffer diferent, per diferenciar el que és el model del que són les dades físiques de cada cos.

3.1.2 Pipeline gràfica

Per a poder visualitzar la figura per pantalla necessitem una pipeline que tingui un vertex shader i un fragment shader enllaçats. Per això haurem de crear unes estructures anomenades *VkPipelineShaderStageCreateInfo*, que contenen dos camps importants: un *stage* que hem d'especificar a partir d'uns flags i un *module* de tipus *VkShaderModule* el qual és el codi compilat en SPIR-V dels shaders en qüestió (la *VulkanBase* s'encarrega de carregar aquests arxius binaris com a *VkShaderModule*). Aquests arxius binaris SPIR-V s'obtenen compilant els shaders amb una eina de la SDK de Vulkan. El pseudo-codi corresponent està present en la figura 19.

La pipeline també defineix les dades d'entrada. Com que es fan servir dos buffers, un pels vèrtex del model i un altre per les posicions dels planetes, haurem de definir dues estructures *VkVertexInputBindingDescription*, cadascuna amb un número de *binding* diferents (que ens ajudarà a identificar el buffer en la pipeline i el shader) i amb un *inputRate*. Aquest *inputRate* podrà ser *VK_VERTEX_INPUT_RATE_VERTEX* si parlem a nivell de vèrtex o *VK_VERTEX_INPUT_RATE_INSTANCE* si parlem a nivell de planeta. La diferència resideix en com el shader intpreta les dades, si com a un vèrtex o com una propietat d'instància (igual per tots els vèrtex d'un model). El pseudo-codi corresponent és el de la figura 20.

Hem definit les variables d'entrada per la pipeline, és a dir, els buffers. Ara hem de separar les dades dels buffers en les diferents variables d'entrada per a que el vertex shader les pugui interpretar. Per això s'utilitza l'estructura *VkVertexInputAttributeDescription*. Per cada camp del buffer que volguem utilitzar haurem de crear un 'Attribute-Description', especificant el *binding* corresponent al buffer, el *location* que haurà de coincidir amb el del shader, el *format* de les dades (*VK_FORMAT_R32G32B32_SFLOAT* per un vector float 3D) i l'*offset* que serveix per indicar en quina posició comença el camp especificat. A més, aquesta manera d'assignar els atributs dels shaders ens permet no haver d'assignar com a atributs tots els camps d'un buffer, és a dir, podem passar un buffer amb tres camps, però només fer servir els que siguin necessaris. El pseudo-codi de la figura 21 il·lustra la creació d'aquests atributs en Vulkan.

Cal aclarir que aquest és només un resum del codi original, que s'ha simplificat per explicar amb un exemple les diferents parts del codi.

3.1.3 Selecció dels recursos i dibuixat

Amb totes les estructures explicades fins ara som capaços de dibuixar un planeta en una posició específica (o com se sol anomenar, fer un *drawcall*). Això es fa fent un *binding* (enllaçat) de cada estructura: de la pipeline, del vertex buffer i de l'index buffer. Finalment, un cop hem seleccionat els nostres recursos podem fer servir una funció de Vulkan per dibuixar (*vkCmdDrawIndexed()*, la més estandaritzada és una implementació de *Instancing* [23]).

3.1.4 Bucle principal de l'aplicació

Abans d'entrar al bucle principal, el programa ha d'haver creat i inicialitzat tot el context de Vulkan. A això també s'inclou la creació de la nostra pipeline i els nostres buffers. Un cop fet això s'entra al bucle principal, el qual gestiona l'actualització de dades i després dibuixa continuament. Aquest bucle té els passos:

1. Inicialitzar Vulkan i recursos necessaris.
2. Entrar al bucle:
 - Actualitzar dades, com la càmera o variables de la UI (a partir de les funcions *Read()* o *Update()* dels buffers corresponents).
 - Escollir buffers d'un cos i dibuixar (a partir de les funcions d'enllaç de recursos de Vulkan, com *vkCmdBindPipeline()*).

3.2 Textures

Relacionat amb la representació de l'escena, passem a parlar de les textures. Hem implementat dos tipus: les textures convencionals, consistents d'una imatge, i les textures modificables a partir de soroll.

3.2.1 Textures dels planetes

Volem que cada planeta tingui una textura diferent, és a dir, volem tenir un objecte que representi la Terra i un altre la Lluna. Per aconseguir això podem fer servir diferents enfoc. El que s'aplica en aquest treball és el de fer servir un *Texture Atlas* (figura 6), una única imatge que conté diverses textures. Això ens permet carregar una única imatge a la *VulkanBase*, i jugar amb les coordenades textura de cada vèrtex per definir la textura que volguem. Tot i que això pot semblar una bona opció, estaria bé comparar-ho amb l'enfoc més tradicional (una textura, una imatge). Durant l'aplicació d'aquesta solució ens hem trobat amb un problema, i és que les imatges carregades per Vulkan no poden superar una certa mida de pixels. Per tant, això és un punt important a tenir en compte si volguéssim tenir encara més textures en la nostra imatge. Un altre aspecte a tenir en compte és la resolució de les textures, doncs si no tenen la mateixa, la selecció de la textura es podria complicar. Les textures han sigut obtingudes de [34].

3.2.2 Textures modificables a partir de soroll

Com a funcionalitat extra del treball s'ha implementat una modificació de textures a partir de soroll.

Per fer-ho, hem afegit una nova estructura de dades anomenada *BodyGraphics* dins de la classe dels planetes. *BodyGraphics* està format per un seguit de paràmetres que serveixen per configurar una funció de soroll i els colors que es faran servir per generar la textura. Aquests paràmetres es passen al fragment shader que ja teníem, a partir d'un nou buffer. Com s'ha comentat al desenvolupament, els passos a seguir serien:

1. Crear buffer.
2. Definir-lo dins la pipeline.

3. Afegir els atributs que es faran servir (ja que primer els passarem al vertex shader i després aquest els passarà al fragment shader).
4. Fer el *binding* d'aquest nou buffer a l'hora de dibuixar.

Per generar aquestes textures es fa servir una funció anomenada *fbm* (*Fractional Brownian Motion*, d'en Iñigo Quilez [39]). A la figura 22 es mostra el codi en llenguatge GLSL.

Aquesta funció combina diverses resolucions d'una funció de soroll, utilitzant *vertFrequency* i *vertAmplitude* per definir l'augment de resolució i *vertOctaves* per definir el número de resolucions diferents que es combinen. La funció de soroll s'ha de poder aplicar en superfícies esfèriques, doncs volem mantenir una continuïtat en la textura en les longituds i latituds. Això s'aconsegueix amb una funció de soroll 3D. El codi de la figura 23 mostra la funció del *3D Value Noise*, també d'en Iñigo Quilez [40].

Finalment, aquesta funció de soroll està basada en una altra funció aleatòria. Aquesta aleatorietat s'aconsegueix a partir d'una funció sinusoidal i està basada en repetir aquesta ondulació a partir del paràmetre *vertRandomSeed*. Quan més gran sigui aquest paràmetre menys reconeixible serà el patró del sinus. A més, se li afegeix un altre paràmetre *vertRandomWave* per corbar aquest patró i que encara sigui més difícil d'identificar, donant la sensació d'aleatorietat. La funció *random* es mostra a la figura 24 (es pot accedir a [41] per provar o trobar altres funcions).

3.3 El Skybox

Per acabar amb la part gràfica de la representació de l'escena parlarem del *Skybox*, el fons invariable que es crea i dibuixa d'una manera semblant a la dels planetes.

La principal diferència en la creació és que aquest fa servir un model diferent, creat amb blender, que consisteix en una esfera amb les coordenades textura mapejades cap a l'interior d'aquest. Aquest model està sempre situat en la mateixa posició que la càmera, així dona l'efecte de que l'escena té un fons invariable i llunyà, com el de l'espai. Per part de la textura, aquí sí que s'aplica una lògica "d'una imatge, una textura".

La diferència principal en el dibuixat del *Skybox* respecte als planetes és que fa servir una altra pipeline gràfica, en que no involucra cap mena de físiques (*BodyPhysics*).

3.4 Interfície i càmera

A continuació s'explica com s'actualitzen les dades de l'escena des de la interfície o fent servir la càmera.

3.4.1 Interfície d'ImGUI

Per la interfície, tenim un sistema en el que podem seleccionar un cos i s'obre un sub-menú per modificar les seves característiques des d'ImGUI (físiques, textura, dimensions, etc). El que realment estem fent és modificar les dades d'aquest cos, és a dir, en la CPU. Per tant, el següent pas és actualitzar els buffers corresponents amb aquestes noves dades modificades. Per a que quedi clar, si volem fer servir aquests buffers en la pipeline anterior haurem de afegir el

codi corresponent, com és el cas de les textures. D'aquesta manera aconseguim traslladar les dades de la CPU a la GPU i deixar-les llestes per la interpretació dels shaders. Si no hi ha cap modificació aquest pas no es fa, doncs no és necessari sobreescrivir les dades.

3.4.2 Classe càmera

Per part de la càmera és bastant més complexa, doncs els canvis es fan des del teclat o el ratolí, i en comptes de modificar dades del cos es canvien dades internes de la càmera. Per això primer cal entendre com funciona el nostre model de càmera.

La classe d'aquesta càmera prové del tutorial de 'learnOpenGL' [29]. Consta de tres vectors 3D: la posició origen, un vector *front/direction* que indica cap a on apunta i un vector *up* que apunta cap a dalt. El vector *right* es pot calcular amb el *crossProduct* dels dos vectors anteriors, obtenint un vector tangencial (figura 7, tercera imatge). Aquesta classe *Camera* conté una funció *UpdateCam()* que es crida a la *VulkanBase* abans de dibuixar el frame actual, per obtenir així la part de visió de la matriu MVP. En aquest cas, afegim a la pipeline un *UniformBuffer* que conté les matrius de visió i projecció. *UpdateCam()* crida les diferents funcionalitats depenent del mode en el que està la càmera.

A continuació s'explica com es modifiquen els paràmetres de la càmera pel mode de càmera lliure a partir d'alguns càlculs. Cal entendre primer que amb el "cross product" s'obté el vector tangencial de dos vectors i que la normalització serveix per obtenir un vector unitari.

- Avançar/retrocedir (W/S): en aquest moviment es busca modificar la posició de la càmera, seguint la direcció cap a la que s'està mirant o en sentit invers. Això no es pot fer simplement amb el vector *front* de la càmera, ja que aquesta direcció pot dependre dels tres eixos, i nosaltres només volem moure'ns en el pla X-Z. Per això es busca el vector tangencial *right* i tornem a crear el vector *front* com la tangencial de *right* i *up*, eliminant la influència de l'eix Y de *front*.

```
vec3 right = norm(crossProd(front,up);
position = position +/- norm(crossProd(right,up));
```

- Moviment lateral (A/D): per aquest moviment també es busca modificar la posició de la càmera, però aquest cop podrem fer-ho segons el vector tangencial *right*, calculat a partir de *front* i *up*.

```
position = position +/- norm(crossProd(front,up));
```

- Moviment en l'eix Y (Space/Shift): per aquest moviment també busquem modificar la posició, en aquest cas només sumant o restant a la component Y.

```
position = position +/- vec3(0,1,0);
```

- *LookAround()* (Ratolí): en aquest moviment ja no es tem parlant de canviar la posició de la càmera, sinó el seu vector *front*, és a dir, cap a on mirem. Això ho farem a partir de capturar continuament la posició del ratolí en pantalla. Si tenim les coordenades del frame anterior podrem obtenir l'*offset* per cada component 2D. Podem sumar aquest desplaçament a un sistema d'angles, *yaw* i *pitch*, pertanyents als angles d'Euler [30].

El *yaw* ens indica la inclinació o rotació de la vista respecte l'eix Y. El *pitch* ens indica la inclinació o rotació de la vista respecte el pla format per X i Z. D'acord amb això i ajudant-nos de les funcions de *cos()* i *sin()* podem obtenir el vector *front* modificat pel moviment del ratolí. Es pot observar com en les coordenades X i Z participen els dos angles d'Euler, mentre que per la coordenada Y només participa el *pitch*. Això és degut a les influències de cada angle, que es poden veure a la figura 8.

```
// Offset
offsetX = curr_screenX - prev_screenX;
offsetY = prev_screenY - curr_screenY;
// Recuperem posició actual
prev_screenX = curr_screenX;
prev_screenY = curr_screenY;
// Calculem yaw i pitch amb offset
yaw = yaw + offsetX;
pitch = pitch + offsetY;
front = norm(vec3(cos(yaw) * cos(pitch),
                 sin(pitch),
                 sin(yaw) * cos(pitch)));
```

3.4.3 Selecció de cossos

La selecció de cossos es fa servir per clicar sobre un cos representat per pantalla. Aquesta selecció provoca una aturada de la simulació i, a més, s'obre el menú del cos en qüestió, per a poder definir les seves propietats.

Per solucionar aquesta funcionalitat hem utilitzat l'algorisme més senzill i òptim en el nostre cas que és el RayCasting [35], doncs només necessitem generar una línia des de l'origen de la càmera fins a una distància determinada. Un punt d'aquesta recta serà l'origen de la càmera, però l'altre punt l'haurem de calcular a partir de la posició de pantalla que haguem clicat. Dit d'una altra manera, haurem de fer els càlculs en sentit contrari de quan renderitzem figures per pantalla, ja que haurem de passar de coordenades pantalla a coordenades món. Per això es fan servir una sèrie de passos [36], mostrats en el codi següent:

```
// Obtenir coordenades pantalla d'on s'ha clicat.
xpos = mouse.x; ypos = mouse.y;
// Obtenir coordenades 3D normalitzades segons la pantalla.
nx = (2 * xpos) / WIDTH - 1;
ny = 1 - (2 * ypos) / HEIGHT;
ray_screen = (nx, ny, 1);
// Obtenir coordenades 4D homogenies i 'clipejades'.
ray_clip = (ray_screen.x, ray_screen.y, -1, 1);
// Obtenir coordenades 4D càmera.
ray_cam = inv(projection) * ray_clip;
ray_cam = (ray_cam.x, ray_cam.y, -1, 1);
// Obtenir coordenades 3D món normalitzades.
ray_world = inv(view) * ray_cam;
ray_world = norm(ray_world);
```

Amb aquestes coordenades món normalitzades tenim un vector que apunta des de l'origen de càmera cap a la direcció que hem clicat. D'aquesta manera podem calcular si la distància entre aquest vector i el centre de cada cos és igual o menor que el radi d'aquest, i així seleccionar-lo 9. Per cada cos es detecta si hi ha col·lisió del raig de la forma següent:

1. Crear vector 'origen càmera - centre cos'.

```
camToBody = body.position - cam.position;
```

2. Calcular dot product [37] 'càmera - cos' i "raig de Ray-casting".

```
dp = dot(camToBody, ray_world);
```

3. Calcular producte escalar (longitud del vector 'càmera - cos' projectat en el 'raig de Raycasting').

```
scalar = dp / length(ray_world);
```

4. Calcular coordenades del punt projectat, a partir de l'origen de càmera, el raig i el producte escalar.

```
projected = cam.position + ray_world * scalar;
```

5. Calcular la distància entre el punt projectat i el centre del cos.

```
dist = distance(body.position, projected);
if (dist < body.radius)
    selected = true;
```

3.4.4 Càmera esfèrica

S'ha creat un altre mode de visualització de la càmera, amb el que ara l'acció d'avançar/retrocedir (W/S) serveix per fer zoom sobre un planeta seleccionat. Això, a més, provoca que sempre que hi hagi un planeta seleccionat, el vector *front* haurà d'estar apuntant cap a aquest, cosa que s'aconsegueix restant la posició del cos a la posició de la càmera.

La funció *LookSpherical()* s'assembla a la *LookAround()*, però en comptes de modificar el vector *front*, modifica els vectors *position* i *up*. Aquesta modificació es fa a partir dels mateixos angles *yaw* i *pitch*, però a més, per canviar la posició es fan servir dos paràmetres més: el *zoom* controlat per W/S i el *target* o cos seleccionat. Podem veure el càlcul en el pseudo-codi següent:

```
// Offset
offsetX = curr_screenX - prev_screenX;
offsetY = prev_screenY - curr_screenY;
// Recuperem posició actual
prev_screenX = curr_screenX;
prev_screenY = curr_screenY;
// Calculem yaw i pitch amb offset
yaw = yaw + offsetX;
pitch = pitch + offsetY;
// Modifiquem position i up
pos = vec3(zoom * cos(yaw) * cos(pitch) + target.pos.x,
           zoom * sin(pitch) + target.pos.y,
           zoom * sin(yaw) * cos(pitch) + target.pos.z);
up = vec3(-cos(yaw) * sin(pitch),
           cos(pitch),
           -sin(yaw) * sin(pitch));
```

A la figura 10 tota l'esfera simbolitza les possibles posicions de la càmera (sense zoom) i els angles *theta* (blau cel) i *phi* (rosa) simbolitzen els angles *yaw* i *pitch*, que fem servir per obtenir les diferents posicions de la càmera esfèrica. L'efecte que té el zoom és el de fer més gran o més petita aquesta esfera. En canvi, al canviar d'objecte seleccionat estarem movent aquesta esfera.

3.5 Còmput de les físiques

A continuació s'explica la implementació de les físiques per la simulació i visualització d'òrbites.

3.5.1 Simulació de físiques

Igual que per dibuixar gràfics, per la simulació de físiques també cal crear una pipeline. En aquest cas, farem servir un compute shader, un tipus de shader específic per aprofitar la gràfica per càlcul intensiu no gràfic. Aquest tipus de shader forma part d'una línia diferent de la resta. Per tant s'ha de crear una pipeline diferent i separada de la resta de la part

gràfica (observi's en la figura 5 que el compute shader està separat de la resta, a la dreta).

Els buffers que ens permeten gestionar múltiples dades en un compute shader s'anomenen *StorageBuffers*. Com es pot veure al centre de la figura 5 aquest buffers s'han d'utilitzar a partir d'un *DescriptorSet*. Aquesta estructura de Vulkan té l'objectiu de connectar dades de CPU a la GPU (els *VertexBuffers* i *IndexBuffers* són un cas apart, que no necessiten d'aquesta estructura per poder comunicar-se amb la GPU). Per tant, la definició d'una pipeline de còmput està centrada en crear els *DescriptorSets* pels *StorageBuffers* (o *UniformBuffers*, que es poden utilitzar en la majoria de shaders i també necessiten un *DescriptorSet*), en comptes de definir els atributs d'entrada com es fa amb el vertex shader.

L'aplicació consta d'un *UniformBuffer* (un buffer amb la flag *VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT*) per guardar el pas de temps constant i el número actual de cossos i dos *StorageBuffers* (un buffer amb els flags *VK_BUFFER_USAGE_STORAGE_BUFFER_BIT* i *VK_BUFFER_USAGE_VERTEX_BUFFER_BIT*, aquest segon necessari, doncs passarem la posició de les físiques també al vertex shader). Fem servir dos buffers degut a que per calcular la següent posició dels cossos necessitem les dades del frame anterior. El diagrama de VulkanTutorial [24] (figura 11) representa bastant bé el nostre objectiu, on es veu que l'ús dels dos buffers es va intercalant per cada frame). Per crear un *DescriptorSet* primer hem de definir el *binding* i el tipus de buffer a partir de l'estructura *VkDescriptorSetLayoutBinding* (pseudo-codi a la figura 25).

Aquests *StorageBuffers* estan formats per elements de tipus *BodyPhysics*, un atribut propi de la classe *Planetes*, el qual conté totes les dades per la simulació física d'un planeta: la massa, la velocitat i la posició.

Després hem de definir el *VkDescriptorPoolSize*, que indica a Vulkan quants *DescriptorSets* s'esperen. Observem que definim el *count* com 2. Això és degut a que dupliquem els recursos, per a que mentre un frame es dibuixa el següent ja s'estigui preparant. En aquest cas fem servir dues imatges o frames, però res ens limita de poder fer-ne servir més. Podem observar el pseudo-codi de la figura 26 per entendre la nostra implementació.

El següent pas es guardar tota aquesta informació dins d'un *VkDescriptorSet* (en el nostre cas 2, un per cada frame). Per sintetitzar-ho, imaginem que un array de dos *VkDescriptorSets* anomenat 'descriptorSet' conté tota aquesta informació. L'últim que ens queda és assignar els nostres buffers (recordem que és la estructura que realment conté les dades) al nostre 'descriptorSet'. Això ho podem fer a partir de l'estructura *VkDescriptorBufferInfo* per emmagatzemar la informació dels buffers i *VkWriteDescriptorSet* per assignar aquesta informació al 'descriptorSet' a partir dels binding prèviament definits (pseudo-codi a la figura 27)

Al codi veiem que fem servir un bucle *for* de dues iteracions, doncs volem fer aquest emmagatzematge (en *VkDescriptorBufferInfo*) i assignació (en *VkWriteDescriptorSet*) per cada frame, i en el nostre cas hem dit que fèiem servir dos frames per mantenir el hardware ocupat.

Amb tot això ja podem procedir amb la trucada de la pipeline de còmput i els *descriptors* adients. En el cas específic dels compute shaders no fem crides mitjançant *drawcalls*, sinó *dispatches*. Aquests *dispatches* es fan en

grups d'una mida de tres dimensions. Cada grup s'anomena *workgroup* i estarà format per *invocations*, també organitzats en tres dimensions. A l'esquerra de la figura 12 es pot veure la representació en tres dimensions dels *workgroups* dels que es fa *dispatch*. A la dreta veiem una representació de tres dimensions de les invocacions (els *threads* en computació paral·lela) de cada *workgroup*. El diagrama separa cada *workgroup*, ja que això simbolitza com s'executen les invocacions o *threads*, i que si aquests pertanyen al mateix *workgroup* comparteixen espai de memòria. Podem aprofitar la configuració de les dimensions dels *workgroups* i *invocations* per resoldre alguns problemes o aplicar algorismes paral·lels.

Finalment, tenint la part de la pipeline només queda veure com s'executa dins del compute shader. Hem de pensar que el pseudo-codi de la figura 28 s'executa per a cada planeta present en el sistema, ja que els compute shaders funcionen com qualsevol altre mètode de paral·lelització.

3.5.2 Representació d'òrbites

Aprofitant la capacitat de còmput de Vulkan es pot calcular un sistema d'òrbites, ja que hem d'obtenir les posicions futures d'un cos, no només en un frame, sinó en múltiples.

Per implementar aquesta funcionalitat es fa servir una altra pipeline de còmput amb un nou compute shader. Aquest té com a entrada les propietats físiques dels cossos (les mateixes que per la simulació, les de *BodyPhysics*) i dos nous paràmetres, els *steps* i el *delta*:

- Els *steps* són la quantitat de posicions futures que volem obtenir per representar la nostra òrbita. En altres paraules, són la quantitat de vèrtex/posicions que es retornaran.
- El *delta* és un paràmetre que controla cada quan es calculen aquestes futures posicions. En altres paraules, controlarà la precisió de la representació de l'òrbita.

Una peculiaritat de la forma en que es calculen aquestes òrbites és que, per cada posició següent d'un cos hauréu d'actualitzar la posició següent de la resta, doncs sempre voldrem calcular la influència del frame corresponent i no de l'inicial. Per tant el procediment per cada *step* es mostra en les figures 13 i 14 amb diferents valors de *steps* i *delta*.

A les imatges anteriors es veu la diferència de fer servir un *delta* més gran o més petit. Amb un *delta* petit es guanya precisió, però s'han d'augmentar els *steps* per dibuixar tot el recorregut de l'òrbita. En canvi, amb un *delta* gran ens estalviem dibuixar línies, però aquestes no defineixen tan bé el moviment futur del cos.

Podem veure el pseudo-codi que genera les òrbites en la figura 29.

4 RESULTATS I DEMOSTRACIONS

A continuació es mostren els resultats destacables obtinguts d'aquest treball.

1. Sistema solar realista.
2. Creació i modificació de textures.
3. Interfície simple i intuïtiva.

4.1 Sistema solar realista

Un dels objectius principals del treball era representar el sistema solar, per tal de validar la correctesa de la simulació.

A partir de codi hem creat el Sol, els vuit planetes principals, la Lluna i el cometa Halley. Això ho hem fet utilitzant les dades del dia 1 de Gener del 2026, obtingudes de la pàgina de *horizons* de la NASA [31]. Aquestes dades són de magnituds enormes (observi's les masses o les distàncies), per això hem aplicat factors d'escala als diferents paràmetres, sense perdre la forma de les òrbites reals:

- Factor de massa = $1:10e-24$.
- Factor de distància = $1:10e-7$.
- Factor de velocitat = $1:1$.

La figura 15 mostra la vista del sistema solar des de la Terra, on es visible la Lluna i el Sol i en la figura 16 podem veure totes les òrbites del sistema solar i del cometa Halley.

Per a validar aquestes posicions representades per les òrbites (línies blanques) hem creat una escena amb el Sol i la Terra. La Terra està en la seva posició del 9 de Juny del 2026, segons les dades de la NASA. Llavors el que hem fet ha sigut crear un altre cos de color blanc i posicionar-lo on la NASA diu que estarà la Terra el dia 9 de Decembre. D'aquesta manera, i com es veu a la figura 17, podem dir que la simulació i traçat de l'òrbita és correcta, doncs passa just per la posició corresponent al 9 de Decembre.

4.2 Creació i modificació de textures

Amb la funcionalitat de modificació de textures a partir de soroll es poden aconseguir aspectes ficticis, jugant amb els diferents paràmetres. També, es pot escollir d'activar-los o desactivar-los mitjançant la UI. Un exemple del que es pot aconseguir amb això es pot observar a la figura 18.

4.3 Interfície simple i intuïtiva

Un altre punt fort d'aquest treball es la capacitat de parametritzar qualsevol variable. Un exemple d'això són els diferents menús presents en l'aplicació. Permeten a l'usuari configurar lliurement la seva escena aconseguint allò que hem parlat a la introducció: que es puguin fer representacions realistes, però també fictícies o provar realitats alternatives.

Alguns exemples d'aquests menús es poden veure en les figures anteriors:

- A les figures 13 i 14, a l'esquerra s'observa el menú principal amb la secció de configuracions globals oberta (com es pot veure, serveix per provocar canvis en tots els cossos presents en la escena).
- A la figura 15, a l'esquerra s'observa el menú principal amb la secció de configuració de càmera oberta.
- A la figura 16, a l'esquerra a baix de tot del menú principal hi ha 'checkboxes' per simular l'escena i per activar/desactivar la visualització d'òrbites.
- A la figura 17, a l'esquerra es pot veure la secció de la llista de cossos de l'escena del menú principal.

- A la figura 18, abaix a l'esquerra s'observa el submenú del cos amb la secció de gràfics oberta.

A més, en totes aquestes captures es veu una finestra d'ImGUI addicional (adalt a l'esquerra) que mostra els recursos consumits per ImGUI així com els fotogrames per segons de l'aplicació i de la simulació.

5 CONCLUSIONS

Com a conclusions cal destacar:

- S'ha pogut aconseguir una aplicació apta per configurar escenes amb dades realistes de l'espai i que permet visualitzar simulacions en temps real, sense espere massa llargues. Això ha sigut possible gràcies a fer servir la GPU tant per càlcul intensiu com pel processament dels fotogrames, fent servir Vulkan com a columna vertebral del software.
- Utilitzar Vulkan complica el desenvolupament en primera instància, doncs és una API gràfica molt més densa que OpenGL. Però és aquesta densitat el que permet generar aplicacions més costumitzables i aptes per l'execució en una gran varietat de dispositius, ja siguin actuals o més antics.
- Hem après a gestionar pipelines, tant de gràfics com de còmput, així com crear, llegir i actualitzar les dades dels diferents buffers utilitzats, coneixement el qual ens permet crear entorns com el que s'ha presentat en aquest treball. Per exemple, els flags de Vulkan són molt pràctics a l'hora de definir buffers, doncs si t'equivocoques en el seu ús, les *Validation Layers* (una eina de depuració que ofereix la mateixa API de Vulkan) de la *VulkanBase* t'indiquen l'error.
- Finalment, crec que es pot treure molt profit d'aquest treball, doncs no només s'ha creat una aplicació, sinó que s'ha formalitzat el seu desenvolupament. Crec que el resultat final podria ser una molt bona base per un entorn encara més ampli i parametritzable o, fins i tot, per un videojoc de l'espai. No descarto la possibilitat de seguir el desenvolupament de l'aplicació, doncs ha sigut un treball molt apassionant el crear escenaris dels cossos que ens envolten, però potser mai arribarem a veure d'aprof (o de cossos que no existeixen).

Com a millores destaquem:

- Per l'optimització de les físiques podem utilitzar l'algorisme de Barnes-Hut [32]. Amb aquest algorisme s'evita fer una avaluació N-N dels cossos de l'escena per calcular les forces actuant entre ells. Això és degut a que l'algorisme interpreta a cossos molt propers com un de sol, evitant així moltes comparacions.
- Una altra millora està en el càlcul de les òrbites, que està implementat com un bucle for dins d'un compute shader. Això, amb uns *steps* i *deltaTime* molt grans provoca que el càlcul s'alenteixi, i per algunes GPUs pot, fins i tot, aturar l'aplicació. Per això crec que aquest aspecte s'ha de tenir en compte per optimitzar el màxim possible, canviant l'algorisme (per exemple, limitant els recursos gestionats per una invocació).

- Sovint ens podem trobar que podem solucionar un problema de moltes maneres diferents amb Vulkan. Un d'aquests cassos és la implementació de textures per *Instancing*, que es pot efectuar a partir de *Descriptor-Indexing* [33]. Amb això tindriem l'oportunitat de dibuixar molts cossos a la vegada, però amb textures diferents.

ÚS DE LA IA GENERATIVA

L'ús de la IA generativa en aquest treball s'ha limitat a la recerca de fonts sobre conceptes molt concrets. Un exemple és la integració d'Euler, la qual és un aspecte que moltes vegades es dona per suposat en entorns com el desenvolupament, però és més difícil de trobar explicacions concretes. És a dir, l'ús ha estat el de buscador d'ús puntual més que el de consultora. En cap moment s'ha utilitzat per trobar la solució a problemes, generar codi o redactar, doncs considero que el context d'aquest treball és massa extens i se'n pot treure molt profit de trobar les solucions per un mateix.

AGRAÏMENTS

En primer lloc m'agradaria agrair al tutor d'aquest TFG, l'Enric Martí Gòdia, pel seu esforç fonamental en aquest treball, ja que m'ha ajudat a formalitzar-lo i a prendre les millors decisions amb el seu entusiasme i la seva disposició.

Un altra persona fonamental en el desenvolupament d'aquest treball ha sigut en Raúl Velázquez Gómez, ja que sense la seva base de Vulkan aquest treball hagués sigut molt més complicat.

També vull aprofitar per donar les gràcies al meu pare i la meva mare per deixar-me expressar lliurement sempre que m'ha fet falta i per confiar en mi, a més de que he compartit amb molta il·lusió tot el treball que he aconseguit.

Finalment, vull agrair al meu germà, doncs entre altres coses és el meu referent acadèmic i els seus consells sempre m'han sigut d'ajuda.

REFERÈNCIES

- [1] Khronos Group, *Vulkan*. <https://www.vulkan.org/> (data últim accés: maig 2026).
- [2] Khronos Group, *OpenGL - The Industry Standard for High Performance Graphics*. <https://www.opengl.org/> (data últim accés: abril 2026).
- [3] Mobius Digital Games, *Outer Wilds official website*. <https://www.mobiusdigitalgames.com/outerwilds.html> (data últim accés: març 2026).
- [4] Unity Technologies, *Unity*. <https://unity.com/es> (data últim accés: juny 2026).
- [5] S. Radivoev, *Planetary System github repository*. <https://github.com/simeonradivoev/PlanetarySystem> (data últim accés: maig 2026).
- [6] Wikipedia Community, *Newton's law of universal gravitation*. https://en.wikipedia.org/wiki/Newton's_law_of_universal_gravitation (data últim accés: maig 2026).

- [7] Microsoft, *Visual Studio*. <https://visualstudio.microsoft.com/es/> (data últim accés: març 2026).
- [8] Microsoft, *Github*. <https://github.com/> (data últim accés: maig 2026).
- [9] J. Perkins, *Premake*. <https://premake.github.io/> (data últim accés: març 2026).
- [10] I. Quilez and P. Jeremias, *Shadertoy*. <https://www.shadertoy.com/> (data últim accés: març 2026).
- [11] Microsoft, *Visual Studio Code*. <https://code.visualstudio.com/> (data últim accés: març 2026).
- [12] Blender Foundation, *Blender*. <https://www.blender.org/> (data últim accés: març 2026).
- [13] Google, *Google Sheets workspace*. <https://workspace.google.com/intl/es/products/sheets/> (data últim accés: maig 2026).
- [14] M. Perkins, C. Adams and C. Obrecht, *Canva*. https://www.canva.com/es_es/ (data últim accés: maig 2026).
- [15] M. Geelnard and C. Löwy, *GLFW*. <https://www.glfw.org/> (data últim accés: març 2026).
- [16] G-Truc Creation, *GLM github repository*. <https://github.com/g-truc/glm> (data últim accés: març 2026).
- [17] Ocornut, *ImGui github repository*. <https://github.com/ocornut/imgui> (data últim accés: març 2026).
- [18] A. Gessler, T. Schulze and K. Kulling, *ASSIMP github repository*. <https://github.com/assimp/assimp> (data últim accés: març 2026).
- [19] Atlassian, *Kanban*. <https://www.atlassian.com/es/agile/kanban> (data últim accés: març 2026).
- [20] N. Elisa Chisari and M. Zaldarriaga, *Connection between Newtonian simulations and general relativity*. <https://journals.aps.org/prd/abstract/10.1103/PhysRevD.83.123505> (data últim accés: abril 2026).
- [21] Wikipedia Community, *General relativity*. https://en.wikipedia.org/wiki/General_relativity (data últim accés: maig 2026).
- [22] J. Díaz, *Ecuaciones diferenciales ordinarias*. https://www.uv.es/diazj/mne_tema1.pdf (data últim accés: juny 2026).
- [23] Wikipedia Community, *Geometry instancing*. https://en.wikipedia.org/wiki/Geometry_instancing (data últim accés: maig 2026).
- [24] A. Overvoorde, *Vulkan-tutorial, Compute shaders*. <https://vulkan-tutorial.com/Compute.Shader> (data últim accés: maig 2026).
- [25] R. Velázquez Gómez, *Raúl Velázquez Gómez github profile*. <https://github.com/Barometz1o> (data últim accés: maig 2026).
- [26] A. Overvoorde, *Vulkan-tutorial*. <https://vulkan-tutorial.com/Introduction> (data últim accés: maig 2026).
- [27] J. Santell, *Model View Projection*. <https://jsantell.com/model-view-projection/> (data últim accés: juny 2026).
- [28] E. Martí Gòdia, *Visualització Gràfica Interactiva (VGI)*. (data últim accés: juny 2026).
- [29] J. De Vries, *Learn OpenGL*. <https://learnopengl.com/> (data últim accés: maig 2026).
- [30] Wikipedia Community, *Euler angles*. https://en.wikipedia.org/wiki/Euler_angles (data últim accés: maig 2026).
- [31] NASA, *Horizons System*. <https://ssd.jpl.nasa.gov/horizons/app.html> (data últim accés: juny 2026).
- [32] Wikipedia Community, *Barnes-Hut simulation*. https://en.wikipedia.org/wiki/BarnesHut_simulation (data últim accés: juny 2026).
- [33] S. Dubois, *Vulkan Descriptors Indexing*. <https://gist.github.com/DethRaid> (data últim accés: juny 2026).
- [34] Solar System Scope, *Solar Textures*. <https://www.solarsystemscope.com/textures/> (data últim accés: maig 2026).
- [35] Wikipedia Community, *Ray casting*. <https://en.wikipedia.org/wiki/RayCasting> (data últim accés: maig 2026).
- [36] A. Gerdelan, *Mouse Picking with Ray Casting*. <https://antongerdelan.net/opengl/raycasting.html> (data últim accés: maig 2026).
- [37] G. Sanderson, K. Bruns and J. Schloss, *Dot products and duality*. <https://www.3blue1brown.com/lessons/dot-products> (data últim accés: maig 2026).
- [38] J. Prunier, *Scratchpixel*. <https://www.scratchapixel.com/> (data últim accés: juny 2026).
- [39] I. Quilez, *fBM - 2019*. <https://iquilezles.org/articles/fbm/> (data últim accés: juny 2026).
- [40] I. Quilez, *value noise derivatives - 2008*. <https://iquilezles.org/articles/morenoise/> (data últim accés: juny 2026).
- [41] P. Gonzalez Vivo and J. Lowe, *The Book of Shaders*. <https://thebookofshaders.com/10/> (data últim accés: juny 2026).

APÈNDIX

A.1 Figures

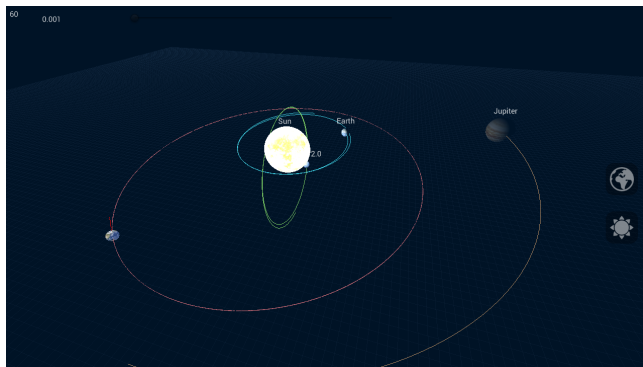


Fig. 1: Entorn gràfic del "Planetary System" de simeonradivoev.



Fig. 2: Exemple de Kanban en Google Sheets.

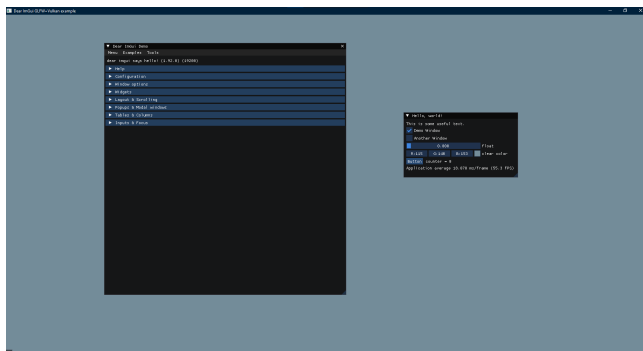


Fig. 3: Finestra de demostració d'una interfície d'ImGUI.

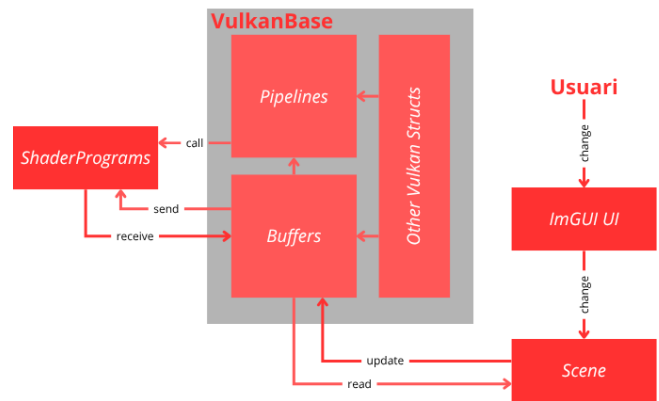


Fig. 4: Esquema general de l'aplicació, dividit en mòduls.

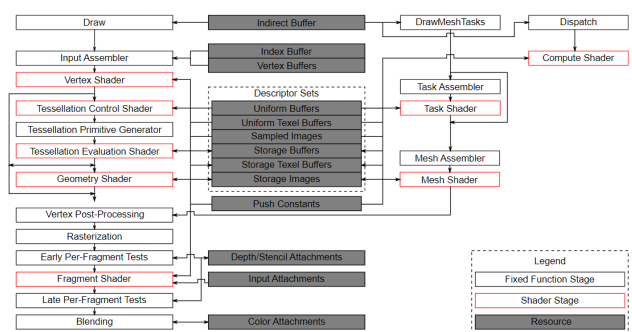


Fig. 5: Diagrama dels diferents tipus de Shaders i quines estructures es fan servir per comunicar-se entre elles.

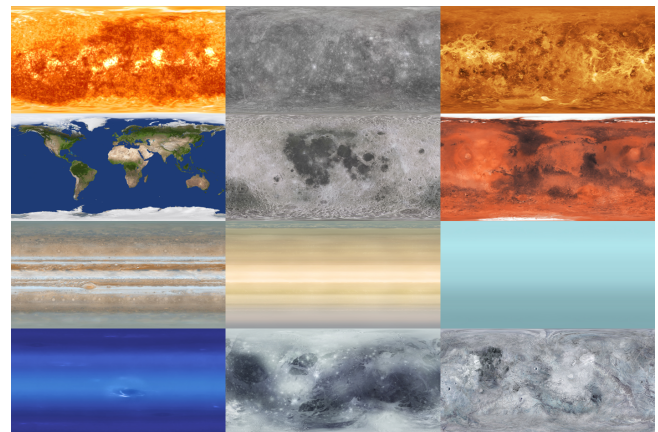


Fig. 6: Texture Atlas que es fa servir en l'aplicació.
D'esquerra a dreta i de dalt a baix: Sol, Mercuri, Venus, Terra, Lluna, Mart, Júpiter, Saturn, Urà, Neptú, Plutó, Eris.

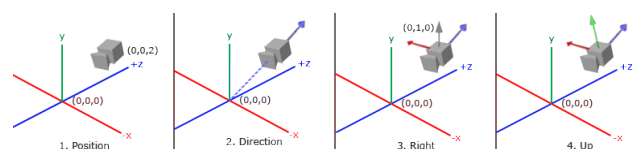


Fig. 7: Representació dels principals vectors de la càmera.

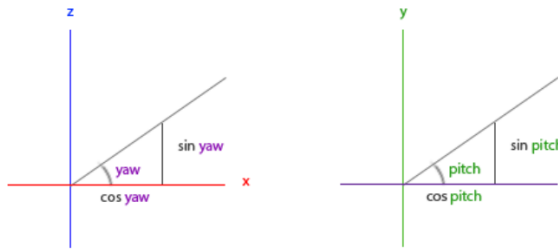


Fig. 8: Esquema dels angles d'Euler yaw (esquerra) i pitch (dreta).

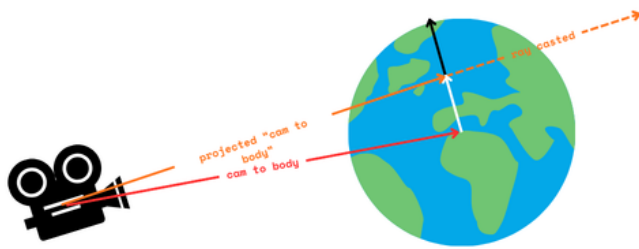


Fig. 9: Representació gràfica dels passos per determinar la selecció d'un cos. En aquest cas el cos se seleccionaria, doncs el vector blanc es menor que el negre.

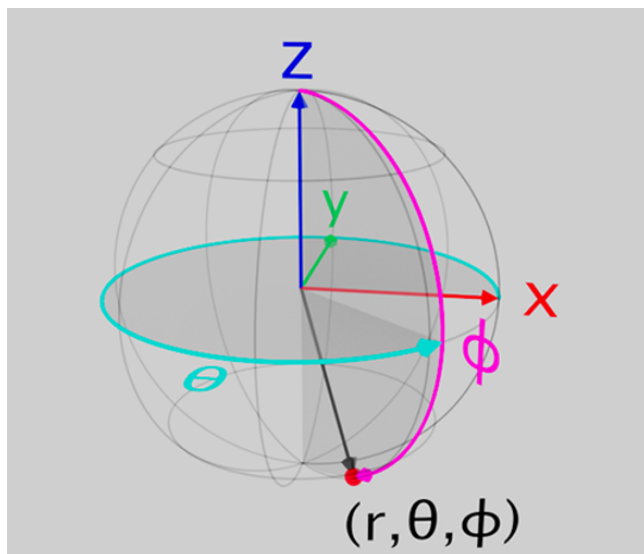


Fig. 10: Diagrama de *scratchpixel* [38] la càmera esfèrica, tenint en compte que el centre és l'objecte seleccionat.

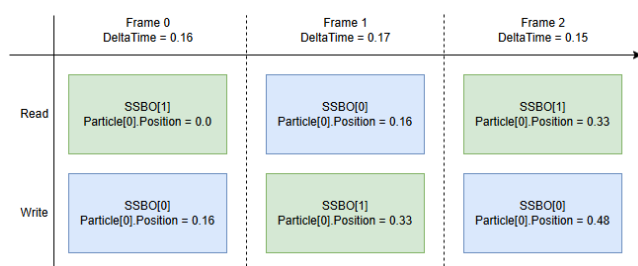


Fig. 11: Diagrama de l'ús de cada *StorageBuffer* per processar cada frame.

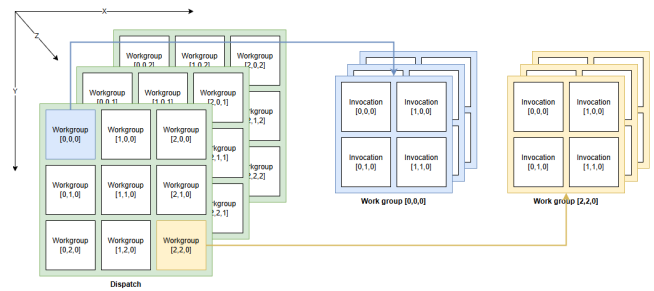


Fig. 12: Diagrama de *workgroups* i *invocations* dels compute shaders.

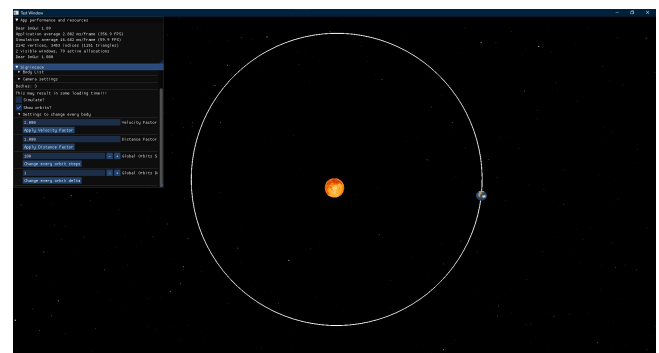


Fig. 13: Òrbita amb $\text{step}=100$, $\text{delta}=1$.

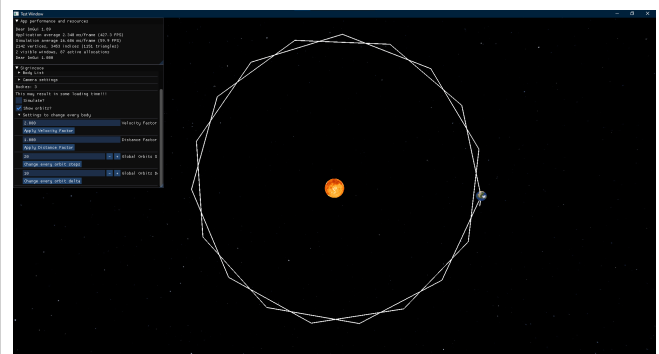


Fig. 14: Òrbita amb $\text{step}=20$, $\text{delta}=10$.

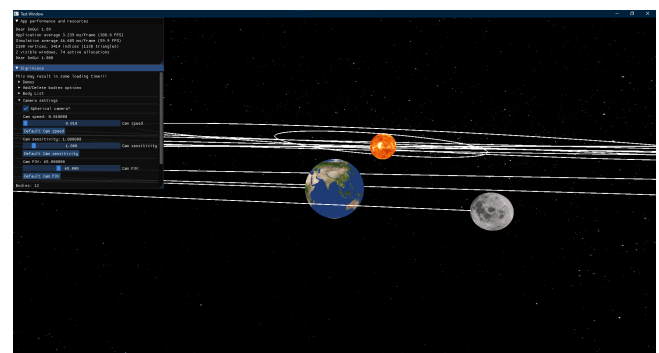


Fig. 15: Imatge de la escena de simulació del sistema solar, vist des de la Terra.

A.2 Codi

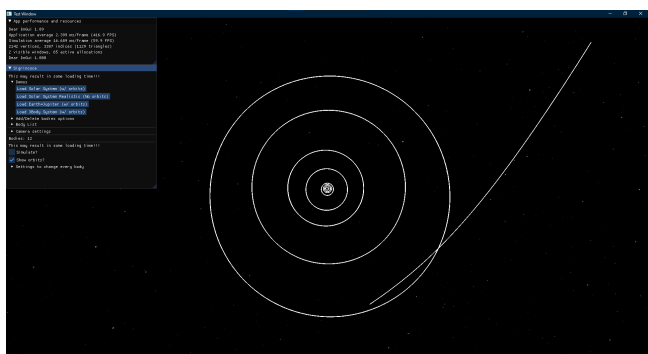


Fig. 16: Sistema solar vist des de dalt, amb la representació de totes les òrbites.

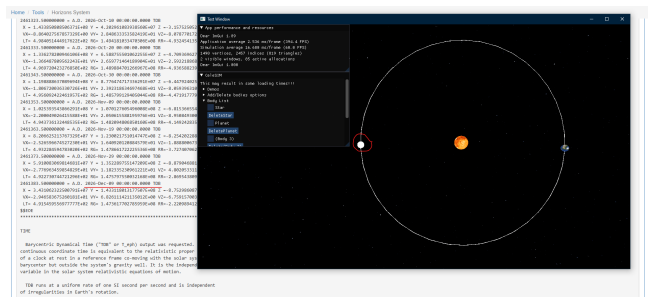


Fig. 17: Demostració de la correctesa de la simulació i representació, contrastat amb dades de la NASA.

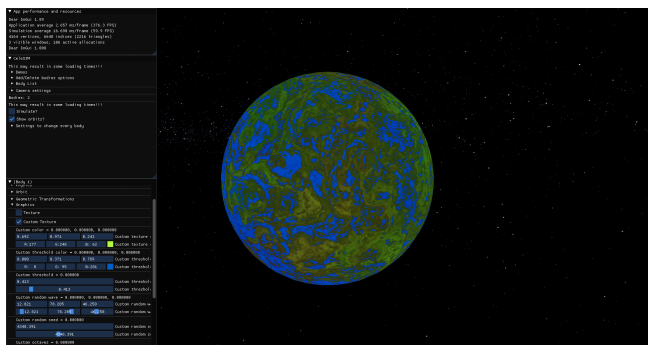


Fig. 18: Planeta creat amb un aspecte semblant a la Terra amb textura modificable a partir de soroll.

```
// Vertex shader stage
VkPipelineShaderStageCreateInfo vertexShaderInfo = {};
vertexShaderInfo.stage = VK_SHADER_STAGE_VERTEX_BIT;
vertexShaderInfo.module = vertexShaderModule;
// Fragment shader stage
VkPipelineShaderStageCreateInfo fragmentShaderInfo = {};
vertexShaderInfo.stage = VK_SHADER_STAGE_FRAGMENT_BIT;
vertexShaderInfo.module = fragmentShaderModule;
```

Fig. 19: Pseudo-codi per assignar els *ShaderModules* a una pipeline.

```
vector<VkVertexInputBindingDescription> bindings(2);
// Vertex buffer binding
bindings[0] = {};
bindings[0].binding = 0;
bindings[0].stride = sizeof(Vertex);
bindings[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
// Physics buffer binding
bindings[1] = {};
bindings[1].binding = 1;
bindings[1].stride = sizeof(BodyPhysics);
bindings[1].inputRate = VK_VERTEX_INPUT_RATE_INSTANCE;
```

Fig. 20: Pseudo-codi per definir els buffers a utilitzar en la pipeline.

```
vector<VkVertexInputAttributeDescription> attributes(2);
// Vertex coords Attribute
attributes[0].binding = 0;
attributes[0].location = 0;
attributes[0].format = VK_FORMAT_R32G32B32_SFLOAT;
attributes[0].offset = offsetof(Vertex, pos);
// Body position Attribute (Instance)
attributes[1].binding = 1;
attributes[1].location = 1;
attributes[1].format = VK_FORMAT_R32G32B32_SFLOAT;
attributes[1].offset = offsetof(BodyPhysics, position);
```

Fig. 21: Pseudo-codi per definir els atributs d'entrada als shaders a partir dels buffers de la pipeline.

```
float fbm (in vec3 st) {
    // Initial values
    float value = 0.0;
    float amplitude = vertInitAmplitude;
    float frequency = 0.;
    //
    // Loop of octaves
    for (int i = 0; i < vertOctaves; i++) {
        value += amplitude * noise(st*vertFrequency);
        st *= vertFrequency;
        amplitude *= vertAmplitude;
    }
    return value;
}
```

Fig. 22: Pseudo-codi de la funció FBM en llenguatge GLSL.

```
float noise (in vec3 st) {
    vec3 i = floor(st);
    vec3 fp = fract(st);

    float a = random( i+vec3(0,0,0) );
    float b = random( i+vec3(1,0,0) );
    float c = random( i+vec3(0,1,0) );
    float d = random( i+vec3(1,1,0) );
    float e = random( i+vec3(0,0,1) );
    float f = random( i+vec3(1,0,1) );
    float g = random( i+vec3(0,1,1) );
    float h = random( i+vec3(1,1,1) );

    float k0 = a;
    float k1 = b - a;
    float k2 = c - a;
    float k3 = e - a;
    float k4 = a - b - c + d;
    float k5 = a - c - e + g;
    float k6 = a - b - e + f;
    float k7 = - a + b + c - d + e - f - g + h;

    vec3 u = fp * fp * (3.0 - 2.0 * fp);

    return k0 + k1*u.x + k2*u.y + k3*u.z + k4*u.x*u.y +
           k5*u.y*u.z + k6*u.z*u.x + k7*u.x*u.y*u.z;
}
```

Fig. 23: Pseudo-codi de la funció noise en llenguatge GLSL.

```
float random (in vec3 st) {
    return fract(sin(dot(st.xyz,
                        vertRandomWave)) *
                vertRandomSeed);
}
```

Fig. 24: Pseudo-codi de la funció random en llenguatge GLSL.

```
std::vector<VkDescriptorSetLayoutBinding> bindings(3);
bindings[0] = {};
bindings[0].binding = 0;
bindings[0].descriptorType = VK_TYPE_UNIFORM_BUFFER;
bindings[1] = {};
bindings[1].binding = 1;
bindings[1].descriptorType = VK_TYPE_STORAGE_BUFFER;
bindings[2] = {};
bindings[2].binding = 2;
bindings[2].descriptorType = VK_TYPE_STORAGE_BUFFER;
```

Fig. 25: Pseudo-codi dels bindings dels buffers utilitzant *DescriptorSets*, per la pipeline de còmput.

```
std::vector<VkDescriptorPoolSize> poolSize(2);
poolSize[0] = {};
poolSize[0].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
// 2dos frames i un UniformBuffer
poolSize[0].count = 2 * 1;
poolSize[1] = {};
poolSize[1].type = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
// dos frames i dos StorageBuffers
poolSize[1].count = 2 * 2;
```

Fig. 26: Pseudo-codi de la creació del *PoolSize* dels *DescriptorSets* de còmput, per dos frames.

```
for (int i = 0; i < 2; i++)
    VkDescriptorBufferInfo uniformData = {};
    descriptorBuffersUni.buffer = uniBuffer.buffer[i];
    VkDescriptorBufferInfo frame0 = {};
    frame0.buffer = buffersBodyPhysics.buffer[(i-1)%2];
    VkDescriptorBufferInfo frame1 = {};
    frame1.buffer = buffersBodyPhysics.buffer[i];

    std::vector<VkWriteDescriptorSet> writeDS(3);
    writeDS[0].dstSet = descriptorSet[i];
    writeDS[0].dstBinding = 0;
    writeDS[0].descriptorType = VK_TYPE_UNIFORM_BUFFER;
    writeDS[0].pBufferInfo = &descriptorBuffersUni;
    writeDS[1].dstSet = descriptorSet[i];
    writeDS[1].dstBinding = 1;
    writeDS[1].descriptorType = VK_TYPE_STORAGE_BUFFER;
    writeDS[1].pBufferInfo = &frame0;
    writeDS[2].dstSet = descriptorSet[i];
    writeDS[2].dstBinding = 2;
    writeDS[2].descriptorType = VK_TYPE_STORAGE_BUFFER;
    writeDS[2].pBufferInfo = &frame1;
```

Fig. 27: Pseudo-codi per emmagatzemar les dades dels buffers (*DescriptorSet*) i assignar-les (*WriteDescriptorSet*).

```
bodyNext[index].velocity = bodyCurr[index].velocity;
for (int i = 0; i < ubo.numParticles; i++)
    if (i != index)
        // Calcul distància
        float d = distance(bodyNext[i].pos,
                           bodyCurr[index].pos);
        // Calcular força
        float F = G * bodyCurr[index].mass *
                particle2.mass/(d*d);
        // Calcular vector força
        vec4 FDir = normalize(bodyNext[i].pos-
                              bodyCurr[index].pos);
        // Calcular acceleració
        vec4 acc = F * FDir / bodyCurr[index].mass;
        // Calcular velocitat
        bodyNext[index].velocity += acc * ubo.deltaTime;
// Calcular posició següent
bodyNext[index].pos += bodyNext[index].velocity *
ubo.deltaTime;
```

Fig. 28: Pseudo-codi del compute shader de les físiques.

```
// Calcular velocitats de tots els cossos fent servir
l'equació de gravitació universal.
for (int i = 0; i < ubo.numParticles; i++)
    if (i != index)
        // Calcul distància
        float d = distance(bodyNext[i].pos,
                           bodyCurr[index].pos);
        // Calcular força
        float F = G * bodyCurr[index].mass *
                particle2.mass/(d*d);
        // Calcular vector força
        vec4 FDir = normalize(bodyNext[i].pos-
                              bodyCurr[index].pos);
        // Calcular acceleració
        vec4 acc = F * FDir / bodyCurr[index].mass;
        // Calcular velocitat
        bodyNext[index].velocity += acc
                * ubo.deltaTime;
// Calcular posició step actual per tots els cossos.
bodyNext[index].pos += bodyNext[index].velocity *
ubo.deltaTime;
// Retornar posició només si la distància entre la
última posició retornada i l'actual és de delta steps.
if (step % delta == 0)
    orbits.push_back(bodyCurr[i].position);
```

Fig. 29: Pseudo-codi del compute shader de les òrbites.