Universitat de Girona
Escola Politècnica Superior

Treball final de grau

Estudi: Grau en Enginyeria Informàtica

Títol: Motor gràfic: Planeta procedural

Document: Memòria Treball Final de Grau

Alumne: Xavier Avivar Rubio

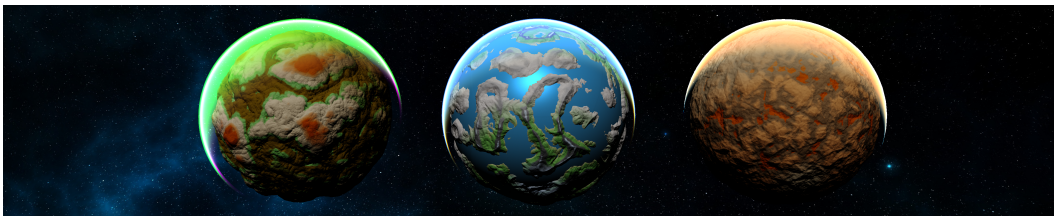
Tutor: Gustavo Patow
Departament: IMAE
Àrea: LSI

Convocatòria (mes/any): Febrer 2022

Universitat de Girona
Escola Politècnica Superior

PROJECTE FI DE GRAU

Motor gràfic: Planeta procedural



Autor:
Xavier AVIVAR RUBIO

Febrer 2022

Grau en Enginyeria Informàtica

Tutor:
Gustavo PATOW
Departament: IMAE
Àrea: LSI

Agraïments

Per començar vull agrair molt especialment a tots els meus companys que han estat allà fins al final, no ho podria haver aconseguit sense vosaltres. Agraïment a la meva família que m'ha mantingut durant gran part del proces. A la **EPS** i a tots els seus professors, que gracies a que som una universitat petita, podem tenir un tutelat-ge bastant mes personalitzat i personal amb el professorat...

Índex

CAPÍTOL 1

Introducció

Un motor gràfic és un **framework** que permet dissenyar videojocs. L'idea principal es oferir l'usuari una forma fàcil d'interactuar amb la targeta gràfica. Avui dia existeixen motors gràfics ja fets que proporcionen una capa d'abstracció considerable per l'usuari, com **Unreal Engine** o **Unity**, però això no sempre ha estat així. Antigament els desenvolupadors de videojocs havien de construir el seu propi motor. Això ens porta a l'objectiu d'aquest treball, desenvolupar un motor gràfic.

Per construir un motor gràfic hem de decidir primer quina API volem fer servir. Una API gràfica és una capa d'abstracció que permet accedir de forma fàcil a la **graphics pipeline**, explicada en detall més endavant. Seguidament hem de decidir quin llenguatge de programació farem servir, d'una llista reduïda compatible amb la API que hem escollit. Finalment, cal decidir quina estructura ha de tenir el motor i escollir una forma de estructurar l'escena a representar.

La construcció d'un motor gràfic complet és un procés llarg on intervenen moltes persones. Dividir la feina en mòduls és una forma de canalitzar el volum de feina, però pot causar problemes ja que molts mòduls requereixen d'altres mòduls, i es requereix un nivell d'organització elevat. En aquest treball jo m'he centrat principalment en el mòdul de generació de terreny procedural.

Una part molt important de tot videojoc, és la representació interna del *mapa*. A mesura que ha anat avançant la capacitat gràfica de còmput, els *mapes* dels videojocs han esdevingut més complexos. Inicialment eren mapes petits, on es mirava de representar una petita part d'una ciutat o fortalesa, i més contemporàniament, amb els coneguts com jocs tipus **openworld**, on el *mapa* passa a ser un món sencer totalment interactiu.

1.1 Motivació personal

Com molta gent, em vaig introduir al món de la informàtica gràcies als videojocs. Els videojocs també em van introduir al món de la programació, on alguns jocs de tipus **sandbox** ofereixen eines per programar internament.

Al llarg dels anys he vist evolucionar els videojocs, veient com oferien millor gràfics a mesura que la tecnologia avançava. Però una cosa que em fascinava encara més que els gràfics, era veure com, a mesura que anàvem tenint més memòria, avançaven els entorns on esdevenien aquests videojocs.

La meva afició per els jocs **sandbox** em va portar al joc **Space Engineers**, que inicialment era un **sandbox** que permetia construir naus espacials fent ús de asteroides per aconseguir recursos. El joc em va fascinar tant que vaig decidir entrar dins de la comunitat de **Modding** (que es refereix a usuaris del joc que decideixen aportar alguna cosa al desenvolupament, en forma de **Mod**), molt extensa i amb el suport total dels desenvolupadors. Dins d'aquest entorn vaig desenvolupar un mod anomenat **Saturn Moon Mods**, on a partir d'asteroides molt grans i fent ús d'efectes, vaig proporcionar al joc la primera iteració de planetes.Més endavant vaig entrar dins del seu **CTG** (Closed Testing Group), on vaig ajudar als desenvolupadors a testejar la seva implementació de planetes.

Finalment com objectiu de la carrera, des de bon inici, em vaig plantejar que volia construir un motor gràfic que s'apropés el màxim possible al resultat aconseguit per **Space Engineers**.

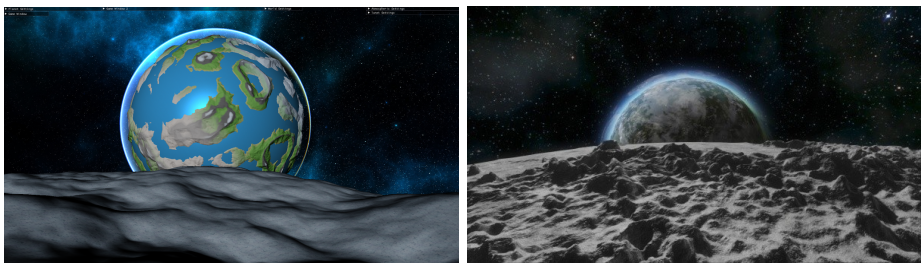


Figura 1.1: Comparació resultats entre el meu motor i **Space Engineers**

CAPÍTOL 2

Viabilitat

Pel desenvolupament d'aquest projecte hem intentat reduir el cost al mínim, fent ús només de programari lliure pel què esdevé la part del software del projecte.

Per part dels diferents models i textures, tos són d'ús públic, obtinguts a través de la web. En cas de que es volgués fer un ús comercial hauria de validar les llicències amb les directrius de **CreativeCommons**.

2.1 Recursos tècnics

El programari i hardware utilitzat és el següent:

Software:

- **SO**: Ubuntu 20.10
- **IDE**: CLion amb llicencia d'estudiant.

Hardware:

- **CPU**: AMD Ryzen 7 2700 Eight-Core Processor
- **Targeta gràfica**: NVIDIA GeForce RTX 2060

A més, per l'edició d'imatges, hem utilitzat **GIMP** i per l'edició de models, **Blender**, tot programari lliure que no suposa cap cost pel desenvolupador.

2.2 Recursos humans

Al desenvolupar només un mòdul del motor gràfic, únicament apareixen dues figures, descrites a continuació:

- **Programador**: Encarregat de desenvolupar el mòdul i del disseny de l'estructura. Sou base 26€/h.
- **Project Manager**: Supervisa i coordina les diferents tasques que ha de complir el programador. Sou base 32€/h.

2.3 Resum costos

Resumim el temps que ha dedicat el programador a les diferents tasques per a la construcció del motor:

Tasca	**Hores**	**Cost total (€)**
Estudi OpenGL	10	260
Creació d'una pantalla	2	52
Renderitzat de primitives bàsiques i aprenentatge bàsic de shaders	6	156
Renderització d'objectes formats per petites malles de vèrtex	4	104
Creació d'un parser fent ús de la llibreria rapidxml per llegir models i crear primitives bàsiques	10	260
Creació d'una càmera que permet mourem per l'escena	5	130
Afegir iluminacio local, global i en forma de llanterna	12	312
Afegir efectes de post processat	4	104
Afegir efecte de cel infinit	2	52
Estudi i implementació de la tècnica de rendering Instance Rendering	8	208
Creació de terreny bàsic fent ús de tècniques de soroll	13	338
Creació d'ombres al terreny fent ús de la tècnica Shadow Mapping	18	468
Afegir textures al terreny amb restricció d'alçada	15	390
Creació de terreny en forma esfèrica (planeta)	15	390
Afegir textures al planeta amb restricció d'alçada	4	104
Millora de la càmera per fer ús de quaternions i permetrem mourem lliurement en qualsevol axis	20	520
Afegir GUI per modificar les propietats del planeta des de l'editor	2	52
Afegir efectes atmosfèrics i GUI per editar-los	20	520
Millora la qualitat de les textures amb la tècnica Normal Mapping	6	156
Parametritzar els planetes de manera que es puguin carregar des del parser	8	208
Afegir una nova restricció a textures del planeta de manera que es puguin crear zones geogràfiques (biomes)	12	312
Afegir tècniques de tessel·lació i nivell de detall basat en distancia al shader	8	208
Millores de rendiment a nivell de shader	22	572
Afegir aigua amb reflexions i animacions	12	312
Total	203	5278

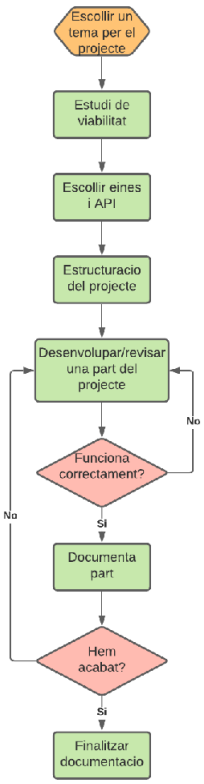
Finalment el temps dedicat pel cap de projecte (professor) suposa un 25% del temps del programador (jo), ja que s'ha de coordinar amb altres programadors:

	Hores	Cost (€)
Programador:	203	5278
Cap de projecte:	50.75	1624
Total:	253.75	6542

CAPÍTOL 3

Metodologia

Per el desenvolupament d'aquest treball s'ha seguit una metodologia personalitzada acordada amb el cap del projecte. Aquesta metodologia consisteix en fer un estudi previ del projecte, decidir les diferents eines que farem servir, fer una estructuració clara del projecte en parts i desenvolupar aquestes de manera independent assegurant que aquestes funcionen correctament abans de passar a la següent part. Podem resumir la dinàmica de treball amb el següent diagrama:



Descripció detallada de la metodologia:

- Inicialment s'escull un tema de treball i es fa un estudi de la seva viabilitat.
- Una vegada decidit el tema, s'escull les eines i el **framework** sobre el que volem treballar.
- Seguidament es divideix el projecte en mòduls. Si algun d'aquests mòduls resulta massa complex, es torna a fer una subdivisió recursiva fins que la tasca sigui assumible.
- A continuació es desenvolupa/revisa una part i es comprova el funcionament.
  - Si aquesta funciona correctament, es documenta breument i es comprova si hi ha més mòduls a realitzar.
  - En cas de que aquesta no funciones correctament, es revisa o es torna a desenvolupar de zero.
- Finalment, si ja no queden més parts a realitzar, es finalitza la documentació.

Aquesta metodologia proporciona un flux de treball modular, on cada mòdul es revisat pel cap de projecte i es decideix si passar o no al següent mòdul. D'aquesta manera cada mòdul és independent de la resta i s'assegura que continua funcionant després de realitzar canvis.

En el cas concret de desenvolupar un motor gràfic, els mòduls independents continuen funcionant, però es pot donar el cas que funcionalitats antigues no estiguin adaptades a funcionalitats noves i, en cas d'afegir una funcionalitat nova, s'hauria de comprovar cada vegada que aquesta funciona amb la resta, o decidir que dues funcionalitats són incompatibles.

La interacció amb el cap de projecte es realitza cada 15 dies, on es fa una reunió, es revisa la feina feta i es decideix si es dóna per bona o no. En el cas favorable, el mòdul realitzat s'incorpora al projecte final, mantenint un control de versions estricte que ens permet tornar enrere en cas de problemes inesperats.

CAPÍTOL 4

Planificació

4.1 Planificació del motor bàsic

L'inici del projecte es remunta al **Juny 2021** on inicialment vam fer un estudi previ de les diferents API's gràfiques que existeixen, seguit per una construcció bàsica d'un motor que permet:

- Renderitzar objectes amb il·luminació i ombres.
- Càmera estil **First-Person Shooter o FPS** per moure's per l'escena.
- **Parser** per representar una escena en **XML** i poder fer-ne modificacions.
- Aplicar efectes de postprocessat a la pantalla.
- Carregar de forma fàcil qualsevol tipus de model/textura.

Per fer això vam decidir fer servir la API de **OpenGL**, ja que té l'avantatge que és fàcil d'aprendre i és multiplataforma. Com a llenguatge de programació vam escollir **C++**, per la familiaritat i versatilitat que dóna. Per la creació de finestres, vam optar també per una opció fàcilment adaptable a qualsevol plataforma, coneguda com la llibreria **GLFW**, que permet creació de finestra i captura d'esdeveniments de teclat i ratolí.

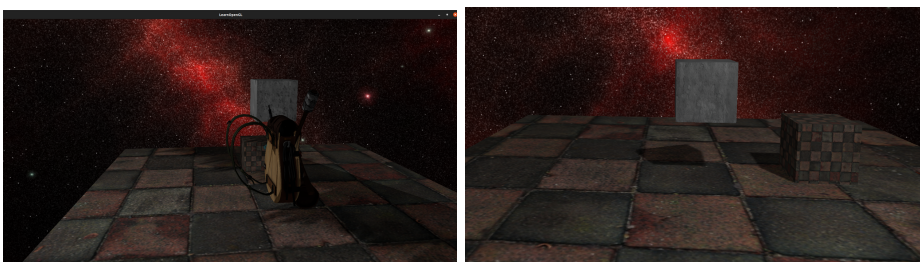


Figura 4.1: Models amb textures, il·luminació i ombres

4.2 Planificació del terreny procedural

Un cop creada l'estructura del motor, es va afegir un mòdul bàsic de terreny procedural fent ús de tècniques de soroll (**noise**) per crear la geometria, un algorisme bàsic per interpolar textures basat en l'alçada del terreny i l'habilitat d'afegir objectes (models amb textures) amb ombres.

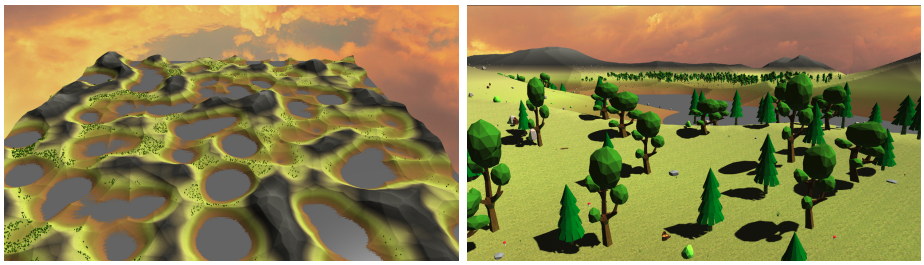


Figura 4.2: Terreny procedural amb objectes a l'escena

Per fer això es van realitzar les següents tasques:

- Estudi i implementació de la tècnica de rendering **Instance Rendering**
- Estudi previ de les diferents tècniques de soroll i diferents llibreries que la implementen.
- Aprendre a utilitzar la llibreria **FastNoise Lite** per generar la geometria.
- Creació de terreny bàsic fent ús de **FastNoise Lite**.
- Creació d'ombres al terreny amb la tècnica de **Shadow Mapping**.
- Creació de milers d'objectes al terreny amb la tècnica d'**Instance Rendering**.

4.3 Planificacio planeta procedural

Aconseguit el terreny, vam decidir extrapolar aquest a una superfície esfèrica i, gràcies a la llibreria **FastNoise Lite**, crear un planeta procedural a partir d'una llavor, i amb l'ajuda de la llibreria **ImGui** crear una petita interfície per modificar-lo. Veure figura 4.3:

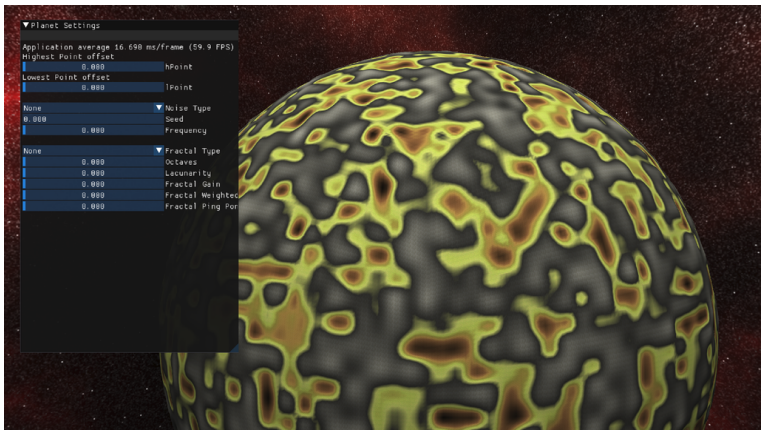


Figura 4.3: Primera iteració de planeta procedural

Fet això, el següent pas va ser millorar els paràmetres de la funció de soroll i afegir una petita atmosfera amb la interfície per modificar-la. Veure figura 4.4:

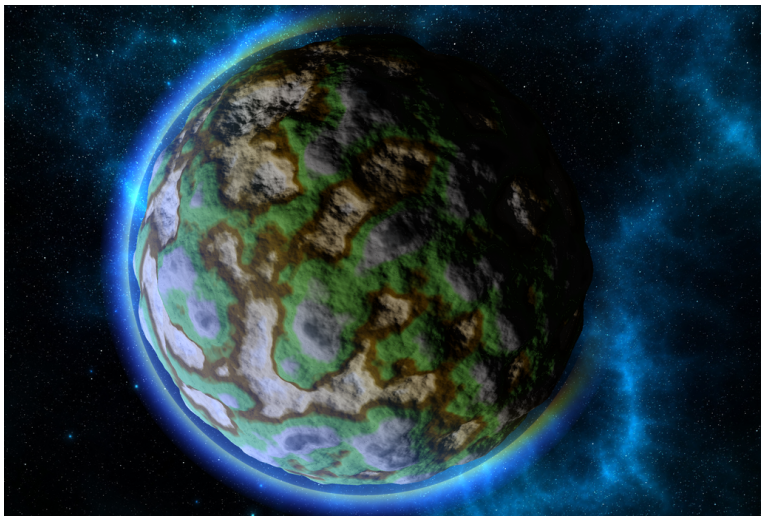


Figura 4.4: Planeta amb atmosfera

Seguidament, per millorar la qualitat del planeta, es va afegir una nova restricció a la texturació, en funció de la latitud de l'esfera, creant diferents biomes. Veure figura 4.5 :

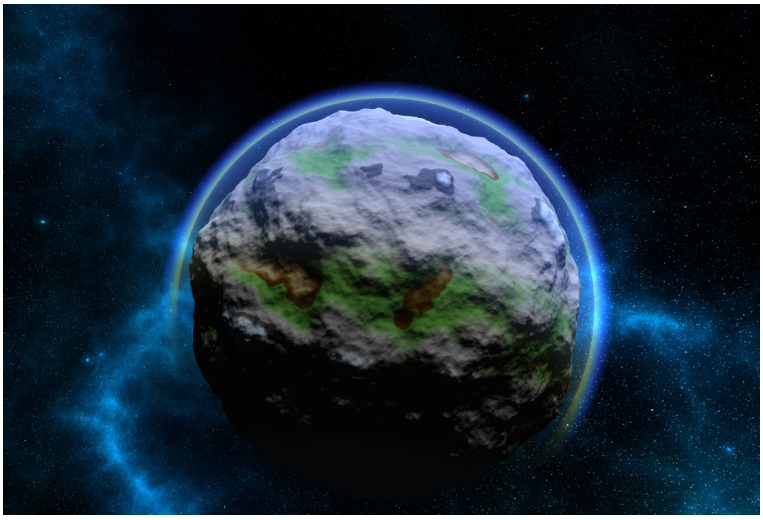


Figura 4.5: Planeta amb diferents biomes

Finalment es van fer moltes millores en l'algorisme per millorar rendiment, a la vegada que es va parametritzar els planetes perquè es puguin carregar des del fitxer **XML**, i es va afegir aigua amb reflexions, efectes de moviment i l'habilitat de poder visualitzar el planeta a qualsevol distància sense perdre qualitat. Veure figura 4.6:

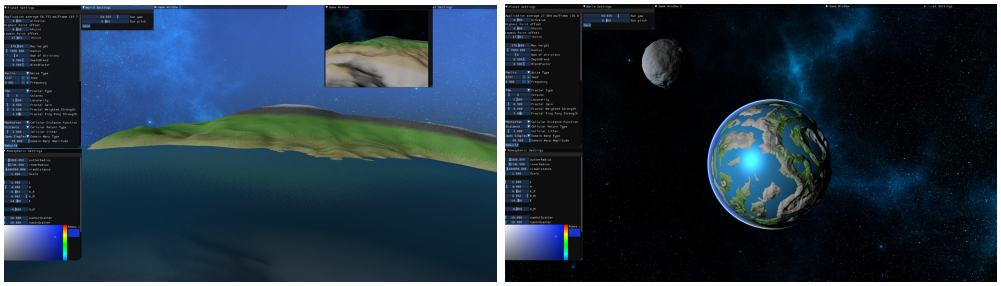


Figura 4.6: Planeta amb oceans

Resumint les tasques realitzades per l'elaboració del planeta:

- Estudi de les diferents formes de crear una esfera geomètrica.
- Decisió i implementació d'una esfera geomètrica construïda a partir de subdividir un cub.
- Fer ús de la llibreria **FastNoise Lite** per aplicar la funció de soroll en 3D.
- Modificar l'algorisme de interpolació de textures perquè funcioni en una esfera.
- Estudi de com implementar efectes atmosfèrics (**scattering effects**).
- Decisió i implementació d'efectes atmosfèrics.
- Implementació d'una nova restricció a l'aplicació de textures basada en la latitud de l'esfera (biomes).
- Implementació de tessel·lació i nivell de detall basat en distància.
- Millora de qualitat en distàncies grans.
- Implementació d'aigua amb reflexions i efectes de moviment.

4.4 Resum planificació

La feina feta s'ha estructurat en 3 grans mòduls, cada mòdul amb uns objectius fixats resumits en un conjunt de tasques, on cada tasca s'ha realitzat majoritàriament aïllada del conjunt final fent ús del control de versions de git.

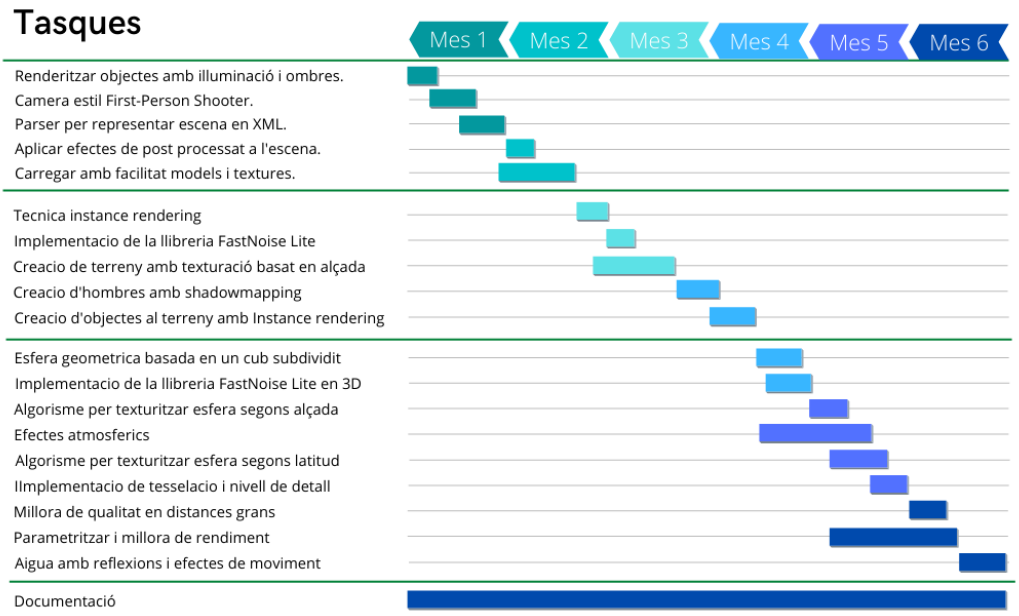


Figura 4.7: Resum planificació

CAPÍTOL 5

Marc de treball i conceptes previ

5.1 API's gràfiques

Una **API** (Application Programming Interface) gràfica proporciona una capa d'abstracció entre el programador i la targeta gràfica, mitjançant accés a la **Graphics Pipeline** a través d'un llenguatge de programació a alt nivell. A mesura que el món dels videojocs s'ha tornat més popular, aquesta capa d'abstracció s'ha tornat més necessària per tal d'atraure a més desenvolupadors. Existeixen diverses **API's** gràfiques, les més conegudes i populars són les següents:

- **Direct3D**: Microsoft DirectX incorpora el conegut com Direct3D una **API** construïda per Microsoft Windows que ofereix accés a la *graphics pipeline* a través del llenguatge de programació a alt nivell **HLSL** (High Level Shader Language).
- **OpenGL**: **OpenGL** (Open Graphics Library) és l'**API** escollida per realitzar aquest treball a causa de la facilitat i extensa documentació que existeix a la web. Creada originalment per **Silicon Graphics** al 1992 i mantinguda per **Khronos Group** des del 2006, ofereix accés a la *Graphics Pipeline* a través de **GLSL** (OpenGL Shading Language).
- **Vulkan**: **Vulkan**, creat també pel **Khronos Group**, neix de la demanda de desenvolupadors per tenir accés a més baix nivell a la *Graphics Pipeline*. Vulkan millora considerablement l'accés a CPU i facilita molt la feina a l'hora de paral·lelitzar el treball.

La complexitat inicial de la decisió d'escollir **OpenGL** vs **Vulkan**, ve de la dificultat inicial que té aprendre **Vulkan**. **OpenGL** és l'opció recomanada per iniciar-se al món gràfic, ja que abstrau de coneixements sobre els controladors gràfics.

5.2 OpenGL

5.2.1 Representació de la geometria

A **OpenGL** tot està representat en un espai 3D, però la nostra pantalla és plana (2D). Per tant, per tal de representar geometria 3D, l'hem de representar en 2D píxels. Això s'aconsegueix mitjançant la **Graphics Pipeline** que, resumidament, té com a entrada unes coordenades en 3D i les converteix a píxels de diferents colors a la pantalla. Algunes parts de la *pipeline* són configurables mitjançant petits programes coneguts com **Shaders**, escrits en el llenguatge **GLSL**.

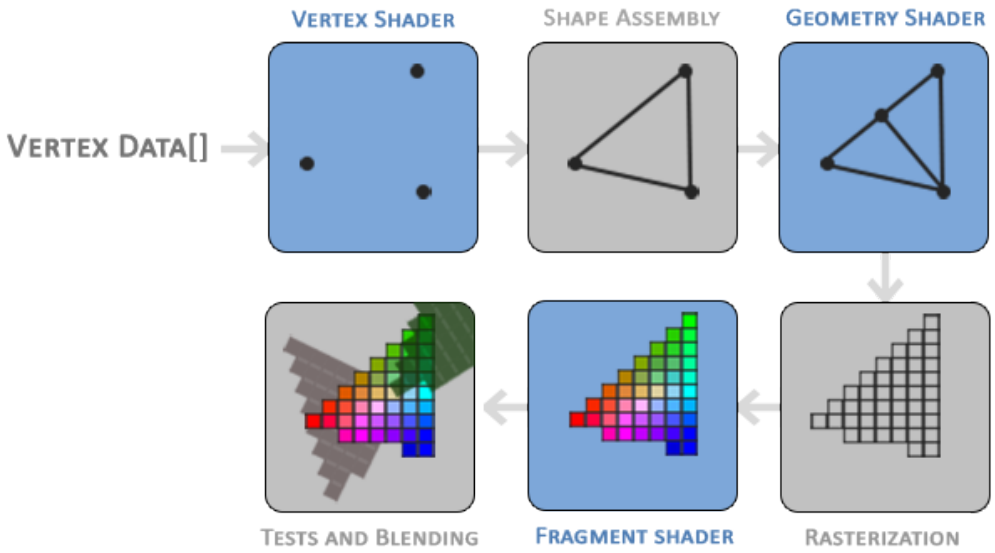


Figura 5.1: Representació de la **Graphics Pipeline**

L'entrada a la *pipeline* és un array de **vèrtex**, on cada **vèrtex** és un conjunt de dades per cada coordenada tridimensional. Aquestes dades s'especifiquen mitjançant **vèrtex attributes** on podem codificar dades com la normal del vèrtex, la posició, les coordenades de textures.

També hem d'especificar de quin tipus de primitiva es tracta, si volem que es formin triangles, punts o línies.

OpenGL permet configurar diferents etapes de la *pipeline*. Inicialment és treballa amb dos etapes d'aquesta, el **vèrtex** i **fragment shader**.

La primera part de la *pipeline* és el **vèrtex shader** que rep com a input un únic **vèrtex** i fa diferents transformacions a les dades per tal que es puguin visualitzar a la pantalla.

Seguidament es fan una sèrie de transformacions segons el tipus de primitiva seleccionada, i a l'etapa de rasterització es converteix en un conjunt de píxels o fragments.

El **Fragment Shader** rep com entrada els fragments resultants de l'etapa de rasterització i s'encarrega de decidir quin color tindrà cada píxel. En aquesta etapa es tenen en consideració factors com la llum i el color, ombres, diferents efectes.

Finalment es comprova la profunditat del fragment i la transparència, si un fragment s'ha de descartar perquè hi ha un altre davant o, en el cas de tenir transparència, barrejar els dos colors segons una especificació donada.

5.2.2 Shaders

Com hem mencionat anteriorment, els shaders són petits programes escrits en **GLSL** que ens permeten programar les diferents etapes de la *pipeline*. Aquests programes estan aillats uns dels altres i només es comuniquen via entrada i sortida de variables.

5.2.2.1 GLSL

GLSL és un llenguatge de programació molt semblant al C, enfocat a transformacions de vectors i matrius. Cada shader programat en GLSL ha d'anar acompanyat de l'especificació versió de OpenGL, una funció **main**, les diferents entrades i sortides, i un conjunt de variables de tipus **uniform**.

Els diferents tipus que pot tenir una variable a **GLSL** són int, floats, double, uint i bool, i per representar conjunts tenim vectors i matrius:

- **Vectors**: Es representen amb el prefix **vecn** on n és un nombre (1,2,3 o 4) que determina la mida del vector. També es poden definir vectors de diferents tipus com **bvecn**, que representa un vector de mida n de booleans.
- **Matrius**: Es representen amb el prefix **matn** o **matnxm** on n i m són respectivament nombres (2,3 o 4) que diuen el nombre de columnes i files que té la matriu. Les matrius només poden ser de tipus float o, en versions més avançades d'**OpenGL**, de tipus double amb el prefix **dmatn**.

Les variables d'entrada i sortida es representen amb el prefix *in* i *out* respectivament, seguit del tipus i el nom de la variable. A les etapes de **vèrtex** i **fragment shader** s'han de definir dues sortides obligatòries, **gl_Position** de tipus **vec4** al **vèrtex shader** amb la posició transformada del vèrtex i FragColor de tipus **vec4**, representant el color final del píxel al **Fragment Shader**.

Les variables de tipus **uniform** ens proporcionen una forma de passar informació del programa (**CPU**) als shaders (**GPU**). Aquestes variables són d'àmbit global, i només de lectura. Per tant, podem accedir a una variable **uniform** des de qualsevol etapa de la **pipeline**.

Existeixen altres variables de tipus **Uniform Buffer Objects** o **UBO's**, però no formen part dels coneixements previs i les explicarem en capítols posteriors.

5.2.3 Textures

Definim textura com una imatge bidimensional que proporciona detall a un objecte. Per saber a on ha d'anar aquesta textura, codifiquem a cada vèrtex un atribut que descriu les coordenades de textures. Aquestes coordenades van de 0 a 1 en els eixos x i y, i tenen l'origen a (0,0), que correspon a la part esquerra i avall de l'espai de coordenades. Veure figura 5.2:

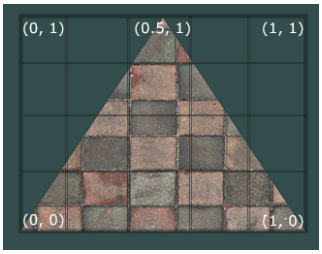


Figura 5.2: Espai de coordenades de textures

Quan emboliquem un objecte amb una textura, en el cas que l'objecte sigui més gran que aquesta, ens sortim de l'espai de coordenades. Llavors **OpenGL** ens proporciona diferents opcions per tractar amb aquesta casuística. Veure figura 5.3:

- **GL_REPEAT**: Repeteix el patró de la textura per tot l'objecte.
- **GL_MIRRORED_REPEAT**: Igual que l'anterior, però amb la peculiaritat que també reflecteix la imatge a cada repetició.
- **GL_CLAMP_TO_EDGE**: Limita les coordenades entre 0 i 1. Per tant, fa que a valors més grans a 1 la imatge s'estiri.
- **GL_CLAMP_TO_BORDER**: Coordenades fora del rang se'ls hi aplica un color de vora especificat.

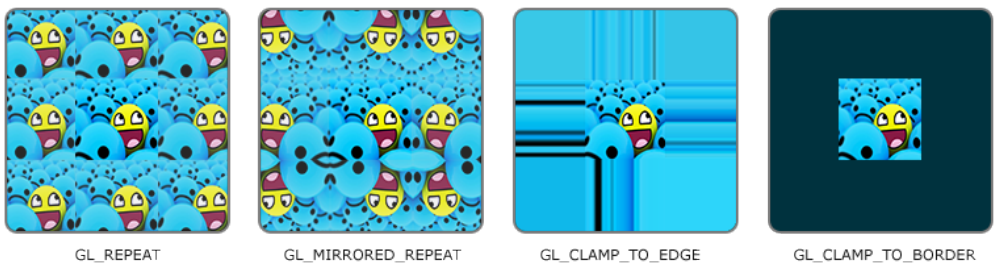


Figura 5.3: Opcions per tractar casuística d'avaluació d'una textura fora de l'espai de coordenades

CAPÍTOL 6

Requisits del sistema

Per poder compilar i fer funcionar aquest projecte, hem de complir amb una serie de requeriments:

6.1 Requisits funcionals

Els requeriments funcionals de l'aplicació són els següents:

- L'usuari ha de poder obrir una escena amb un o més planetes i navegar per aquesta.
- L'usuari ha de poder modificar una escena a través de la interfície gràfica o l'edició del fitxer **XML**.
- L'usuari ha de poder guardar l'escena modificada a disc.
- L'usuari ha de poder afegir / modificar els efectes atmosfèrics a un planeta determinat.
- L'usuari ha de poder afegir els efectes aquàtics a un planeta determinat.

6.2 Requisits no funcionals

6.2.1 Requisits de software

Per desenvolupar aquest projecte hem fet servir una sèrie de llibreries que són necessàries pel funcionament d'aquest:

- GLFW (versió 3)
- RapidXML (última versió)
- FastNoise Lite (última versió)
- ImGUI (última versió)

- GLAD (última versió)
- stb image (última versió)
- GLM (última versió)

6.2.2 Requisits de hardware

Pel que comporta hardware, els requisits mínims són els següents:

Sistema operatiu:	Ubuntu 20.10 o superior
CPU:	AMD Ryzen 7 2700 / Intel Core i7-8700
Targeta gràfica:	Nvidia Geforce 1060Ti / AMD Radeon RX 580/590

CAPÍTOL 7

Estudi i decisions

7.1 Sistema operatiu

Figura 7.1: Ubuntu Logo

La realització del projecte s'ha dut a terme fent servir Ubuntu Linux 20.10 i, posteriorment, amb Ubuntu Linux 21.04.

7.2 Llenguatge de programació

Figura 7.2: C++ Logo

C++ es un llenguatge molt versàtil que proporciona al desenvolupador manipulació de memòria a baix nivell. Aquesta funcionalitat fa que sigui el llenguatge per excel·lència, alhora de desenvolupar un motor gràfic. Creat per **Bjarne Stroustrup** com una extensió del llenguatge C amb classes, al llarg del temps s'han anat afegint noves funcionalitats, mantenint sempre la eficiència i flexibilitat.

7.3 IDE

Figura 7.3: CLion Logo

CLion de **IntelliJ** es un **IDE** (*Integrated Development Enviorment*) que proporciona l'usuari un entorn còmode i ràpid on desenvolupar projectes en C++. Aquest entorn conté moltes funcionalitats, de las que destaquem les següents:

- **Editor de text**: Proporciona totes les funcionalitats de la majoria de editors de codi, com ressaltat de codi, auto-completar el codi, comprovació d'errors i una funcionalitat no tant comú on es mostra el nom de la variable que pren com a paràmetre una funció quan es crida.
- **Editor de configuracions per compilar**: Permet crear diferents perfils de compilació on es pot incloure diferents paràmetres o llibreries.
- **Debugger**: Permet trobar fàcilment errors en temps d'execució i/o compilació, amb l'habilitat de poder veure l'estat de les variables en un precís moment afegint **breakpoints** (punts de tall) al codi.
- **Control de versions**: Permet interactuar d'una forma fàcil amb git, amb l'habilitat de clonar repositoris, fer **push**/**pull**, creació de branques i facilitat per canviar entre aquestes, creació de **pull-request**, editor que mostra les diferencies entre local i remot, i moltes més.
- **Refractor**: Permet fer canvis massius al projecte, com fer un canvi de nom de variable, de forma segura i ràpida.

7.4 OpenGL

Figura 7.4: OpenGL Logo

OpenGL es una **API** (*Application Programming Interface*) gràfica multiplataforma, que permet representar gràfics 2D i 3D en una pantalla. Aquesta **API** permet interactuar directament amb la targeta gràfica i aconseguir un **renderitzat** basat en hardware.

El verb renderitzar fa esmena a representar en un dispositiu de sortida d'un ordinador (una imatge o una escena), generalment en dues dimensions, simulant-ne els efectes òptics de llum, ombra, color, textura o moviment a partir de les dades d'un model computacional en 3D.

L'ultima versió disponible es la 4.6 (31 de Juliol, 2017) i des de llavors, els creadors, **Khronnos Group**, han abandonat el desenvolupament per centrar-se en la seva versió millorada, **Vulkan**.

Per la creació d'aquest projecte hem hagut d'aprendre les diferents funcionalitats que ofereix **OpenGL**, tant a nivell de software (**CPU**) com a nivell de hardware (**GPU**).

7.4.1 Buffer Objects

Buffer objects són objectes de **OpenGL** que guarden matrius de dades a la memòria de la targeta gràfica. Es pot guardar informació sobre els vèrtex, píxels d'una imatge i molt més.

Tot objecte a **OpenGL** s'ha de crear inicialment, associar-lo a una regió de memòria amb un apuntador, carregar les dades i finalment a l'acabar, eliminar-lo.Les dades es guarden de forma linear dins de la memòria, amb un estat **mutable** o **immutable**:

- **Immutable**: Les dades tenen una regió de memòria prefixada i no es pot canviar. Per actualitzar el buffer, s'ha de netejar prèviament.
- **Mutable**: Permet editar la informació que existeix en una regió de memòria.

Existeixen molts tipus de **buffer objects**, a continuació fem una descripció dels que hem utilitzat en aquest treball i quina funció desenvolupen:

- **VBO**: *Vertex Buffer Object* o **VBO** permet carregar dades del vèrtex (posició, normal, coordenades de textura...) directament a la memòria de la gràfica.
- **EBO**: *Element Buffer Object* o **EBO**, serveixen pel **render indexat**. Aquesta tècnica permet reduir la quantitat de vèrtex considerablement fent ús d'índexs que formen primitives bàsiques. En el cas que estiguem construint el nostre model a base de triangles, podem reaprofitar un vèrtex d'un triangle adjacent per construir el nou triangle.
- **FBO**: *Frame Buffer Objects* o **FBO**, permet la creació de **FrameBuffers** definits per l'usuari. A **OpenGL** l'escena es renderitza per defecte a un **FBO**. Poder crear el teu propi **FBO** permet renderitzar sense que es mostri per pantalla amb diverses finalitats, com per exemple, renderitzar des d'un angle diferent, guardar l'escena a una imatge i crear diverses finestres...

Sabem que un vèrtex conté diferents atributs, per tant la gràfica ha de poder diferenciar entre aquests. El procediment per gestionar la geometria a la gràfica passa per l'ús del que es coneix com a *Vertex Attribute Pointer* o **VAO**, que consisteix en tenir un **VBO** associat i opcionalment, un **EBO**, i a continuació fer una subseqüent crida a **glVertexAttribPointer** on li diem a la gràfica on pot trobar la informació. El procediment a seguir es el següent:

- Inicialment creem el **VAO** amb **glGenVertexArray** que rep com a paràmetre la mida del buffer i un nombre natural per refèrencia, que tornarà l'apuntador a memòria. Seguidament, fem el mateix pel **VBO** i **EBO** amb els mateixos paràmetres, però amb la funció **glGenBuffer**.
- Seguidament marquem com actiu el **VAO** amb la funció **glBindVertexArray** que rep com a paràmetre l'apuntador que hem obtingut al generar-lo. A partir d'ara totes les subseqüents crides quedaran associades a aquest **VAO**.
- Continuem activant el **VBO** amb **glBindBuffer**, funció genèrica que rep com a paràmetre el tipus de buffer, en aquest cas concret **GL_ARRAY_BUFFER**, que diu que la informació es tracta d'una matriu, i l'apuntador del objecte.

- Carreguem la informació al **VBO** amb **glBufferData** que rep com a paràmetres:
  - El tipus de buffer amb el que estem treballant, en aquest cas concret és **GL_ARRAY_BUFFER**.
  - La mida de la informació en bytes.
  - Un apuntador a la primera posició de la matriu de vèrtex.
  - Una opció si volem que aquesta memòria sigui estàtica o dinàmica.
- Seguidament repetim el procediment anterior amb l'**EBO** especificant el tipus de dades **GL_ELEMENT_ARRAY_BUFFER** i passant la matriu d'índexs.
- Finalment, pels n atributs del vèrtex, l'activem amb **glEnableVertexAttribArray**, que rep com a paràmetre un enter que servirà per indexar l'atribut, i especifiquem a on es troba amb **glVertexAttribPointer**, que rep com a paràmetre:
  - L'index de l'atribut.
  - El nombre d'elements que té l'atribut.
  - El tipus de l'atribut.
  - Especifica si volem les dades normalitzades.
  - El desplaçament de la memòria.
  - On comença l'atribut dins de la memòria.

Aquest procediment ens permet simplificar el procediment de renderitzar diferents objectes, ja que únicament hem d'activar el **VAO** i fer una crida al render, i **OpenGL** ja sabrà quina regió de memòria ha de fer servir i a on es troba cada atribut dins d'aquesta regió, quina mida tenen... Veure figura 7.5

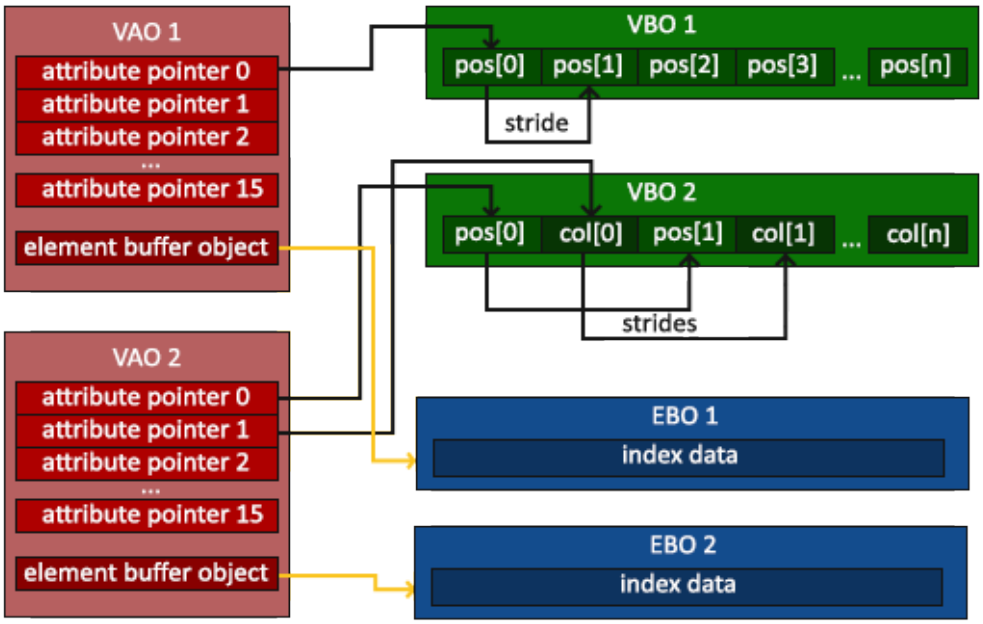


Figura 7.5: Representació de diferents **VAO's** [de Vries 014]

7.4.2 Shaders

Els **Shaders** son petits programes que treballen amb la **GPU**, de forma paralela, en una secció determinada de la **Graphics Pipeline**. El llenguatge que fa servir **OpenGL** es **GLSL**. En aquest apartat explicarem cada part de la *pipeline* de forma detallada i quin ús en fem.

7.4.2.1 Vertex Shader

El **vertex shader** rep com a entrada un únic **vèrtex**, configurat prèviament amb els atributs a nivell de **CPU**, carregat a la memòria de la **GPU** i enllaçat fent servir el **VAO**. Quan configurem el **VAO** associem un enter a cada atribut del **Vertex**, i al **Shader** el recuperem de la següent manera:

layout (location = **n**) in **tipus** nom

On **n** determina l'enter que hem fet servir quan configurem el **VAO** i tipus ens diu el tipus de la variable. En aquest punt de la *pipeline* ens interessa fer les transformacions necessàries a la geometria per tal que altres seccions de la *pipeline* puguin treballar amb aquesta informació.

Aquest **Shader** té variables de sortida (**out**) predefinides per **OpenGL**, una d'ús obligatori, i dues d'ús opcional:

- **gl_Position**: La posició transformada a l'espai normalitzat. (Obligatòria)
- **gl_PointSize**: En cas que la primitiva sigui **GL_POINT**, determina la mida d'aquest. (Opcional)
- **gl_ClipDistance**: **OpenGL** permet definir *clip-planes*, on podem definir una zona acotada per un pla on objectes per sota/sobre d'aquest no es renderitzen. (Opcional)

7.4.2.2 Tesselation Control Shader (TCS)

En les versions més modernes de **OpenGL** s'aprofita una propietat de les gràfiques que permet subdividir la geometria en triangles més petits. Això s'anomena **Tesselation** o tessel·lar. Aquesta secció de la *pipeline* es divideix en 2 parts.

En la primera secció (**TCS**), inicialment definim la sortida de la següent manera:

layout (vertices = n) out

On **n** es el nombre de **punts de control** que tindrà la sortida. Aquests punts de control ens diuen quantes vegades s'executara aquest **shader** per **vertex**. Això condiciona l'entrada d'aquest **shader**, ja que inicialment del **Vertex Shader** surt un únic valor per variable, al **TCS** arribarà un vector que tindrà tants elements com punts de control.

Amb aquest vector d'entrada construïm la nova geometria, on podem decidir si volem fer les subdivisions en funció de variables externes com la distància, orientació...

Finalment, la sortida és definida per vectors de tants elements com punts de control, que rep la següent secció de la *pipeline*.

7.4.2.3 Tesselation Evaluation Shader (TES)

En aquesta secció de la *pipeline* tenim d'entrada un conjunt de vectors de dades, això suposa un problema pel següent pas, ja que ens interessa que a l'última secció de la *pipeline* ens arribi un únic valor. El que fem es interpolar els diferents valors que ens venen de la **TCS** per tal que el **Fragment Shader** rebi un únic valor.

7.4.2.4 Fragment Shader

L'última secció programable de la *pipeline* rep els valors interpolats del **TES** i genera un *fragment* o *píxel* amb un color determinat i un valor que determina la profunditat a l'escena.

En aquesta secció de la *pipeline* es calcula l'il·luminació, ombres, efectes, transparència i es genera un color. Aquest color és una sortida obligatòria que ha de tenir tot **Fragment Shader**. A mes **OpenGL** proporciona variables que et permeten canviar la profunditat d'un fragment, o aplicar un algoritme propi per calcular la profunditat.

En aquest apartat també apliquem les textures al model. Aquestes tenen un tipus especial **sampler2D/sampler3D**, que explicarem en més detall mes endavant.

Finalment, quan hem calculat les propietats que tindrà el material final, definim una sortida de tipus **vec4** que determina el color final del *píxel*.

7.4.3 Textures

Una textura és una imatge que interpretem com una matriu on cada element pot contenir un o més valors segons el color de la imatge:

- **Greyscale**: Una imatge en blanc i negre.Només conté un valor per element de la matriu, entre 0..255.
- **RGB**: Una imatge acolorida, on cada element conté tres valors, vermell, verd i blau, entre 0..255.
- **RGBA**: Igual a l'anterior però amb un component mes que determina la transparència de la imatge, també entre 0..255.

Per carregar textures a la nostra aplicació fem ús de la llibreria **stb_image** que donada la ruta de la imatge, ens retorna un vector de caràcters, on cada caràcter representa un color, l'amplada i alçada de la imatge, i el nombre de components descrits anteriorment. A aquesta textura li associem un **ID** que generem prèviament amb **glGenTextures**, enllacem o activem amb **glBindTextures** i finalment carreguem la informació a la memòria de la gràfica amb **glTextImage**.

Cada model tindrà el seu llistat de textures carregades a memòria, i abans de renderitzar les haurem de marcar com actives, carregar-les al **shader** en forma de **uniform** i dir de quina textura es tracta. **OpenGL** permet fins a 32 textures per cada crida al **shader**, amb **glActiveTexture**, que té com a paràmetre **GL_TEXTURE0..31** on informem **OpenGL** de quina textura es tracta.

Seguidament, al **Fragment Shader** rebem la textura com a **uniform sampler2D/3D**, la carreguem amb funcions internes d'**OpenGL** com **texture2D** o **textureGrad**, que rep com a paràmetre el **sampler2D** i les coordenades de textura, i retorna un **vec4** que representa el color.

Finalment, podem aplicar la textura com ve donada, o interpolar amb altres textures amb funcions internes d'**OpenGL** com **mix**, que rep dues textures i un enter que determina la proporcionalitat de la primera textura amb la segona i genera un color nou.

7.4.4 Transformacions

A l'apartat del **Vertex Shader** hem explicat que s'apliquen transformacions al model, aquestes transformacions determinen la posició, orientació, escala del model i posició relativa a la **càmera**. Per fer això definim 3 matrius de 4x4 que enviem al **Vertex Shader** en forma d'**uniforms**:

- **Model**: Cada objecte o model tindrà la seva pròpia matriu, que representa la translació, rotació i escalat del model. Aquesta variable de tipus uniform s'actualitza cada vegada que el model és renderitzat el que permet modificar l'objecte en temps d'execució. Construïm la matriu de la següent manera:
  - Inicialment definim una matriu de 4x4 identitat.
  - Apliquem una translació a aquesta matriu amb la posició del model.
  - Apliquem una rotació al resultat d'aplicar la translació.
  - Apliquem l'escalat al resultat d'aplicar la rotació.
- **View**: La matriu de vista és global per tots els models i ve determinada per la **càmera**. Expliquem en més detall en el següent capítol com es construeix aquesta, de forma resumida, converteix la posició del vèrtex de global a una posició acotada i normalitzada dins de la vista de la **càmera**.
- **Projection**: Determina la perspectiva que tindrà la vista de la **càmera**. Quan definim la perspectiva definim el pla proper i el pla llunyà. Això ens construeix una piràmide truncada on la geometria fora d'aquesta queda descartada.

La posició del fragment, que anomenem **FragPos** es construeix amb la posició global, multiplicat per la matriu model. Llavors la posició normalitzada dins de la vista, que determina la posició final del vèrtex (**gl_Position**) es calcula multiplicant la projecció, amb la vista i la posició del fragment convertida a **vec4**.

7.4.5 Camera

Quan parlem de la càmera a **OpenGL** parlem de les coordenades dels vèrtexs transformats amb la matriu de perspectiva de la càmera. La matriu **View** converteix les coordenades globals a coordenades relatives a la posició de la càmera i la seva direcció. Per definir la càmera necessitem la seva posició global, la direcció cap a on mira, un vector que apunti amunt i un que apunti a la dreta. Veure figura 7.6:

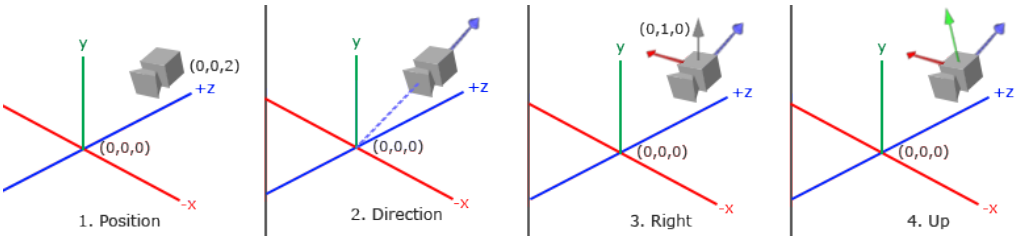


Figura 7.6: Sistema de coordenades de la càmera

Per calcular la direcció de la càmera tenim dues maneres. Una és mitjançant **Euler angles** i l'altre mitjançant **quaternions**. Inicialment, vam decidir implementar una basada en Euler angles, però posteriorment vam adaptar aquesta a quaternions ja que ens oferien mes flexibilitat.

Euler Angles.
Euler angles son 3 valors que poden representar qualsevol rotació en un espai tridimensional. Aquest valors reben el nom de **pitch**, **yaw** i **roll**. Veure figura 7.7:

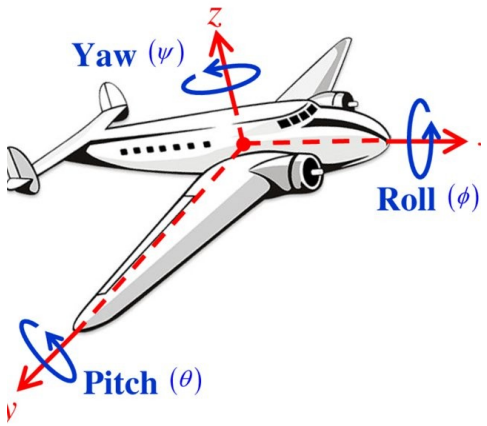


Figura 7.7: Representació dels angles pitch, yaw i roll

Yaw determina l'angle lateral, l'angle que varia quan girem d'esquerra a dreta, **Pitch** determina l'angle vertical, aquell que varia quan girem amunt/avall i **Roll** una rotació centrada en l'axis de la càmera.

Per calcular la direcció frontal de la càmera (on mirem), ho fem en funció del moviment del ratolí. Inicialment, comencem amb un valor de **pitch** i **yaw** per defecte i a mesura que movem el ratolí variem els angles. Si el ratolí es mou en l'eix vertical, incrementem el **pitch** amb la variació d'aquest moviment, altrament si es mou lateralment, incrementem el **yaw**.

Per calcular el vector frontal, convertim els angles a radians i calculem els 3 axis del vector de la següent manera:

$$x = \cos(Yaw) * \cos(Pitch)$$
$$y = \sin(Pitch)$$
$$z = \sin(Yaw) * \cos(Pitch)$$

Fet això ja podem calcular els altres vectors pendents. Per calcular el vector que apunta a la dreta, inicialment definim un vector que apunta amunt, i fem el producte vectorial entre la direcció i aquest per obtenir el vector perpendicular que apunta a la dreta.

Finalment per calcular el vector que apunta amunt, apliquem de nou el producte vectorial entre la direcció de la càmera i el vector que apunta a la dreta.

Una vegada tenim calculats tots els vectors i podem orientar la càmera lliurament per l'escena, ens interessa també poder moure'ns endavant/enrere o lateralment fent servir les tecles W/A/S/D. Per fer això, capturem quan l'usuari pressiona una d'aquestes tecles, definim una velocitat i modifiquem la posició:

- **Endavant o enrere**: Sumem/restem a la posició el vector frontal multiplicat per un escalar que hem definit com a velocitat.
- **Lateral dreta o esquerra**: Sumem/restem a la posició el vector lateral multiplicat per la velocitat.
- **Amunt o avall**: Sumem/restem a la posició el vector vertical multiplicat per la velocitat.

Finalment, calculem la matriu **View** amb la funció de **GLM lookAt**, que rep tres paràmetres: La posició de la càmera, la direcció on mira (vector frontal) i un vector que apunta amunt (vector vertical).

7.5 GLFW

GLFW és una llibreria **Open Source** i multiplataforma per **OpenGL** i **Vulkan** que permet crear finestres, rebre entrades del teclat o capturar moviments del ratolí entre d'altres.

Per crear una finestra i capturar el **FrameBuffer** a on renderitza **OpenGL** seguim els següents passos:

- Inicialment declarem una finestra nova amb tipus **GLFWwindow**, obtenim el monitor i creem la finestra amb **glfwCreateWindow**, que rep com a paràmetres la resolució, el títol de la finestra i el monitor. Determinem la resolució en funció de la resolució del monitor per obtenir la pantalla completa. Altrament, es pot definir una resolució personalitzada i prescindir del monitor.

- Seguidament, cridem a **glfwSetKeyCallback** que rep com a paràmetre la finestra, una funció **key_callback** i ens permet capturar quan l'usuari pressiona una tecla.

- El següent pas és capturar el context de **OpenGL**. Per fer això primer hem de cridar la funció **glfwMakeContextCurrent** que rep la finestra i la marca com a primària.

  Seguidament, declarem una funció **framebuffer_size_callback** que crida a **glViewport** d' **OpenGL** on es renderitza el **FrameBuffer**. Aquesta funció la passem com a paràmetre, juntament amb la finestra a **glfwSetFramebufferSizeCallback**, i el **FrameBuffer** es renderitza en aquesta finestra.

- Finalment, capturem el moviment del ratolí amb la funció **glfwSetCursorPosCallback** que rep com a paràmetres la finestra i una funció **mouse_callback.**

Per a capturar l'entrada de teclat, tenim dues opcions. La primera es fer servir el **callback** que hem definit prèviament, amb l'avantatge que disposem tant de la tecla que hem pressionat, com de l'acció (si hem pressionat, estem mantenint la tecla pressionada o si hem deixat anar la tecla). Les tecles i accions venen definides per enumeradors:

- Per tecles tenim els enumeradors **GLFW_KEY_X** on **X** és la tecla en qüestió. Per exemple **GFLW_KEY_ESCAPE** fa referència a la tecla escape.

- Per accions tenim els numeradors:
  - **GLFW_KEY_PRESS**: Captura quan una tecla ha sigut pressionada.
  - **GLFW_KEY_RELEASE**: Captura quan una tecla deixa de ser pressionada.
  - **GLFW_KEY_REPEAT**: Captura quan una tecla és pressionada múltiples vegades.

L'altre opció per capturar l'entrada de teclat es definir un mètode propi, que rep com a paràmetre la finestra, i fent ús de la funció **glfwGetKey** que rep com a paràmetre la finestra, el numerador amb la tecla que es vol detectar i, retorna l'acció. Com **GLFW** està fet en C, implementar la funcionalitat dels **callback** en una classe porta dificultats. Aquesta és una bona opció per capturar l'entrada de teclat dins d'una classe.

Per altra banda, per capturar els esdeveniments de ratolí també disposem de les dues opcions, fer servir el **callback** que ens proporciona **GLFW** o definir un mètode propi amb la finestra. Similar a com obtenim l'entrada de teclat obtenim la captura de botons del ratolí, amb el mètode **glfwGetMouseButton** podem detectar si s'ha pressionat un botó del ratolí i amb el mètode **glfwGetCursorPos** podem obtenir la posició d'aquest en pantalla.

Finalment **GLFW** també ens deixa capturar la captura del scroll del ratolí amb un **callback** i calcular el temps delta, que ens diu quant de temps en milisegons ha passat entre diferents frames.

7.6 GLM

OpenGL Maths o **GLM** és una llibreria per realitzar càlculs complexos de forma fàcil i intuïtiva. Principalment, treballem amb vectors, matrius, quaternions i angles i s'encarrega de tots els càlculs i transformacions sobre aquests.

A continuació fem un resum dels diferents tipus i les operacions més destacades que podem realitzar:

Vectors

GLM ofereix vectors de 2, 3 o 4 elements definits de la següent manera:

- **glm::vec2(x,y)**
- **glm::vec3(x,y,z)**
- **glm::vec4(x,y,z,w)**

Sobre aquest tipus podem realitzar diferents operacions, en destaquem les següents:

- **length** ens retorna la mida del vector.
- **cross** que té com a paràmetre dos vectors de 3 elements i realitza el producte vectorial entre aquests.
- **dot** que te com a paràmetre dos vectors de 3 elements i realitza el producte escalar entre aquests.
- **normalize** converteix tots els elements del vector en un valor entre 0..1.
- **distance** que ens diu la distància entre dos punts.

Matrius

Per altra banda, **GLM** ofereix matrius de fins a 4x4 elements. A **OpenGL** nomes fem servir aquestes per realitzar transformacions geomètriques i és defineixen com a **glm::mat4**.

Les operacions que destaquem sobre aquest tipus són les següents:

- **translate** que construeix una matriu de 4x4 a la que se li ha aplicat una translació que ens ve donada com un vector de 3 elements.

- **rotate** que construeix una matriu de 4x4 amb un nombre real que determina l'angle de rotació i un vector de 3 elements que determina l'axis on es realitza aquesta rotació.
- **scale** que construeix una matriu de 4x4 a la que se li ha aplicat un escalat a partir d'un vector de 3 elements.
- **prespective** que donat un angle que especifica el camp de visió, dos reals que representen la resolució de la pantalla, un real que especifica la distància al pla de visió més proper i un real que especifica la distància al pla de visió més llunyà, construeix una matriu de 4x4 que ens dona el camp de visió.
- **lookAt** que, donada una posició, una direcció i un vector que apunta cap amunt, construeix una matriu 4x4 que ens permet rotar la càmera.

Angles

A **GLM** representem els angles com a reals i permet totes les operacions trigonomètriques bàsiques, com **sinus**, **cosinus**, **tangent**... també permet convertir de radians a graus i viceversa.

Quaternions

Els quaternions en el context actual s'utilitzen per representar rotacions en un espai tridimensional. **GLM** ens permet definir aquests amb **glm::quat(x,y,z,w)**. Destaquem les següents operacions:

- **angleAxis** que, donat un angle i un vector de 3 elements representant l'axis de rotació, ens construeix un quaternió.
- **eulerAngles** que, donat un quaternió, ens retorna un vector de 3 elements amb **pitch**, **yaw** i **roll**.
- **mat4_cast** que, donat un quaternió ens retorna aquest representat en una matriu de rotació de 4x4.
- **quat_cast** que, donada una matriu de rotació de 4x4, ens retorna un quaternió que representa aquesta rotació.

7.7 FastNoise Lite

FastNoise Lite [Auburn] és una llibreria **Open Source** que conté diferents algoritmes per la generació de **soroll**, que consisteix a construir imatges on cada *píxel* és un valor *pseudoaleatori* entre 0..255. Veure figura 7.8:

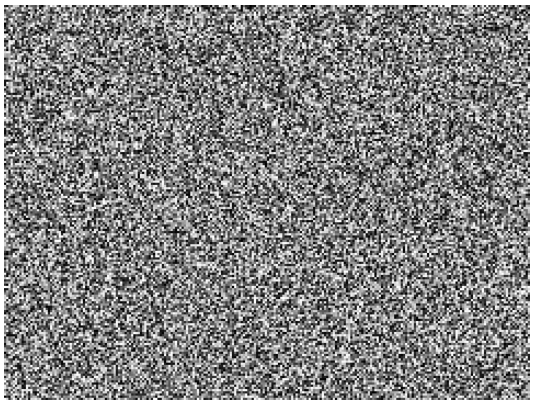


Figura 7.8: Soroll perlin

Aquesta llibreria ens permet parametritzar els algoritmes de soroll de forma fàcil i intuïtiva. Expliquem a continuació les diferents categories [Kafitz 2020]:

- **General**: Ens permet escollir l'algoritme de soroll, la llavor que determina la pseudoaleatorietat i la freqüència de repetició. Veure figura 7.9

Figura 7.9: OpenSimplex 2

- **Fractal**: Algoritme que s'aplica sobre l'algoritme de soroll seleccionat i ens permet modificar la freqüència i l'amplitud. Destaquem els següents paràmetres. Veure figura 7.10:
  - **Octaves**: Determina el nombre d'iteracions que el codi ha de fer, proporcionant més detall a cada iteració.
  - **Lacunarity**: Millora la qualitat superposant sorolls de diferents mides.
  - **Amplitude**: S'aplica sobre **lacunarity** i afecta la longitud de la desviació entre sorolls.
  - **Gain**: Es refereix a la constant que es suma a cada iteració d'un **octave**.

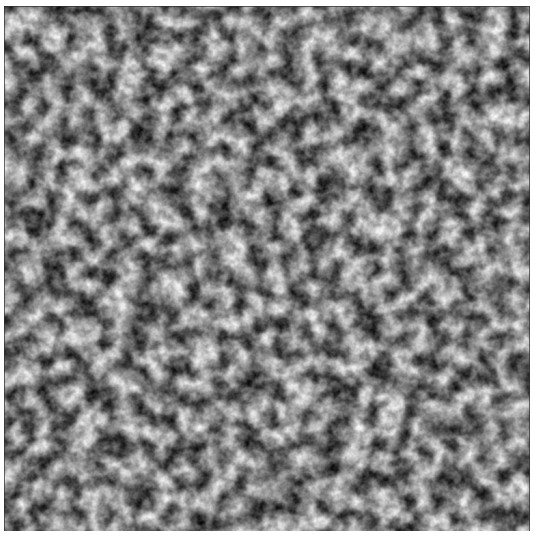


Figura 7.10: OpenSimplex 2 amb Fractal Brownian Motion

- **Cellular**: Algoritme que distribueix diferents punts en espais organitzats com a xarxes, escull un punt aleatori, i per cada localització agafa la distància al punt més proper i fa servir això per controlar la informació sobre el color. Veure figura 7.11

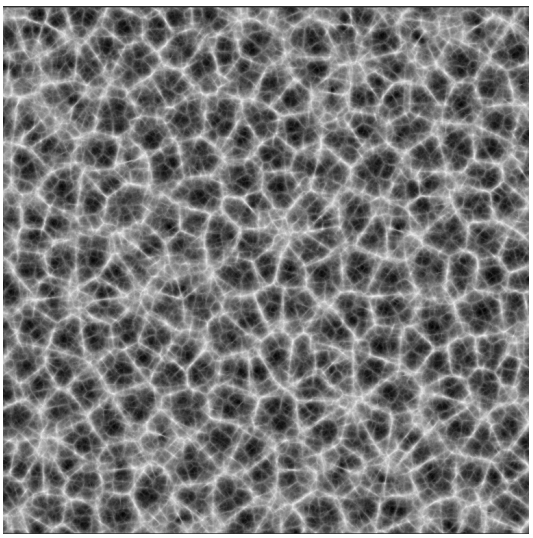


Figura 7.11: OpenSimplex 2 amb Cellular

- **Domain Warp**: Permet distorsionar el soroll creant patrons més naturals. Veure figura 7.12

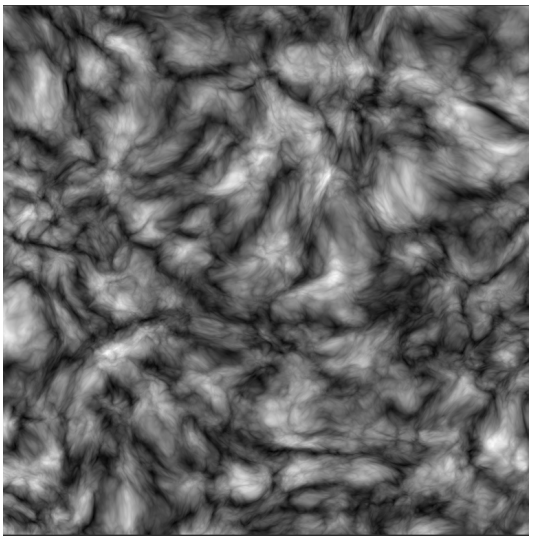


Figura 7.12: OpenSimplex 2 amb Cellular i Domain Warp

7.8 ImGUI

ImGUI [Cornut] és una llibreria exclusiva per **C++** que permet construir de forma fàcil una interfície gràfica. El principal punt a favor d'aquesta llibreria és que permet construir la interfície a qualsevol part del codi, cosa que permet estalviar-nos connectors entre la interfície i la resta de classes.

Figura 7.13: Exemple d'interfície amb ImGUI

El funcionament és simple i el resumim en els següents passos:

- Inicialment creem un nou context fent ús de **ImGUI::CreateContext**.
- Seguidament, associem la finestra creada amb **GLFW** amb **ImGui_ImplGlfw_InitForOpenGL** i li diem que funcioni sota **OpenGL** amb **ImGui_ImplOpenGL3_Init**.

- Tot seguit creem un nou frame que és on aniran a parar tots els elements de la interfície amb **ImGui_ImplOpenGL3_NewFrame()**, seguit de **ImGui_ImplGlfw_NewFrame()** i seguit de **ImGui::NewFrame()**.
- Fet tot això ja podem crear un nou element per la interfície:
  - **ImGUI::Begin** que rep com a paràmetre el títol/identificador, marca l'inici d'aquest element de la interfície.
  - Seguidament podem crear diferents modificadors dins d'aquest element. En mencionem els més destacats:
    - **Sliders**: Barres de control lliscant que permeten seleccionar valors numèrics:

Figura 7.14: Barra de control lliscant

    - **Botons**: Botó que permet realitzar accions:

Figura 7.15: Botó

    - **Seleccionador de colors**: Permet seleccionar colors (RGB):

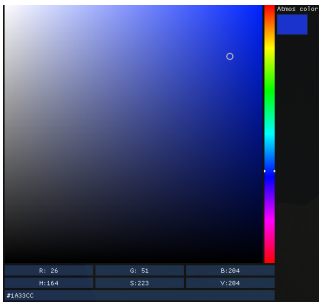


Figura 7.16: Seleccionador de colors

  - Per finalitzar l'element de la interfície ho fem amb **ImGui::End()**.
- Una vegada creats tots els elements de la interfície, cridem a **ImGUI::Render** i aquests apareixen per pantalla.
- Per finalitzar cridem **ImGui_ImplOpenGL3_Shutdown()**, **ImGui_ImplGlfw_Shutdown()** i **ImGui::DestroyContext()** per alliberar memòria i destruir el context creat.

7.9 RapidXML

Per la representació de l'escena escollim el llenguatge **XML** ja que proporciona flexibilitat a l'hora d'afegir/eliminar atributs dels objectes, i és un llenguatge relativament ràpid de parsejar. **RapidXML** és un parser relativament ràpid, **Open Source** i fet per **C++**.

Aquesta llibreria incorpora diferents objectes amb els quals treballarem, on destaquem 3:

- **xml_document**: Carrega tot el document **XML** a memòria.
- **xml_node**: Carrega un node de l'arbre de parsing, que pot ser l'arrel, branca o fulla.
- **xml_attribute**: Guarda el valor que tenen els nodes.

Per poder obrir, llegir i així carregar la nostra escena a memòria, definint els diferents objectes que la formen, realitzem el següent procediment:

- Inicialment carreguem tot el document com a text en un vector de caràcters.
- Amb el xml_document definit, cridem a la funció **parse**, on li passem el vector de caràcters.
- Obtenim l'arrel de l'arbre amb el document i la funció **first_node** que rep com a paràmetre l'identificador d'aquest node.
- Amb l'arrel de l'arbre ja podem accedir a la resta de nodes amb la funció **first_node**, que rep com a paràmetre l'identificador del node que estem cercant.
- Per obtenir un atribut del node cridem a la funció **first_attribute** que retorna un **xml_attribute** on sobre aquest, cridem la funció **value** que torna el valor en cadena de caràcters.

També ens interessa poder guardar valors que hem modificat en temps d'execució. El procediment és molt semblant a llegir: Obrim i carreguem el document, cerquem el node que volem modificar, obtenim l'atribut i a la funció **value** li passem com a paràmetre el valor que volem modificar.

CAPÍTOL 8

Anàlisi i disseny del sistema

La interacció de l'usuari amb l'aplicació consta de dues parts, per afegir planetes a l'escena o afegir una atmosfera, es fa des d'un fitxer **XML** que permet configurar l'escena. Per altra banda, també hem introduït una interfície que permet modificar fàcilment els diferents atributs del planeta, atmosfera, il·luminació i guardar els canvis al fitxer **XML**.

Distingim tres parts de la interfície:

- **Finestra amb atributs de l'atmosfera**: Permet modificar els diferents atributs de l'atmosfera. Veure figura 8.1

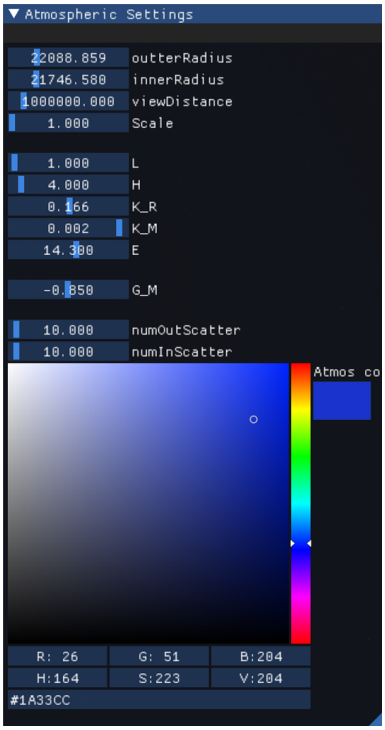


Figura 8.1: Finestra amb atributs de l'atmosfera

- **Finestra amb atributs del planeta**: Permet modificar la mida, qualitat i terreny del planeta. Veure figura 8.2

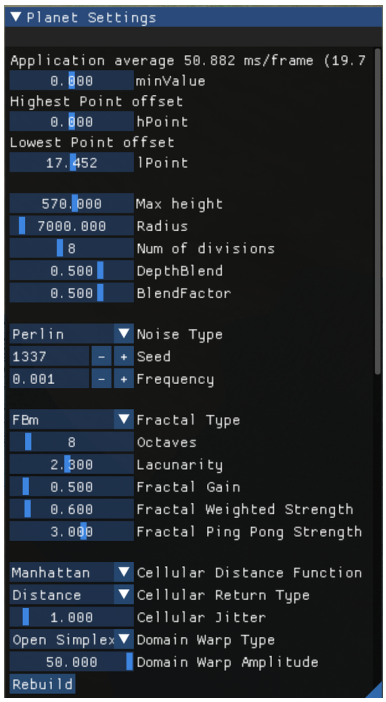


Figura 8.2: Finestra amb atributs del planeta

- **Finestra modificació de l'escena:** Permet modificar la direcció de la llum i guardar els canvis realitzats a l'escena.

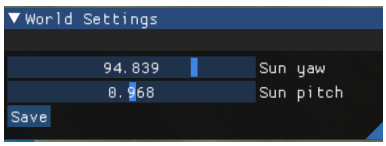


Figura 8.3: Finestra modificació de l'escena

8.1 Diagrama i fitxes de cas d'ús

Al diagrama de casos d'ús podem distingir l'actor principal **usuari** que pot realitzar les diferents tasques de modificació de l'escena, ja sigui via edició del fitxer **XML** o interactuant amb els elements de la interfície. Veure figura 8.4

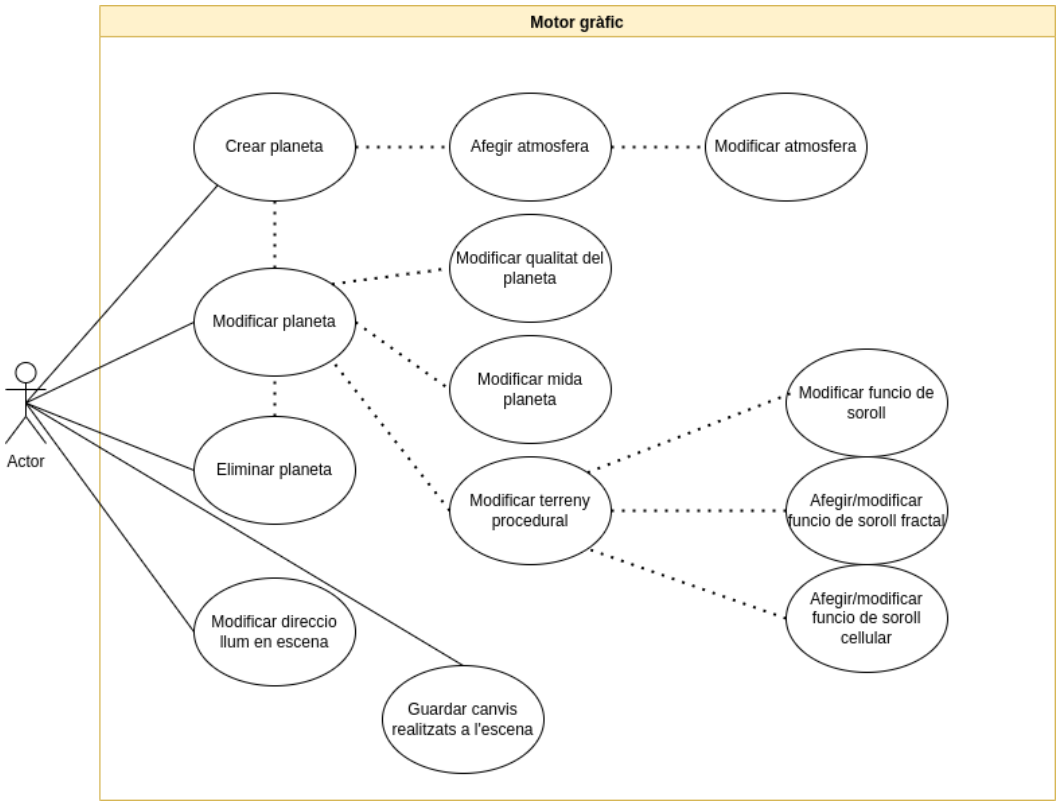


Figura 8.4: Diagrama de cas d'ús

Seguidament fem un resum de les principals tasques que pot realitzar l'usuari amb l'aplicació mitjançant les següents fitxes de cas d'ús:

Cas d'ús:	**Crear planeta**
Versió:	Visió anàlisi
Descripció:	Afegeix un planeta a l'escena
Actors:	Usuari
Precondició:	No existeix planeta amb identificador seleccionat
Flux principal:	1. Definir paràmetres: identificador, radi, numero de subdivisions, si té aigua o té atmosfera 2. Definir paràmetre posició 3. Definir paràmetres de funció de soroll 4. SI té atmosfera, definir paràmetres de atmosfera 5. Definir llistat de textures 6. Definir biomes 7. Per cada bioma, definir textures amb restricció per alçada
Postcondició:	S'ha afegit el planeta a l'escena

Cas d'ús:	**Afegir atmosfera**
Versió:	Visió anàlisi
Descripció:	Afegeix una atmosfera a un planeta ja existent
Actors:	Usuari
Precondició:	Hi ha mínim un planeta a l'escena
Flux principal:	1. Modificar atribut planeta per indicar que aquest disposa d'atmosfera 2. Definir paràmetres com mida, lluminositat, densitat i intensitat de la llum 3. Definir el color de l'atmosfera
Postcondició:	S'ha afegit una atmosfera al planeta

Cas d'ús:	**Modificar atmosfera**
Versió:	Visió anàlisi
Descripció:	Modifica una atmosfera d'un planeta ja existent
Actors:	Usuari
Precondició:	Escena conte planeta amb atmosfera
Flux principal:	1. A la interfície seleccionar finestra amb atributs de l'atmosfera 2. Modificar atributs de l'atmosfera 3. Guardar canvis a l'escena
Postcondició:	S'ha modificat l'atmosfera del planeta

Cas d'ús:	**Modificar planeta**
Versió:	Visió anàlisi
Descripció:	Modifica un planeta ja existent
Actors:	Usuari
Precondició:	Escena conte planeta
Flux principal:	1. A la interfície seleccionar finestra amb atributs del planeta 2. Modificar planeta 2.1. Modificar qualitat del planeta 2.2. Modificar mida del planeta 2.3. Modificar terreny procedural 3. Reconstruir planeta 4. Guardar canvis realitzats a l'escena
Postcondició:	S'ha modificat el planeta

Cas d'ús:	**Modificar terreny procedural**
Versió:	Visió anàlisi
Descripció:	Modifica el terreny d'un planeta
Actors:	Usuari
Precondició:	Escena conte planeta
Flux principal:	1. A la interfície seleccionar finestra amb atributs del planeta 2. Modificar terreny del planeta 2.1. Modificar funció de soroll 2.2. Afegir/Modificar funció de soroll fractal 2.3. Afegir/Modificar funció de soroll cellular 3. Reconstruir planeta 4. Guardar canvis realitzats a l'escena
Postcondició:	S'ha modificat el terreny del planeta

Cas d'ús:	**Modificar direcció llum en escena**
Versió:	Visió anàlisi
Descripció:	Modifica la direcció de la llum a l'escena
Actors:	Usuari
Precondició:	Escena conte elements per visualitzar els canvis de llum
Flux principal:	1. A la interfície seleccionar finestra modificació de l'escena 2. Modificar els valors per canviar la direcció de la llum 3. Guardar canvis realitzats a l'escena
Postcondició:	S'ha modificat la direcció de la llum

8.2 Diagrama de classes

Definim l'aplicació amb un total de 12 classes amb les seves relacions, que s'encarregaran de carregar una escena i mostrar-la per pantalla. El diagrama de classes, veure figura 8.5, ens mostra un resum dels mètodes i atributs que té cada classe i la relació entre aquestes.

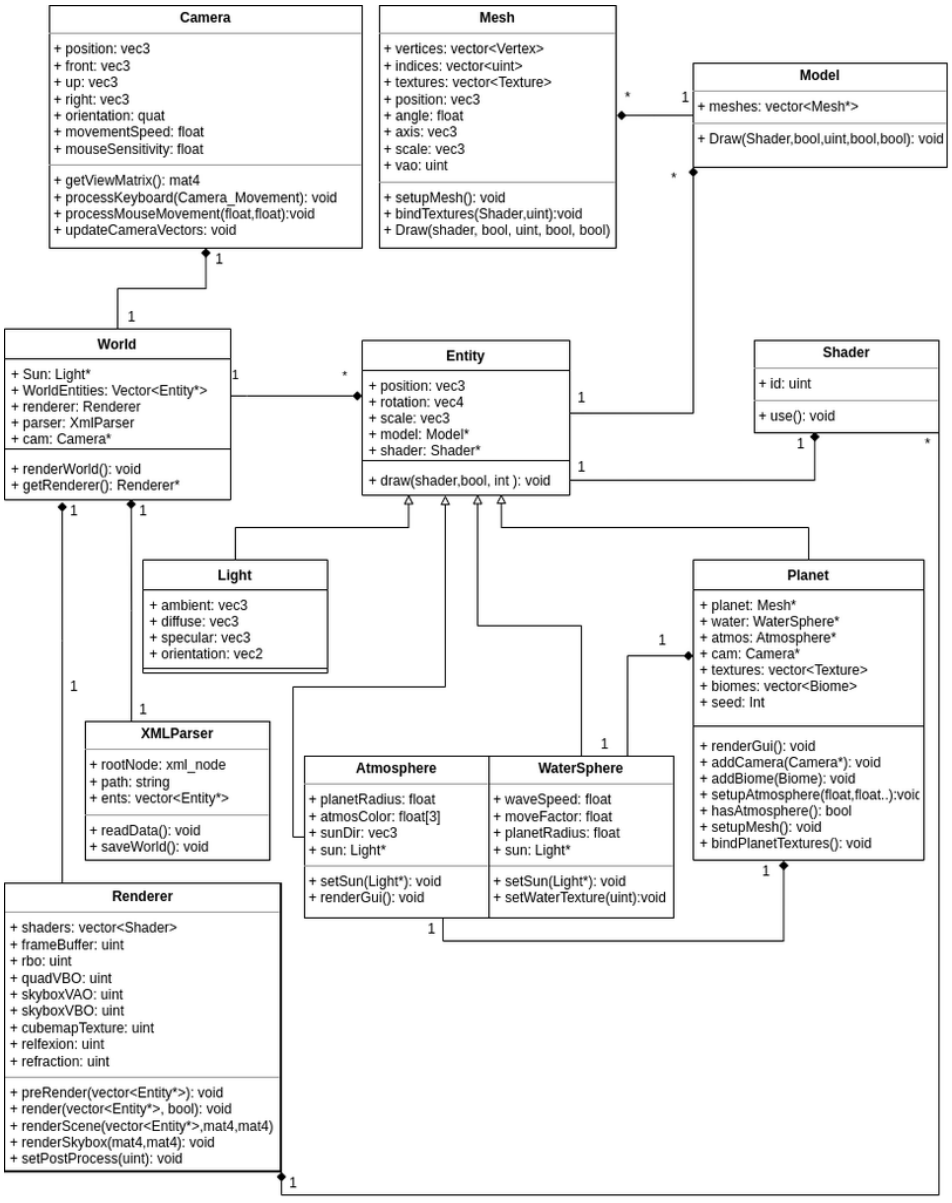


Figura 8.5: Diagrama de classes

8.3 Descripció de les classes

8.3.1 Classe World

8.3.1.1 Descripció

La classe **World** s'encarrega de carregar l'escena i renderitzar-la. Per fer això, inicialment crea un objecte de tipus **Renderer**, un objecte de tipus **XMLParser** i carrega l'escena **XML** en un llistat d'**Entity**.

Tot seguit envia al **Renderer** el llistat d'entitats per renderitzar-les i crea una petita interfície que permet modificar la direcció de la llum i guardar els canvis realitzats a l'escena.

Finalment, quan l'objecte es destrueix, allibera memòria, eliminant tots els elements del vector d'**Entity**, i eliminant l'objecte **XMLParser** i l'objecte **Renderer**.

8.3.1.2 Mètodes i atributs

Atributs

- **sunYaw**: Determina l'angle horitzontal de la llum direccional global.
- **sunPitch**: Determina l'angle vertical de la llum direccional global.
- **sun**: Objecte de tipus **Light** que descriu la llum direccional global.
- **parser**: Objecte de tipus **XMLParser** que llegeix les dades del fitxer **XML**.
- **renderer**: Objecte de tipus **Renderer** que mostra els objectes de l'escena per pantalla.
- **ents**: Llistat de **Entity** que ens genera el **XMLParser** per renderitzar.

Constructor i mètodes de classe

- **World**(const char * scenePath, const char * skyBoxPath, unsigned int scrWidth, unsigned int scrHeight, Camera *camera)
  - **Descripció**: Donada una cadena de caràcters que representa la ruta a l'escena, una cadena de caràcters que representa la ruta a la imatge del fons de l'escena, dos enters que donen l'amplada i alçada de la finestra i la càmera, genera els diferents atributs que componen la lògica de l'escena.

- **void renderWorld**(bool wireframe)

  - **Descripció**: Envia al **Renderer** el vector d'**Entity** per renderitzar amb el paràmetre **wireframe**, que determina si volem que mostrin els triangles dels diferents polígons.

    Aquesta funció també genera la interfície que permet modificar la direcció de la llum i guardar els canvis realitzats a l'escena.

8.3.2 Classe Renderer

8.3.2.1 Descripció

La classe **Renderer** s'encarrega de mostrar els diferents elements que componen l'escena per pantalla. També s'encarrega d'aplicar els efectes de post-processat, crear els diferents **FrameBuffers** pels efectes de reflexió i generar el *cel* a partir d'una textura cúbica.

Per fer això rep el vector d'**Entity** de l'objecte World, carrega els diferents Shaders i genera els diferents **FrameBuffers** que permetran crear les reflexions de l'aigua i aplicar efectes de post-processat.

Finalment, quan l'objecte es destrueix, s'eliminen tots els **FrameBuffers** i Shaders generats.

8.3.2.2 Mètodes i atributs

Atributs

- **shaders**: Llistat dels diferents Shaders que componen l'escena.
- **framebuffer**: **FrameBuffer** principal on es renderitza l'escena.
- **textColorBuffer**: Textura on es renderitza el **FrameBuffer** anterior.
- **quadVAO** i **quadVBO**: **Vertex Attribute Object** i **Vertex Buffer Object** que formen un rectangle posicionat frontalment davant de la càmera on s'apliquen els efectes de post-processat.
- **skyboxVAO** i **skyboxVBO**: Formen una caixa *infinita* a la que s'aplica una textura cúbica creant un efecte de cel infinit.
- **cubemapTexture**: Textura cúbica que s'aplica sobre l'estructura geomètrica anterior.
- **reflexionFBO**: **FrameBuffer** d'una qualitat inferior al de l'escena original on guardem la textura per crear l'efecte de reflexió a l'aigua.
- **reflexion**: Textura on es guarda el **FrameBuffer** anterior.
- **postProcessPath**: Llistat de la ruta als diferents efectes de post-processat que apliquem sobre el **quadVAO**.
- **camera**: Càmera principal que permet visualitzar i interactuar amb l'escena.

Constructor i mètodes de classe

- **Renderer**(unsigned int scrWidth, unsigned int scrHeight, Camera *camera, const char* skyboxPath)
  - **Descripcio**: Donat dos enters que determinen l'amplada i alçada de la pantalla, la **Càmera** de l'escena i la ruta a la textura del cel, crea els diferents **Shaders** que componen l'escena, genera els diferents **FrameBuffers** i **VAO**'s pels efectes de post-processat i reflexions, i crea l'efecte de *cel* infinit.
- **void setupFrameBuffer**()
  - **Descripció**: Defineix l'estructura quadrada on s'aplicara el **FrameBuffer** principal i els efectes de post-processat i genera i associa els diferents **FrameBuffers** a les textures corresponents.
- **void setupSkyBox**(const char * path)
  - **Descripció**: Crea l'estructura en forma de caixa i carrega la textura cúbica a partir de la ruta **path**.
- **void preRender**(vector<Entity*> worldEnts)
  - **Descripció**: S'assegura que totes les Entities tenen tots els atributs necessaris per poder mostrar-les per pantalla.
- **void render**(vector<Entity*> worldEnts,bool wireframe)
  - **Descripció**: Gestiona els diferents **FrameBuffers** que formen l'escena, renderitzant tots els elements diverses vegades per poder generar les diferents textures necessàries per aplicar els diferents efectes. També genera les matrius de **View** i **Projection** a partir de la càmera i la resolució de pantalla.
- **void renderScene**(vector<Entity*> worldEnts,bool wireframe, glm::mat4 view, glm::mat4 projection,float width, float height, bool drawEffects)
  - **Descripció**: Crea la pantalla amb la resolució determinada pels paràmetres **width**(amplada) i **height**(alçada), neteja els diferents **buffers** d'**OpenGL** i renderitza les Entities i el *cel* infinit.

- **void drawEntities**(std::vector<Entity*> worldEnts, glm::mat4 view, glm::mat4 projection, Shader shader, bool drawEffects)
  - **Descripció**: Posa les diferents **Uniforms** de les matrius **view** i **projection**, necessàries pel Shader i renderitza les Entities.
- **void renderSkybox**(glm::mat4 view, glm::mat4 projection)
  - **Descripció**: Modifica el paràmetre de profunditat d'**OpenGL** per tal que qualsevol objecte es superposi al *cel* infinit, posa les **uniforms view** i **projection** al **shader** del cel i dibuixa l'estructura en forma de caixa aplicant la textura cúbica.

8.3.3 Classe Camera

8.3.3.1 Descripció

La classe **Camera** permet a l'usuari interaccionar amb l'escena mitjançant el teclat i el ratolí. Inicialment, quan es crea la càmera, inicialitzem els diferents atributs i processem les diferents entrades de ratolí i teclat.

La interacció que fem amb el teclat determina la posició de la càmera i l'orientació frontal tal que:

- **W**: Permet moure'ns endavant.
- **S**: Permet moure'ns enrere.
- **A**: Permet moure'ns lateralment a l'esquerra.
- **D**: Permet moure'ns lateralment a la dreta.
- **Q**: Permet rotar frontalment a l'esquerra.
- **E**: Permet rotar frontalment a la dreta.

La interacció amb el ratolí consisteix en pressionar el botó dret sobre l'escena i moure el ratolí verticalment/horitzontalment per modificar l'angle lateral/vertical.

8.3.3.2 Mètodes i atributs

Atributs

- **position**: Determina la posició de la càmera a l'escena.
- **front**: Determina el vector que apunta frontalment amb origen a la càmera.
- **right**: Determina el vector que apunta lateralment amb origen a la càmera.
- **up**: Determina el vector que apunta verticalment amb origen a la càmera.
- **orientation**: Quaternió que descriu l'orientació de la càmera a l'escena.
- **movementSpeed**: Real que determina la velocitat a la qual ens movem per l'escena.
- **mouseSensitivity**: Real que determina la sensibilitat del ratolí quan ens orientem per l'escena.

- **Pitch**,**Yaw** i **Roll**: Registren les variacions a l'angle a partir del moviment del ratolí o les tecles **Q** o **E**.

Constructor i mètodes de classe

- **Camera**(float posX, float posY, float posZ, float upX, float upY, float upZ, float yaw, float pitch)
  - **Descripció**: Construeix la càmera amb una posició i orientació determinada pels paràmetres d'entrada.
- **void updateCameraVectors()**
  - **Descripció**: Crea el quaternió que descriu l'orientació de la càmera a partir dels atributs **pitch**, **yaw** i **roll** i calcula els nous valors pels vectors **front**, **right** i **up**.
- **glm::mat4 GetViewMatrix**()
  - **Descripció**: Retorna la matriu que descriu la posició i orientació de la càmera.
- **void Camera::ProcessKeyboard**(Camera_Movement direction, float deltaTime)
  - **Descripció**: Donada una direcció i un real que determina el temps que ha passat entre *frames*, registra les entrades de teclat i modifica la posició/orientació acorde a la tecla pressionada, l'atribut **movementSpeed** i el temps entre *frames*.
- **void ProcessMouseMovement**(float xoffset, float yoffset)
  - **Descripció**: Donat dos reals que descriuen la variació horitzontal i vertical respectivament del ratolí, actualitza els angles **pitch** i **yaw**.

8.3.4 Classe Mesh

8.3.4.1 Descripció

La classe **Mesh** s'encarrega de guardar els diferents atributs que formen la geometria i color d'un objecte.

La geometria ens ve donada per un vector de **Vertex** i un vector d'**Indices** amb el que construirem el **VAO** de l'objecte.

Per altra banda el color ens ve donat per les textures que formen l'objecte. Cada **Mesh**, inicialment carregara totes les textures i associarà aquestes a un apuntador a memòria.

Finalment, quan acabem amb l'objecte, alliberem la memòria de la gràfica eliminant les textures i els elements que formen la geometria.

8.3.4.2 Mètodes i atributs

Atributs

- **vertices**: Llistat de **Vertex** on cada vèrtex conté els següents atributs:
  - **position**: Vector de 3 elements que determina la posició del punt.
  - **normal**: Vector de 3 elements que determina el vector normal orientat perpendicularment a la superfície del triangle que forma aquest vèrtex.
  - **texCoords**: Vector de 2 elements que determina les coordenades de textura que corresponen al vèrtex.
- **indices**: Llistat d'enters que determinen l'ordre dels vèrtexs per tal de reduir-ne el nombre.
- **textures**: Llistat format per l'estructura Texture que conté els següents atributs:
  - **id**: Enter que identifica la textura.
  - **path**: Cadena de caràcters amb la ruta on es troba aquesta textura.
  - **type**: Cadena de caràcters que ens diu de quin tipus de textura es tracta.
- **position**: Vector de 3 elements que determina la posició de la **Mesh**.

- **angle**: Real que determina la variació de l'angle de la **Mesh**.
- **axis**: Vector de 3 elements que determina els 3 angles de rotació del **Mesh**.
- **scale**: Vector de 3 elements que determina l'escalat de l'objecte en els tres eixos.
- **VAO**: Enter que determina el **Vertex Attribute Pointer** de la **Mesh**.

Constructor i mètodes de classe

- **Mesh**(std::vector<Vertex> vertices, std::vector<unsigned int> indices, std::vector<Texture> textures)
  - **Descripció**: Construeix una nova **Mesh** associant els diferents atributs als paràmetres d'entrada i inicialitzant la resta.
- **void setupMesh**()
  - **Descripció**: Construeix el **VAO**, **EBO** i **VBO** explicat a, **Capítol 7: OpenGL**.
- **void bindTextures**(Shader shader)
  - **Descripció**: Associa les textures de l'objecte al **shader** que rep per paràmetre.
- **void Draw**(Shader shader,bool wireframe, bool patches)
  - **Descripció**: Renderitza el **Mesh** amb el **shader** que rep per paràmetre amb l'opció de visualitzar els triangles (**wireframe**) o si l'objecte té la **Tesselation Shader** activa (**patches**).

8.3.5 Classe Model

8.3.5.1 Descripció

La classe **Model** permet carregar models en format **wavefront** (**.obj**) especificant la ruta. Aquests models poden estar formats per una o més **Mesh** i un llistat de textures. Per la versió final del projecte, aquesta classe no es fa servir, però va formar part de l'estudi inicial i vam decidir incloure una petita descripció.

8.3.6 Classe Shader

8.3.6.1 Descripció

La classe **Shader** s'encarrega de llegir, compilar i gestionar els **uniforms** dels diferents **shaders** que formen l'escena.

Aquesta classe té 2 formes d'inicialitzar-se, ja sigui llegint i compilant 2 parts de la *pipeline* amb el **Vertex** i **Fragment Shader** o 4 parts, introduint **Tesselation**.

8.3.6.2 Mètodes i atributs

Atributs

- **ID**: Enter que identifica al **Shader**.

Constructor i mètodes de classe

- **Shader**(const char* vertexPath, const char* fragmentPath)
  - **Descripció**: Donades les rutes de dos fitxers, que contenen el codi característic d'un **Vertex** i **Fragment Shader**, els compila i genera un identificador.
- **Shader**(const char* vertexPath, const char* fragmentPath, const char* tcsPath, const char *tesPath)
  - **Descripció**: Donades les rutes de quatre fitxers, que contenen el codi característic d'un **Vertex**, **TCS**, **TES** i **Fragment Shader**, compila aquests i genera un identificador.
- **void setFloat**(const std::string name, float value)
  - **Descripció**: Envia l'uniform amb nom *name* i valor de tipus **real** *value* al shader.

- **void setVec3**(const std::string name, glm::vec3 value)
  - **Descripció**: Envia l'uniform amb nom *name* i valor de tipus **vec3** *value* al shader.
- **void setMat4**(const std::string name,glm::mat4 value)
  - **Descripció**: Envia l'uniform amb nom *name* i valor de tipus **mat4** *value* al shader.

8.3.7 Classe Entity

8.3.7.1 Descripció

La classe **Entity** és una classe genèrica que descriu els diferents atributs i mètodes que han de tenir totes les classes que la implementen.

8.3.7.2 Mètodes i atributs

Atributs

- **position**: Vector de 3 elements que descriu la posició de l'entitat a l'escena.
- **rotation**: Vector de 4 elements que descriu els eixos i angle de rotació de l'entitat a l'escena.
- **scale**: Vector de 3 elements que determina l'escalat de l'entitat a l'escena.
- **model**: Model que descriu la geometria i color de l'entitat a l'escena. Per la generació de terreny aquest atribut s'ignora i es treballa amb una única Mesh.
- **shader**: Si l'entitat té un **shader** prefixat, es guarda en aquest atribut.
- **type**: Enter que determina el tipus de l'entitat.

Constructor i mètodes de classe

Al tractar-se d'una classe genèrica, no disposa de constructor, aquest s'implementa a les classes derivades. També obliga a les classes derivades a implementar el mètode **draw**.

8.3.8 Classe Light

8.3.8.1 Descripció

La classe **Light**, derivada de la classe **Entity**, s'encarrega de descriure la il·luminació global que rep l'escena. Per fer això guardem diferents atributs que descriuen el color, la intensitat de la llum, l'efecte especular, i l'orientació que té aquesta a l'escena.

8.3.8.2 Mètodes i atributs

Atributs

- **ambient**: Vector de 3 elements que descriu el color que han de tenir les zones fosques de l'escena.
- **diffuse**: Vector de 3 elements que descriu el color de la llum.
- **specular**: Vector de 3 elements que descriu el color de l'efecte especular de la llum.
- **orientation**: Vector de 2 elements que descriu els dos angles (**yaw** i **pitch** respectivament) que formen l'orientació de la llum a l'escena.

Constructor i mètodes de classe

- **Light**(glm::vec3 ambient, glm::vec3 diffuse, glm::vec3 specular,glm::vec2 orientation)
  - **Descripció**: Crea una nova llum a partir dels paràmetres d'entrada.
- **void draw**(Shader shader)
  - **Descripció**: S'encarrega d'actualitzar les diferents **uniforms** que componen la llum.
- **glm::vec3 getDirVector**()
  - **Descripció**: Construeix un vector direccional de 3 elements a partir dels angles **yaw** i **pitch**.

8.3.9 Classe Planet

8.3.9.1 Descripció

La classe **Planet**, derivada de la classe **Entity**, descriu el terreny procedural esfèric que caracteritza un planeta i les diferents textures i efectes que el componen.

Inicialment, creem una esfera d'un radi determinat a partir de subdividir un cub **n** vegades, on a cada subdivisió, incrementem el nombre total de **vèrtex** i en conseqüència, la qualitat final de l'objecte. Veure figura 8.6

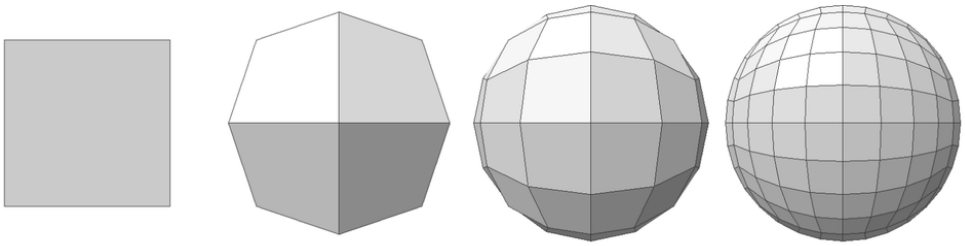


Figura 8.6: Proces de subdividir un cub en una esfera [Ahn 021]

A mesura que anem creant els **vèrtexs** anem aplicant la funció de soroll amb els paràmetres seleccionats i passem d'una superfície esfèrica plana, a una superfície *pertorbada*, donant l'efecte de terreny.

Per l'aplicació de les textures sobre el terreny, fem ús d'una estructura de dades **Biome**, que determina el rang de valors que representen la latitud mínima i màxima on s'aplicarà aquest bioma. Per cada bioma disposem d'un llistat de textures, on per cada textura tenim l'identificador, i un rang de valors que representen l'alçada mínima i màxima on s'aplicarà la textura.

Finalment per crear l'efecte atmosfèric i aquàtic, disposem de 2 objectes de tipus **Atmosphere** i **Watersphere**.

8.3.9.2 Mètodes i atributs

Atributs

- **biomes**: Llistat d'elements de tipus **Biome**.
- **planet**: Objecte de tipus **Mesh** que guarda la geometria i textures del planeta.
- **skyDome**: Objecte de tipus **Atmosphere**.

- **waterSphere**: Objecte de tipus **Watersphere**.
- **radius**: Real que determina el radi del planeta.
- **nSeg**: Enter que determina el nombre de subdivisions que s'han aplicat sobre la geometria.
- **maxHeight**: Real que determina l'alçada màxima a la qual pot arribar el terreny.

A més conte tots els atributs que permeten configurar **FastNoise Lite** explicats a **Capítol 7: FastNoise Lite**.

Constructor i mètodes de classe

- **Planet**(float radius, int nSeg, bool hasAtmos, float maxHeight...
  - **Descripció**: Assigna al planeta els atributs llegits pel **XMLParser**.
- **void setupMesh**()
  - **Descripció**: Construeix la geometria de l'objecte a partir d'aplicar la funció de soroll a una esfera creada subdividint un cub **n** vegades.
- **void bindPlanetTextures**()
  - **Descripció**: Carrega a la **GPU** les textures del planeta.
- **void setupEffects**(float atmosRadius, float kr, float km...)
  - **Descripció**: Crea l'objecte de tipus **Atmosphere** i l'objecte de tipus **Watersphere** amb els atributs llegits de l'objecte **XMLParser**.
- **void renderGUI**()
  - **Descripció**: Crea la interfície gràfica que permet modificar els atributs que formen la geometria del planeta.
- **void draw**(Shader shader)
  - **Descripció**: Aplica al **shader** les diferents **uniforms** per definir els biomes i crida al mètode **Draw** de l'atribut **planet** per dibuixar la geometria.

8.3.10 Classe Atmosphere

8.3.10.1 Descripció

La classe **Atmosphere** s'encarrega de crear l'efecte atmosfèric. Per fer això, descriu una esfera que envolta el planeta i aplica sobre aquesta un efecte mitjançant un **shader**.

La descripció dels atributs d'aquesta classe consisteix a entendre la funció de fase que descriu la dispersió de la llum en **Rayleigh** i **Mie** explicats a continuació [O'Neil 2005]:

- **Rayleigh**: Causat per petites molècules a l'aire, dispersa la llum en ones més curtes, potenciant principalment el color blau, seguit del verd i seguit del vermell.
- **Mie**: Causat per molècules més grans, tendeix a dispersar la llum en ones equitativament.

8.3.10.2 Mètodes i atributs

Atributs

- **k_m**: Real que determina el factor de dispersió de la llum **Mie**.
- **k_r**: Real que determina el factor de dispersió de la llum **Raylieg**.
- **H**: Real que determina la densitat de l'atmosfera en funció de l'alçada.
- **L**: Real que determina la lluminositat interna de l'atmosfera.
- **E**: Real que determina la quantitat de llum solar que absorbeix l'atmosfera modificant el color.
- **g_m**: Real que afecta la simetria de la dispersió.
- **numInScatter**: Enter que determina el nombre d'iteracions per calcular la dispersió interna.
- **numOutScatter**: Enter que determina el nombre d'iteracions per calcular la dispersió externa.
- **color**: Vector de 3 elements que determina el color de l'atmosfera.
- **innerRadius**: Real que determina el radi intern de l'atmosfera.
- **outterRadius**: Real que determina el radi extern de l'atmosfera.

Constructor i mètodes de classe

- **Atmosphere**(float planetRadius, float atmosRadius, Camera *cam, float kR, float kM...)
  - **Descripció**: Construeix una atmosfera assignant els paràmetres d'entrada llegits pel **XMLParser** als atributs de la classe i construint una esfera que envolta el planeta de radi **atmosRadius**.
- **void draw**(Shader shader)
  - **Descripció**: Renderitza l'atmosfera enviant les **uniforms** necessàries al **shader** i configurant la transparència de l'objecte.
- **void renderGUI**()
  - **Descripció**: Crea la interfície que permet modificar els atributs de l'atmosfera en temps d'execució.

8.3.11 Classe Watersphere

8.3.11.1 Descripció

La classe **Watersphere** genera una esfera interna a la qual s'aplica un conjunt de textures i efectes simulant àrees aquàtiques del planeta [ThinMatrix 2015].

A aquesta esfera s'apliquen dos efectes clau. A una distància llunyana vista des de l'espai, s'aplica una textura amb patró de repetició i efecte especular. Veure figura 8.7

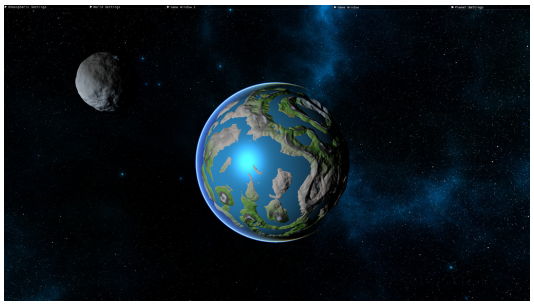


Figura 8.7: Efecte aigua des de l'espai

A una distancia més propera, s'aplica un efecte de distorsió, simulant el moviment de l'aigua i aplicant una textura capturada d'un **FrameBuffer** on hem renderitzat l'escena en menor qualitat, ja que aplicarem un efecte de dispersió sobre aquesta, i des d'un angle diferent. Veure figura 8.8

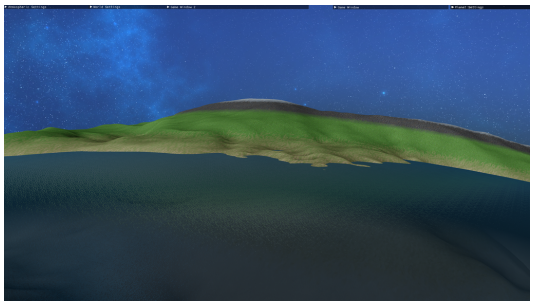


Figura 8.8: Efecte aigua des del planeta

8.3.11.2 Mètodes i atributs

Atributs

- **waveSpeed**: Real que determina l'increment de velocitat del moviment de les onades.
- **moveFactor**: Real entre 0..1 que acumula el desplaçament de l'aigua en funció de **waveSpeed** i el **deltaTime** de la **Camera**.
- **waterMesh**: Objecte de tipus **Mesh** que guarda la geometria de l'objecte i les textures que el formen.
- **cam**: Objecte de tipus **Camera** que ens permet obtenir el **deltaTime** i la posició d'aquesta per determinar la distància a la que ens trobem del cos aquàtic.
- **waterShader**: **Shader** que s'encarrega de crear els diferents efectes que componen l'aigua.

Constructor i mètodes de classe

- **WaterSphere**(float planetRadius, float waterRadius, Camera *cam, glm::vec3 position)
  - **Descripció**: S'encarrega de construir l'objecte de tipus **Mesh** i assignar els paràmetres als diferents atributs.
- **void draw**(Shader shader)
  - **Descripció**: Configura la transparència de l'objecte, envia al **shader** les **uniforms** necessàries per a renderitzar l'objecte i actualitza el moviment de les onades.

8.3.12 Classe XMLParser

8.3.12.1 Descripció

La classe **XMLParser** s'encarrega de llegir el fitxer **XML** que descriu la nostra escena. Per fer això fem ús de la llibreria **rapidXML** explicada en detall a **Capítol 7: RapidXML**.

La idea és, donat un fitxer en format **XML** amb un node arrel anomenat **Scene**, llegir i carregar els atributs dels diferents objectes que componen l'escena. A continuació destaquem els nodes més importants:

- **Camera**: Llegeix la **Camera** de l'escena amb atributs que descriuen la posició i orientació inicial.
- **Entity**: Que pot descriure un objecte de tipus **Light** o **Planet** amb els atributs pertinents.

8.3.12.2 Mètodes i atributs

Atributs

- **rootNode**: Objecte de tipus **xml_node** que guarda l'arrel de l'escena.
- **worldEnts**: Vector d'**Entities** que guarda els elements que formen l'escena.
- **cam**: Objecte de tipus **Camera** que guarda la càmera de l'escena.
- **sun**: Objecte de tipus **Light** que descriu la llum global a l'escena.

Constructor i mètodes de classe

- **XmlParser**(std::string path,Camera *cam)
  - **Descripció**: Donada una ruta **path**, obre i llegeix el document que es troba en aquesta i carrega els nodes principals.
- **void readData**()
  - **Descripció**: Recorre el sub-arbre d'entitats i crea els diferents objectes amb els atributs pertinents.
- **void saveWord**()
  - **Descripció**: Recorre l'arbre complet i guarda l'estat dels atributs dels elements que componen l'escena al fitxer **XML**.

CAPÍTOL 9

Implementació i proves

9.1 Programa principal

El programa principal de l'aplicació s'encarrega de crear la finestra on és renderitzara l'escena, crear la càmera i el món, gestionar les entrades de teclat i ratolí, i inicialitzar la interfície d'usuari.

9.1.1 Creació de la finestra

Per crear la finestra, fem ús de la següent funció, que s'encarrega de crear la finestra amb una resolució per defecte o en pantalla completa, iniciar les funcions de *callback* per capturar l'entrada de l'usuari i capturar el context d'**OpenGL**:

```
1  GLFWwindow * createWindow()
2  {
3
4      glfwInit();
5      glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4.5);
6      glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 4.5);
7      glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
8
9      GLFWwindow* window;
10     if (fullScreen)
11     {
12         GLFWmonitor* monitor = glfwGetPrimaryMonitor();
13         const GLFWvidmode* mode = glfwGetVideoMode(monitor);
14
15         glfwWindowHint(GLFW_RED_BITS, mode->redBits);
16         glfwWindowHint(GLFW_GREEN_BITS, mode->greenBits);
17         glfwWindowHint(GLFW_BLUE_BITS, mode->blueBits);
18         glfwWindowHint(GLFW_REFRESH_RATE, mode->refreshRate);
19
20         window = glfwCreateWindow(mode->width, mode->height, "OpenGLEngine", ↩
               monitor, NULL);
21         SCR_HEIGHT = mode->height;
22         SCR_WIDTH = mode->width;
23     } else
24     {
25         window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT, "OpenGLEngine", NULL, NULL↩
               );
```

```
    }

    if (window == NULL)
    {
        std::cout << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return NULL;
    }

    glfwSetKeyCallback(window,key_callback);
    glfwMakeContextCurrent(window);
    glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
    glfwSetCursorPosCallback(window, mouse_callback);
    glfwSetScrollCallback(window, scroll_callback);
    if (!enableCursor)
        glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);

    if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
    {
        std::cout << "Failed to initialize GLAD" << std::endl;
        return NULL;
    }

    return window;
}
```

9.1.2 Input de l'usuari

Per la gestió del **input** de l'usuari, definim dues funcions. La primera s'encarrega de gestionar l'entrada de teclat, on, segons la tecla que es pressioni, modifiquem la càmera. Per altra banda, també interactua amb el **Renderer** per tal d'aplicar els diferents efectes de post-processat a l'escena, i modificar la velocitat de la càmera:

```
void processInput(GLFWwindow *window,World *world)
{
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);
    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        cam->ProcessKeyboard(FORWARD, deltaTime);
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        cam->ProcessKeyboard(BACKWARD, deltaTime);
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        cam->ProcessKeyboard(LEFT, deltaTime);
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        cam->ProcessKeyboard(RIGHT, deltaTime);
    if (glfwGetKey(window,GLFW_KEY_SPACE) == GLFW_PRESS)
        cam->ProcessKeyboard(UP,deltaTime);
    if (glfwGetKey(window,GLFW_KEY_C) == GLFW_PRESS)
        cam->ProcessKeyboard(DOWN,deltaTime);
    if (glfwGetKey(window,GLFW_KEY_Q) == GLFW_PRESS)
        cam->ProcessKeyboard(RLEFT,deltaTime);
    if (glfwGetKey(window,GLFW_KEY_E) == GLFW_PRESS)
        cam->ProcessKeyboard(RRIGHT,deltaTime);
```

```
21      if (glfwGetKey(window,GLFW_KEY_C) == GLFW_PRESS)
22          cam->ProcessKeyboard(DOWN,deltaTime);
23      if (glfwGetKey(window,GLFW_KEY_LEFT_SHIFT) == GLFW_PRESS)
24          cam->MovementSpeed += 50;
25      if (glfwGetKey(window,GLFW_KEY_LEFT_CONTROL) == GLFW_PRESS)
26          cam->MovementSpeed -= 50;
27      if (glfwGetKey(window,GLFW_KEY_0) == GLFW_PRESS)
28          world->getRenderer()->setPostProcess(5);
29      if (glfwGetKey(window,GLFW_KEY_1) == GLFW_PRESS)
30          world->getRenderer()->setPostProcess(0);
31      if (glfwGetKey(window,GLFW_KEY_2) == GLFW_PRESS)
32          world->getRenderer()->setPostProcess(1);
33      if (glfwGetKey(window,GLFW_KEY_3) == GLFW_PRESS)
34          world->getRenderer()->setPostProcess(2);
35      if (glfwGetKey(window,GLFW_KEY_4) == GLFW_PRESS)
36          world->getRenderer()->setPostProcess(3);
37      if (glfwGetKey(window,GLFW_KEY_5) == GLFW_PRESS)
38          world->getRenderer()->setPostProcess(4);
39
40  }
```

Per altra banda, la funció que gestiona el moviment del ratolí captura el moviment d'aquest i habilita el moviment en cas que l'usuari tingui pressionat el botó secundari:

```
1   void mouse_callback(GLFWwindow* window, double xpos, double ypos)
2   {
3       float xoffset = xpos - lastX;
4       float yoffset = lastY - ypos; // reversed since y-coordinates go from bottom to↩
              top
5       if (glfwGetMouseButton(window,GLFW_MOUSE_BUTTON_2))
6       {
7           glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
8           enableCursor = false;
9       }
10      else
11      {
12          glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_NORMAL);
13          enableCursor = true;
14
15      }
16      lastX = xpos;
17      lastY = ypos;
18      if (!enableCursor)
19          cam->ProcessMouseMovement(xoffset, yoffset,true);
20
21  }
```

9.1.3 Programa principal

El programa principal s'encarrega de crear la finestra, iniciar el context de **ImGUI**, crear el món i gestionar el bucle principal, on processem l'entrada de teclat, renderitzem el món, calculem el temps entre *frames* (**deltaTime**) i preparem **GLFW** pel següent *frame*. Acabat el bucle principal, eliminem el món i finalitzem

ImGUI i GLFW:

```
int main()
{

    GLFWwindow *window = createWindow();

    stbi_set_flip_vertically_on_load(true);
    glPatchParameteri(GL_PATCH_VERTICES,3);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);
    GLint MaxPatchVertices = 0;
    glGetIntegerv(GL_MAX_PATCH_VERTICES, &MaxPatchVertices);
    printf("Max supported patch vertices %d\n", MaxPatchVertices);
    glCullFace(GL_BACK);
    glFrontFace(GL_CCW);

    World *world = new World("../Scenes/Scene1.xml","../Textures/SkyBox/SpaceHres/"
        ,SCR_WIDTH,SCR_HEIGHT,cam);

    IMGUI_CHECKVERSION();
    ImGui::CreateContext();
    ImGuiIO& io = ImGui::GetIO(); (void)io;
    ImGui::StyleColorsDark();
    ImGui_ImplGlfw_InitForOpenGL(window, true);
    ImGui_ImplOpenGL3_Init("#version 150");

    while (!glfwWindowShouldClose(window))
    {
        processInput(window, world);
        if (wireframe)
            glPolygonMode( GL_FRONT_AND_BACK, GL_LINE );
        else
            glPolygonMode( GL_FRONT_AND_BACK, GL_FILL );
        world->renderWorld(wireframe);

        float currentFrame = glfwGetTime();
        deltaTime = currentFrame - lastFrame;
        cam->deltaTime = deltaTime;
        lastFrame = currentFrame;
        glfwSwapBuffers(window);
        glfwPollEvents();
    }
    ImGui_ImplOpenGL3_Shutdown();
    ImGui_ImplGlfw_Shutdown();
    ImGui::DestroyContext();
    delete world;
    glfwTerminate();
    return 0;
}
```

9.2 Creació del món

9.2.1 Parser

L'objecte **XMLParser** s'encarrega de llegir i crear tots els elements que conformen l'escena. Per fer això, primer definim el format que ha de tenir el fitxer **XML** on inicialment llegim la càmera de l'escena amb la posició i orientació determinades, seguit d'un subgrup d'entitats agrupades dins del mateix *tag* **Entities**, que defineixen la llum global i els planetes amb els atributs corresponents:

```
<Scene>
   <Camera type="cam">
      <Orientation pitch="38.995235" yaw="-35.091671" roll="-8.017399"/>
      <Position x="4493.371094" y="15442.373047" z="13443.627930"/>
   </Camera>
   <Entities>
      <Entity type="Light">
         <Light type="dirLight">
            <Orientation pitch="0.968000" yaw="94.838997"/>
            <Ambient x="0.005" y="0.005f" z="0.005f"/>
            <Diffuse x="0.9" y="0.9f" z="0.9"/>
            <Specular x="1.0" y="1.0" z="1.0"/>
         </Light>
      </Entity>
      <Entity type="Planet">
         <Planet id="0" type="planet" hasAtmos="0" hasWater="0" radius="7000" nSeg="8">
            <Position x="-25000" y="0" z="40000"/>
            <NoiseSettings maxHeight="570" noiseFreq="0.00055" octaves="8" lacunarity="2.3" fGain="0.5" fWeStr="0.6" fPinPonStr="3" cellJitter="1" domWarpAmp="50" minValue="0" noiseTypeSel="4" fractalTypeSel="2" cellDistTypeSel="3" cellReturnTypeSel="2" domWarpTypeSel="2"/>
            <AtmosSettings radius="30000" k_r="0.166" k_m="0.0025" e="14.3" H="4.0" L="1.0" g_m="-0.85" numOutScatter="10.0" numInScatter="10.0" scale="1.0">
               <Color x="0.1" y="0.2" z="0.8"/>
            </AtmosSettings>
            <Textures>
               <Diffuse>
                  <d>Planet/worn-bumpy-rock-albedo-1024.png</d>
                  <d>Planet/rock-snow-ice-albedo-1024.png</d>
                  <d>Planet/snow-packed-albedo-1024.png</d>
               </Diffuse>
               <Normal>
                  <n>Planet/worn-bumpy-rock-normal-1024.jpg</n>
                  <n>Planet/rock-snow-ice-normal-1024.jpg</n>
                  <n>Planet/snow-packed-normal-1024.jpg</n>
               </Normal>
            </Textures>
            <Biomes>
               <Biome latStart="1" latEnd="0.9">
                  <Textures>
                     <Texture index="1" hStart="0.0" hEnd="0.5"/>
                     <Texture index="2" hStart="0.5" hEnd="1.5"/>
                  </Textures>
               </Biome>
               <Biome latStart="0.9" latEnd="0.1">
```

```
                    <Textures>
                        <Texture index="0" hStart="0.0" hEnd="0.5"/>
                        <Texture index="1" hStart="0.5" hEnd="1.5"/>
                    </Textures>
                </Biome>
                <Biome latStart="0.1" latEnd="0.0">
                    <Textures>
                        <Texture index="1" hStart="0.0" hEnd="0.5"/>
                        <Texture index="2" hStart="0.5" hEnd="1.5"/>
                    </Textures>
                </Biome>
            </Biomes>
        </Planet>
    </Entity>
  </Entities>
</Scene>
```

Per llegir aquest fitxer, la classe **XMLParser** rep la ruta, obre el fitxer i el guarda com un vector de caràcters i el carrega a la llibreria **RapidXML** que permet iterar dins de l'arbre i crear/actualitzar els diferents objectes que componen l'escena:

```
XmlParser::XmlParser(std::string path, Camera *cam) {
    this->path = path;
    this->cam = cam;
    xml_document<> doc;
    // Read the xml file into a vector
    ifstream theFile (this->path);
    vector<char> buffer((istreambuf_iterator<char>(theFile)), istreambuf_iterator<char>());
    buffer.push_back('\0');
    // Parse the buffer using the xml file parsing library into doc
    doc.parse<0>(&buffer[0]);
    // Find our root node
    _rootNode = doc.first_node("Scene");
    entIndex = 0;
    nPointLights = 0;
    nPlanets = 0;
    xml_node<> * cameraNode = _rootNode->first_node("Camera");
    xml_node<> * camPosNode = cameraNode->first_node("Position");
    xml_node<> * camOrient = cameraNode->first_node("Orientation");

    cam->Position = getValues3(camPosNode);
    cam->Pitch = stof(camOrient->first_attribute("pitch")->value());
    cam->Yaw = stof(camOrient->first_attribute("yaw")->value());
    cam->Roll = stof(camOrient->first_attribute("roll")->value());
    cam->updateCameraVectors();
    xml_node<> * entities = _rootNode->first_node("Entities");
    for (xml_node<> * ent = entities->first_node("Entity");ent;ent = ent->next_sibling("Entity"))
    {
        string entType = ent->first_attribute("type")->value();
        if (entType == "PhysicsObject")
        {
            xml_node<> *model = ent->first_node("Model");
            _ents.push_back(getObject(model));

        } else if (entType == "Light")
        {
            xml_node<> *light = ent->first_node("Light");
```

```
37              _ents.push_back(getLight(light));
38          } else if (entType == "Planet")
39          {
40              xml_node<> *planet = ent->first_node("Planet");
41              _ents.push_back(getPlanet(planet));
42              nPlanets++;
43          }
44          entIndex++;
45      }
46      theFile.close();
47  }
```

Per crear un planeta, recuperem tots els atributs que formen la geometria, llegim el llistat de textures, els atributs de l'atmosfera i finalment tots els biomes que el formen:

```
1   Planet *XmlParser::getPlanet(xml_node<> *planet) {
2       int id = stoi(planet->first_attribute("id")->value());
3       bool hasAtmos = stoi(planet->first_attribute("hasAtmos")->value());
4       bool hasWater = stoi(planet->first_attribute("hasWater")->value());
5       float radius = stof(planet->first_attribute("radius")->value());
6       int nSeg = stoi(planet->first_attribute("nSeg")->value());
7       glm::vec3 position = getValues3(planet->first_node("Position"));
8       xml_node<> * noiseSettings = planet->first_node("NoiseSettings");
9       //--------NOISE SETTINGS--------
10      float maxHeight = stof(noiseSettings->first_attribute("maxHeight")->value());
11      float noiseFreq = stof(noiseSettings->first_attribute("noiseFreq")->value());
12      int octaves = stoi(noiseSettings->first_attribute("octaves")->value());
13      float lacunarity = stof(noiseSettings->first_attribute("lacunarity")->value());
14      float fGain = stof(noiseSettings->first_attribute("fGain")->value());
15      float fWeStr = stof(noiseSettings->first_attribute("fWeStr")->value());
16      float fPinPonStr = stof(noiseSettings->first_attribute("fPinPonStr")->value());
17      float cellJitter = stof(noiseSettings->first_attribute("cellJitter")->value());
18      float domWarpAmp= stof(noiseSettings->first_attribute("domWarpAmp")->value());
19      float minValue = stof(noiseSettings->first_attribute("minValue")->value());
20      int noiseTypeSel = stoi(noiseSettings->first_attribute("noiseTypeSel")->value()↩
            );
21      int fractalTypeSel = stoi(noiseSettings->first_attribute("fractalTypeSel")->↩
            value());
22      int cellDistTypeSel = stoi(noiseSettings->first_attribute("cellDistTypeSel")->↩
            value());
23      int cellReturnTypeSel = stoi(noiseSettings->first_attribute("cellReturnTypeSel"↩
            )->value());
24      int domWarpTypeSel = stoi(noiseSettings->first_attribute("domWarpTypeSel")->↩
            value());
25      //--------TEXTURES--------
26      std::vector<string> pathDiffuse;
27      std::vector<string> pathNormal;
28      xml_node<> * textures = planet->first_node("Textures");
29      xml_node<> * diffPath = textures->first_node("Diffuse");
30      xml_node<> * normPath = textures->first_node("Normal");
31      for (xml_node<> * d = diffPath->first_node("d");d;d = d->next_sibling())
32      {
33          pathDiffuse.push_back(d->value());
34      }
35      for (xml_node<> * n = normPath->first_node("n");n;n = n->next_sibling())
36      {
37          pathNormal.push_back(n->value());
38      }
```

```
39      Planet * newPlanet = new Planet(radius,nSeg,hasAtmos,maxHeight,noiseFreq,↩
            octaves,
40                                      lacunarity,fGain,fWeStr,fPinPonStr,cellJitter,
41                                      domWarpAmp,minValue,noiseTypeSel,fractalTypeSel↩
                                            ,
42                                      cellDistTypeSel,cellReturnTypeSel,↩
                                            domWarpTypeSel,
43                                      pathDiffuse,pathNormal,position,hasWater);
44      newPlanet->id = id;
45      //--------ATMOS SETTINGS--------
46      xml_node<> * atmosSettings = planet->first_node("AtmosSettings");
47      float atmosRadius = stof(atmosSettings->first_attribute("radius")->value());
48      float kr = stof(atmosSettings->first_attribute("k_r")->value());
49      float km = stof(atmosSettings->first_attribute("k_m")->value());
50      float e = stof(atmosSettings->first_attribute("e")->value());
51      float h = stof(atmosSettings->first_attribute("H")->value());
52      float l = stof(atmosSettings->first_attribute("L")->value());
53      float gm = stof(atmosSettings->first_attribute("g_m")->value());
54      float numOutScatter = stof(atmosSettings->first_attribute("numOutScatter")->↩
            value());
55      float numInScatter = stof(atmosSettings->first_attribute("numInScatter")->value↩
            ());
56      float scale = stof(atmosSettings->first_attribute("scale")->value());
57      glm::vec3 color = getValues3(atmosSettings->first_node("Color"));
58      newPlanet->addCamera(cam);
59      if (hasAtmos)
60          newPlanet->setupAtmosphere(atmosRadius,kr,km,e,h,l,gm,numOutScatter,↩
                numInScatter,
61                                     scale,color);
62      xml_node<> * biomeRoot = planet->first_node("Biomes");
63      for (xml_node<> * bio = biomeRoot->first_node("Biome");bio;bio = bio->↩
            next_sibling())
64      {
65          Biome bioAux;
66          bioAux.latStart = stof(bio->first_attribute("latStart")->value());
67          bioAux.latEnd = stof(bio->first_attribute("latEnd")->value());
68          xml_node<> * textIndex = bio->first_node("Textures");
69
70          for (xml_node<>  * textNode = textIndex->first_node("Texture");textNode;↩
                textNode = textNode->next_sibling("Texture"))
71          {
72              TextHeight textHeight;
73              textHeight.index = stoi(textNode->first_attribute("index")->value());
74              textHeight.hStart = stof(textNode->first_attribute("hStart")->value());
75              textHeight.hEnd = stof(textNode->first_attribute("hEnd")->value());
76              bioAux.textHeight.push_back(textHeight);
77          }
78          newPlanet->addBiome(bioAux);
79
80      }
81      return newPlanet;
82  }
```

Similar-ment, per a la llum global, obtenim els atributs que defineixen el color i propietats de la llum, seguit de la seva orientació dins de l'escena:

```
1   Light* XmlParser::getLight(xml_node<> *light) {
2       string typePre = light->first_attribute("type")->value();
3       xml_node<> * amb = light->first_node("Ambient");
4       xml_node<> * diff = light->first_node("Diffuse");
```

```
    xml_node<> * spec = light->first_node("Specular");
    glm::vec3 ambient = getValues3(amb);
    glm::vec3 diffuse = getValues3(diff);
    glm::vec3 specular = getValues3(spec);
    if (typePre == "dirLight")
    {
        xml_node<> * dir = light->first_node("Orientation");
        float sunPitch = stof(dir->first_attribute("pitch")->value());
        float sunYaw = stof(dir->first_attribute("yaw")->value());

        glm::vec2 orientation(sunPitch,sunYaw);
        auto lightAux = new Light(typePre,ambient,diffuse,specular,orientation,
            entIndex);
        lightAux->setDirection(orientation);
        this->sun = lightAux;
        return lightAux;
    }
}
```

Finalment, per guardar la informació de l'escena en temps d'execució, tornem a carregar el document, i guardem el valor dels atributs de tots els objectes que tenim a memòria:

```
void XmlParser::saveWorld() {

    xml_document<> doc;
    // Read the xml file into a vector
    ifstream theFile (this->path);
    vector<char> buffer((istreambuf_iterator<char>(theFile)), istreambuf_iterator<
        char>());
    buffer.push_back('\0');
    // Parse the buffer using the xml file parsing library into doc
    doc.parse<parse_full | parse_no_data_nodes>(&buffer[0]);
    // Find our root node
    _rootNode = doc.first_node("Scene");
    xml_node<> * camNode = _rootNode->first_node("Camera");
    xml_node<> * camNodePos = camNode->first_node("Position");

    std::string x = std::to_string(cam->Position.x);
    std::string y = std::to_string(cam->Position.y);
    std::string z = std::to_string(cam->Position.z);
    camNodePos->first_attribute("x")->value(doc.allocate_string(x.c_str()));
    camNodePos->first_attribute("y")->value(doc.allocate_string(y.c_str()));
    camNodePos->first_attribute("z")->value(doc.allocate_string(z.c_str()));
    xml_node<> * camOrientation = camNode->first_node("Orientation");
    cam->updateCameraVectors();
    std::string x1 = std::to_string(cam->oldPitch);
    std::string y1 = std::to_string(cam->oldYaw);
    std::string z1 = std::to_string(cam->oldRoll);
    camOrientation->first_attribute("pitch")->value(doc.allocate_string(x1.c_str())
        );
    camOrientation->first_attribute("yaw")->value(doc.allocate_string(y1.c_str()));
    camOrientation->first_attribute("roll")->value(doc.allocate_string(z1.c_str()))
        ;
    xml_node<> * ents = _rootNode->first_node("Entities");
    for (xml_node<> * ent = ents->first_node("Entity");ent;ent = ent->next_sibling(
        "Entity"))
    {
        string entType = ent->first_attribute("type")->value();
```

```
33          if (entType == "Light")
34          {
35              xml_node<> *light = ent->first_node("Light");
36              string typePre = light->first_attribute("type")->value();
37              if (typePre == "dirLight")
38              {
39                  xml_node<> * dir = light->first_node("Orientation");
40
41                  string sunSettings = sun->toString();
42                  vector<string> res = split(sunSettings,":");
43                  vector<string> col = split(res[1],",");
44                  dir->first_attribute("pitch")->value(doc.allocate_string(col[0].↩
                        c_str()));
45                  dir->first_attribute("yaw")->value(doc.allocate_string(col[1].c_str↩
                        ()));
46              }
47
48          } else if (entType == "Planet")
49          {
50              xml_node<> *planet = ent->first_node("Planet");
51              int id = stoi(planet->first_attribute("id")->value());
52
53              for (auto _ent : _ents)
54              {
55                  if (_ent->id == id && _ent->getType() == 3)
56                  {
57
58                      xml_node<>* noiseSettings  = planet->first_node("NoiseSettings"↩
                            );
59                      Planet * auxPlanet = dynamic_cast<Planet*>(_ent);
60                      string pSettings = auxPlanet->toString();
61                      vector<string> res = split(pSettings,"\n");
62                      for (const auto& attr : res)
63                      {
64                          vector<string> namVal = split(attr,":");
65                          const char* name = namVal[0].c_str();
66                          const char* val = doc.allocate_string(namVal[1].c_str());
67                          noiseSettings->first_attribute(name)->value(val);
68
69                      }
70                      if (auxPlanet->hasAtmosphere())
71                      {
72                          string atmosSettings = auxPlanet->getAtmosSettings();
73                          vector<string> atmosRes = split(atmosSettings,"\n");
74                          for (const auto& attr : atmosRes)
75                          {
76                              xml_node<> * atmosSettingsNode = planet->first_node("↩
                                    AtmosSettings");
77                              vector<string> namVal = split(attr,":");
78                              const char * name = namVal[0].c_str();
79                              const char * val = namVal[1].c_str();
80                              if (namVal[0] != "color")
81                              {
82                                  atmosSettingsNode->first_attribute(name)->value(doc↩
                                        .allocate_string(val));
83                              }
84                              else
85                              {
86                                  vector<string> colorVal = split(val,",");
87                                  xml_node<> * colorNode = atmosSettingsNode->↩
                                        first_node("Color");
88                                  colorNode->first_attribute("x")->value(doc.↩
```

```
                                    allocate_string(colorVal[0].c_str()));
89                                colorNode->first_attribute("y")->value(doc.↩
                                    allocate_string(colorVal[1].c_str()));
90                                colorNode->first_attribute("z")->value(doc.↩
                                    allocate_string(colorVal[2].c_str()));
91
92                              }
93                          }
94                      }
95                  }
96              }
97          }
98      }
99      ofstream file;
100     file.open(path.c_str());
101     std::string data;
102     rapidxml::print(std::back_inserter(data),doc);
103     file << data;
104     file.close();
105 }
```

9.2.2 Món

La creació del món mitjançant la classe **World** ens permet enllaçar el programa principal amb la resta de l'estructura del motor. Inicialment, quan construïm aquesta classe, obtenim el llistat d'entitats del **Parser**, construïm el **Renderer** i actualitzem els diferents atributs:

```
World::World(const char *scenePath, const char *skyBoxPath, unsigned int scrWidth, unsigned int scrHeight,
             Camera *camera) {
    renderer = new Renderer(scrWidth,scrHeight,camera,skyBoxPath);
    parser =  new XmlParser(scenePath,camera);
    parser->readData();
    worldEntities = parser->_ents;
    renderer->camera = camera;
    this->sun = parser->sun;
    renderer->addSun(sun);
    renderer->preRender(worldEntities);
    sunPitch = sun->getDirection().y;
    sunYaw = sun->getDirection().x;
}
```

Seguidament, des del programa principal, és crida de forma iterativa el següent mètode, que s'encarrega de renderitzar el món i crear la interfície que permet modificar la direcció de la llum i guardar els canvis realitzats a l'escena:

```
void World::renderWorld(bool wireframe)
{
    renderer->render(worldEntities,worldDeco,wireframe);
    ImGui::Begin("World Settings",NULL,ImGuiWindowFlags_MenuBar);
    ImGui::SetWindowFontScale(setting_fontSize);
    sun->setDirection(glm::vec2(sunPitch,sunYaw));
    ImGui::SliderFloat("Sun yaw",&sunYaw,-180,180);
    ImGui::SliderFloat("Sun pitch",&sunPitch,-90,90);
    bool saveWorld = ImGui::Button("Save");
    if (saveWorld) parser->saveWorld();
    ImGui::End();
    ImGui::Render();
    ImGui_ImplOpenGL3_RenderDrawData(ImGui::GetDrawData());
}
```

Finalment, quan destruïm aquesta classe, s'eliminen tots els objectes allocats a memòria:

```
World::~World()
{
    for (auto ent : worldEntities)
        delete ent;
    delete renderer;
    delete parser;
}
```

9.3 Renderer

El **Renderer** ens permet dibuixar l'escena completa per pantalla, aplicar efectes de post-processat sobre aquesta i crear la textura per mostrar reflexions a l'aigua.

9.3.1 Implementació del Renderer

Construïm aquesta classe assignant els diferents atributs que ens arriben per paràmetre i carregant els **shaders**:

```
Renderer::Renderer(unsigned int scrWidth, unsigned int scrHeight, Camera *camera, const char* skyboxPath) {
    this->camera = camera;
    Shader lightShader("../Shaders/lightingShaderVertex.shader",
                       "../Shaders/lightingShaderFragment.shader",
                       "../Shaders/lightingShaderTCS.shader",
                       "../Shaders/lightingShaderTES.shader");

    Shader screenShader = Shader("../Shaders/PostProcess/screenShader.vs","../Shaders/PostProcess/screenShader.fs");
    Shader skyboxShader = Shader("../Shaders/skyboxShader.vs","../Shaders/skyboxShader.fs");

    shaders.push_back(lightShader);//0
    shaders.push_back(screenShader);//1
    shaders.push_back(skyboxShader);//2

    setupFrameBuffer();
    setupSkyBox(skyboxPath);
}
```

Seguidament, generem els diferents **FrameBuffers** que ens permetran aplicar els diferents efectes esmentats:

```
void Renderer::setupFrameBuffer() {
    float quadVertices[] = {
            -1.0f,  1.0f,  0.0f, 1.0f,
            -1.0f, -1.0f,  0.0f, 0.0f,
            1.0f, -1.0f,  1.0f, 0.0f,

            -1.0f,  1.0f,  0.0f, 1.0f,
            1.0f, -1.0f,  1.0f, 0.0f,
            1.0f,  1.0f,  1.0f, 1.0f
    };

    glGenVertexArrays(1, &quadVAO);
    glGenBuffers(1, &quadVBO);
    glBindVertexArray(quadVAO);
    glBindBuffer(GL_ARRAY_BUFFER, quadVBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(quadVertices), &quadVertices, GL_STATIC_DRAW);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 4 * sizeof(float), (void*)0);
    glEnableVertexAttribArray(1);
```

```
20      glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 4 * sizeof(float), (void*)(2 * ↩
            sizeof(float)));
21
22      shaders[3].use();
23      shaders[3].setInt("screenTexture",0);
24
25      glGenFramebuffers(1, &frameBuffer);
26      glBindFramebuffer(GL_FRAMEBUFFER, frameBuffer);
27
28      glGenTextures(1, &textColorBuffer);
29      glBindTexture(GL_TEXTURE_2D, textColorBuffer);
30      glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, setting_scrWidth, setting_scrHeight, 0, ↩
            GL_RGB, GL_UNSIGNED_BYTE, NULL);
31      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
32      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
33      glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, ↩
            textColorBuffer, 0);
34
35      glGenRenderbuffers(1, &rbo);
36      glBindRenderbuffer(GL_RENDERBUFFER, rbo);
37      glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH24_STENCIL8, setting_scrWidth, ↩
            setting_scrHeight); // use a single renderbuffer object for both a depth ↩
            AND stencil buffer.
38      glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT, ↩
            GL_RENDERBUFFER, rbo); // now actually attach it
39      // now that we actually created the frameBuffer and added all attachments we ↩
            want to check if it is actually complete now
40      if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
41          cout << "ERROR::FRAMEBUFFER:: Framebuffer is not complete!" << endl;
42      glBindFramebuffer(GL_FRAMEBUFFER, 0);
43
44      glGenFramebuffers(1, &reflexionFBO);
45      glBindFramebuffer(GL_FRAMEBUFFER, reflexionFBO);
46
47      glGenTextures(1, &reflexion);
48      glBindTexture(GL_TEXTURE_2D, reflexion);
49      glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 800, 600, 0, GL_RGBA, GL_UNSIGNED_BYTE, ↩
            NULL);
50      glGenerateMipmap(GL_TEXTURE_2D);
51      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
52      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
53      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
54      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
55      glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, ↩
            reflexion, 0);
56
57  }
```

Tot seguit, construïm l'estructura on apliquem el **sky box** (*cel* infinit), definint els vèrtexs que la formen amb el corresponent **VAO**, i llegim la textura:

```
1   void Renderer::setupSkyBox(const char * path) {
2       float skyboxVertices[] = {
3               -1.0f,  1.0f, -1.0f,
4               -1.0f, -1.0f, -1.0f,
5               1.0f, -1.0f, -1.0f,
6               1.0f, -1.0f, -1.0f,
7               1.0f,  1.0f, -1.0f,
8               -1.0f,  1.0f, -1.0f...
9       };
```

```
10
11      glGenVertexArrays(1, &skyboxVAO);
12      glGenBuffers(1, &skyboxVBO);
13      glBindVertexArray(skyboxVAO);
14      glBindBuffer(GL_ARRAY_BUFFER, skyboxVBO);
15      glBufferData(GL_ARRAY_BUFFER, sizeof(skyboxVertices), &skyboxVertices, ↩
            GL_STATIC_DRAW);
16      glEnableVertexAttribArray(0);
17      glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
18      std::string format = ".png";
19      vector<std::string> faces
20      {
21          std::string(path)+"right"+format,
22          std::string(path)+"left"+format,
23          std::string(path)+"bottom"+format,
24          std::string(path)+"top"+format,
25          std::string(path)+"front"+format,
26          std::string(path)+"back"+format
27      };
28      cubemapTexture = loadCubemap(faces);
29      shaders[4].use();
30      shaders[4].setInt("skybox",0);
31  }
```

On el mètode **loadCubemap** ens permet carregar una textura cúbica, formada per 6 textures individuals que formen un cub. Veure figura 9.1:

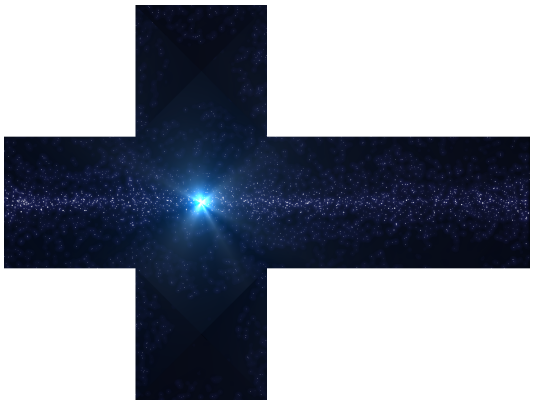


Figura 9.1: Exemple de textura cúbica

El codi que ens permet llegir aquesta textura rep com a paràmetre un vector amb el nom de cada cara, i especificant a **OpenGL** que es tracta d'un **cubemap**, llegim cada textura individualment, la carreguem a memòria i especifiquem els paràmetres:

```
unsigned int Renderer::loadCubemap(vector<std::string> faces)
{
    unsigned int textureID;
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);
    int width, height, nrChannels;
    for (unsigned int i = 0; i < faces.size(); i++)
    {
        unsigned char *data = stbi_load(faces[i].c_str(), &width, &height, &↩
            nrChannels, 0);
        if (data)
        {
            glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGB, width, ↩
                height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
            stbi_image_free(data);
        }
        else
        {
            std::cout << "Cubemap texture failed to load at path: " << faces[i] << ↩
                std::endl;
            stbi_image_free(data);
        }
    }
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);

    return textureID;
}
```

Seguidament, l'objecte món, crida iterativament el mètode **render** que s'encarrega de dibuixar l'escena 2 vegades per guardar el resultat als diferents **FrameBuffers**, crear finestres auxiliars per debugar l'escena amb **ImGUI** i crear les matrius de **View** i **Projection**:

```
1  void Renderer::render(vector<Entity *> worldEnts, bool wireframe) {
2      glEnable(GL_DEPTH_TEST);
3      ImGui_ImplOpenGL3_NewFrame();
4      ImGui_ImplGlfw_NewFrame();
5      ImGui::NewFrame();
6      glm::mat4 view;
7      camera->invertCameraPitch();
8      glm::vec3 oldCamPos = this->camera->Position;
9      this->camera->Position = this->camera->Position + (this->camera->Up * -100.f);
10     view = this->camera->GetViewMatrix();
11     glm::mat4 projection;
12     projection = glm::perspective(glm::radians(this->camera->Zoom), (float)800/600,↩
           setting_near, setting_far);
13     glBindFramebuffer(GL_FRAMEBUFFER,reflexionFBO);
14     renderScene(worldEnts,wireframe,view,projection,800,600,false);
15     ImGui::Begin("Game Window");
16     ImGui::SetWindowFontScale(setting_fontSize);
17     ImVec2 pos = ImGui::GetCursorScreenPos();
18     ImDrawList* drawList = ImGui::GetWindowDrawList();
19     drawList->AddImage((void*)reflexion,
20                        pos,
21                        ImVec2(pos.x + 800, pos.y + 600),
22                        ImVec2(0, 1),
23                        ImVec2(1, 0));
24     ImGui::End();
25     glBindFramebuffer(GL_FRAMEBUFFER,0);
26     camera->invertCameraPitch();
27     this->camera->Position = oldCamPos;
28     view = this->camera->GetViewMatrix();
29
30     if (!wireframe)
31         glBindFramebuffer(GL_FRAMEBUFFER, frameBuffer);
32     projection = glm::perspective(glm::radians(this->camera->Zoom), (float)↩
           setting_scrWidth/setting_scrHeight, setting_near, setting_far);
33     renderScene(worldEnts,wireframe,view,projection,setting_scrWidth,↩
           setting_scrHeight,true);
34     if (!wireframe)
35     {
36         glBindFramebuffer(GL_FRAMEBUFFER, 0);
37         glDisable(GL_DEPTH_TEST);
38         glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
39         glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
40         shaders[3].use();
41         glBindVertexArray(quadVAO);
42         glBindTexture(GL_TEXTURE_2D, textColorBuffer);
43         glDrawArrays(GL_TRIANGLES, 0, 6);
44     }
45 }
```

El mètode **renderScene** s'encarrega de netejar la finestra de l'últim *frame*, els diferents *buffers* de **OpenGL** i de dibuixar el *cel* i les entitats.

```
void Renderer::renderScene(vector<Entity*> worldEnts,
                           bool wireframe,glm::mat4 &view, glm::mat4 &projection, ↩
                               float width, float height, bool drawEffects) {
    glViewport(0, 0, width, height);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
    renderSkybox(view,projection);
    drawEntities(worldEnts,view,projection,shaders[0],!drawEffects);
}
```

Les matrius **View** i **Projection** s'han de passar com **uniforms** al **shader**. En el cas que es tracti del cel, eliminem la translació de la matriu **View** obtenint un espai infinit. Això ho fem amb el següent mètode:

```
void Renderer::renderLoopCamera(Shader shader,glm::mat4 view, glm::mat4 projection,↩
    bool skybox) {
    if (skybox)
        view = glm::mat4(glm::mat3(camera->GetViewMatrix()));
    shader.setMat4("view",view);
    shader.setMat4("projection",projection);
    glm::mat4 cameraModel(1.0f);
    shader.setMat4("model", cameraModel);
}
```

Seguin el mètode **renderScene**, cridem a **renderSkybox** que dibuixa el cel modificant el paràmetre de profunditat d'**OpenGL**, indicant que qualsevol objecte es superposi a aquest:

```
void Renderer::renderSkybox(glm::mat4 view, glm::mat4 projection) {
    glDepthFunc(GL_LEQUAL);
    shaders[2].use();
    renderLoopCamera(shaders[2],view,projection,true);
    glBindVertexArray(skyboxVAO);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_CUBE_MAP, cubemapTexture);
    glDrawArrays(GL_TRIANGLES, 0, 36);
    glBindVertexArray(0);
    glDepthFunc(GL_LESS); // set depth function back to default*/
}
```

El **shader** que s'encarrega de generar la geometria del cel i aplicar la textura cúbica està format pels següents programes:

Vertex Shader

Que transforma la posició en un espai normalitzat entre [0..1], crea les coordenades de textura a partir de les coordenades del vèrtex i modifica la posició z tal que sempre sigui 1, per tal que **OpenGL** consideri la seva profunditat com a màxima.

```
#version 330 core
layout (location = 0) in vec3 aPos;

out vec3 TexCoords;

uniform mat4 projection;
uniform mat4 view;

void main()
{
    TexCoords = vec3(aPos.x,-aPos.y,aPos.z);
    vec4 pos = projection * view * vec4(aPos, 1.0);
    gl_Position = pos.xyww;
}
```

Fragment Shader

Que aplica el color que es correspon a la textura en les coordenades de textura especifiques.

```
#version 330 core
out vec4 FragColor;

in vec3 TexCoords;

uniform samplerCube skybox;

void main()
{
    FragColor = texture(skybox, TexCoords);
}
```

Continuant amb el mètode **renderScene**, cridem al mètode **drawEntities** que s'encarrega de recórrer el vector d'entitats i renderitzar-les. En cas de trobar un planeta, renderitza la seva interfície i actualitza la textura de reflexió de l'aigua:

```
void Renderer::drawEntities(std::vector<Entity*> worldEnts, glm::mat4 view, glm::↩
    mat4 projection,Shader &shader, bool drawEffects) {
    shader.use();
    renderLoopCamera(shader,view,projection);
    shader.setFloat("material.shininess", 64.0f);
    shader.setVec3("viewPos",camera->Position);
    for (int i = 0; i < worldEnts.size();i++)
    {
        if (worldEnts[i]->getType() != 4)
        {
            worldEnts[i]->draw(shader,false,depthMap);
            if (worldEnts[i]->getType() == 3)
            {
                Planet * planet = dynamic_cast<Planet *>(worldEnts[i]);
                planet->renderGui();
                planet->setWaterTexture(reflexion);
                planet->setNoDrawEffects(drawEffects);
            }
        }
    }
}
```

Finalment, quan destruïm aquest objecte, eliminem les textures, **Buffer Objects** i **shaders** creats durant l'execució:

```
enderer::~Renderer() {
    glDeleteVertexArrays(1,&quadVAO);
    glDeleteVertexArrays(1,&skyboxVAO);
    glDeleteBuffers(1,&quadVBO);
    glDeleteBuffers(1,&skyboxVBO);
    glDeleteTextures(1,&cubemapTexture);
    for (auto shader: shaders)
        glDeleteShader(shader.ID);
}
```

9.3.2 Efectes de post-processat

Apliquem els efectes de post-processat sobre la pantalla definint 2 **shaders** diferents, format per un **Vertex Shader** comú i un **Fragment Shader** independent per efecte.

Vertex Shader
Que simplement passa la posició del vèrtex i les coordenades de textura al **Fragment Shader**:

```
#version 330 core
layout (location = 0) in vec2 aPos;
layout (location = 1) in vec2 aTexCoords;

out vec2 TexCoords;

void main()
{
    TexCoords = aTexCoords;
    gl_Position = vec4(aPos.x, aPos.y, 0.0, 1.0);
}
```

Grey-Scale Fragment Shader
On apliquem el mètode de la lluminositat, que consisteix a generar una mitjana dels colors donant més pes al verd i al vermell per compensar la percepció humana. Veure figura 9.2.

```
#version 330 core

out vec4 FragColor;
in vec2 TexCoords;
uniform sampler2D screenTexture;

void main()
{
    FragColor = texture(screenTexture, TexCoords);
    float average = 0.2126 * FragColor.r + 0.7152 * FragColor.g + 0.0722 * ↩
        FragColor.b;
    FragColor = vec4(average, average, average, 1.0);
}
```

Invert Color Fragment Shader
Que llegeix el color de la textura i l'inverteix. Veure figura 9.3.

```
#version 330 core

out vec4 FragColor;
in vec2 TexCoords;
uniform sampler2D screenTexture;

void main()
```

```
8  {
9      vec3 col = texture(screenTexture, TexCoords).rgb;
10     FragColor = vec4(1-col, 1.0);
11 }
```

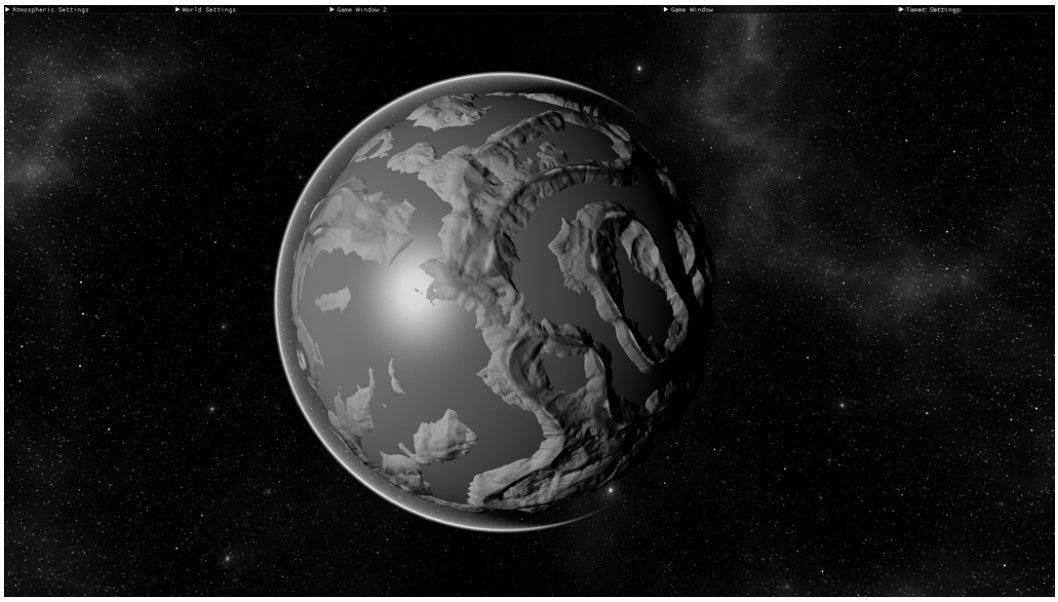


Figura 9.2: Efecte de post processat blanc i negre

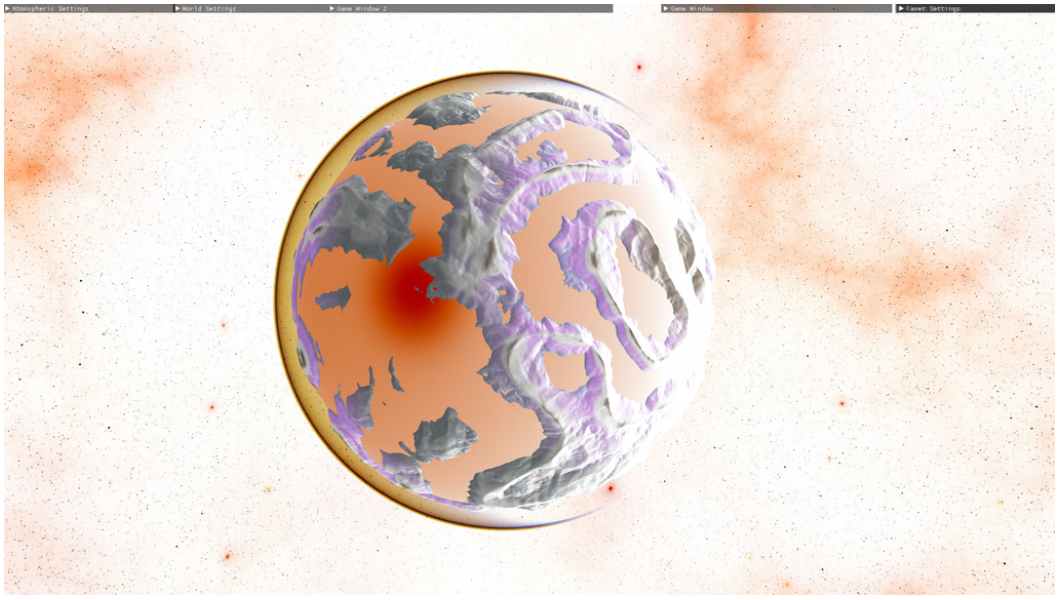


Figura 9.3: Efecte de post processat amb colors invertits

9.4 Càmera

La càmera de l'escena la construïm amb una posició, un vector que apunta amunt i els angles de rotació lateral i vertical determinats:

```
Camera::Camera(glm::vec3 position, glm::vec3 up, float yaw, float pitch)
{
    Position = position;
    WorldUp = up;
    Yaw = yaw;
    Pitch = pitch;
    Roll = 90;
    updateCameraVectors();
}
```

Seguidament, actualitzem els vectors de la càmera creant un quaternió amb els angles de rotació i multiplicant aquest pel quaternió que defineix l'orientació. Fet això, calculem els nous vectors direccionals conjugant el quaternió, obtenint l'invers de la rotació i multiplicant per un vector que correspon a la direcció:

```
void Camera::updateCameraVectors() {
    glm::quat q = glm::quat(glm::vec3(-Pitch, Yaw, -Roll));
    Pitch = Yaw = Roll = 0;

    orientation = q * orientation;
    orientation = glm::normalize(orientation);

    Front = glm::conjugate(orientation) * glm::vec3(0.0f, 0.0f, -1.0f);
    Right = glm::conjugate(orientation) * glm::vec3(1.0f, 0.0f, 0.0f);
    Up = glm::conjugate(orientation) * glm::vec3(0.0f, 1.0f, 0.0f);
}
```

Per moure la càmera, donada una direcció, actualitzem la posició incrementant a aquesta el vector direccional que li pertoca, multiplicat per la velocitat. La velocitat és definida per un valor fix **MovementSpeed** multiplicat pel temps entre *frames* **deltaTime**, que evita que la velocitat total varii en funció de la velocitat de l'aplicació:

```
void Camera::ProcessKeyboard(Camera_Movement direction, float deltaTime)
{
    float velocity = MovementSpeed * deltaTime;
    if (direction == FORWARD)
        Position += Front * velocity;
    if (direction == BACKWARD)
        Position -= Front * velocity;
    if (direction == LEFT)
        Position -= Right * velocity;
    if (direction == RIGHT)
        Position += Right * velocity;
    if (direction == UP)
        Position += Up * velocity;
    if (direction == DOWN)
```

```
15          Position -= Up * velocity;
16      if (direction == RLEFT)
17          Roll += (SENSITIVITY)*1000*deltaTime;
18      if (direction == RRIGHT)
19          Roll -= (SENSITIVITY)*1000*deltaTime;
20      updateCameraVectors();
21  }
```

Per capturar el moviment del ratolí, capturem la variació que registra **GLFW**, multipliquem aquesta per un valor fix **MouseSensitivity** i incrementem els angles de rotació vertical i horitzontal amb aquests valors:

```
1  void Camera::ProcessMouseMovement(float xoffset, float yoffset)
2  {
3      xoffset *= MouseSensitivity;
4      yoffset *= MouseSensitivity;
5
6      Yaw   += xoffset;
7      Pitch += yoffset;
8
9      updateCameraVectors();
10 }
```

Finalment, per recuperar la matriu **View**, convertim el quaternió en **mat4**, creem una nova matriu de translació amb la posició de la càmera i retornem el producte entre les matrius:

```
1  glm::mat4 Camera::GetViewMatrix()
2  {
3      glm::mat4 rotate = glm::mat4_cast(orientation);
4      glm::mat4 translate = glm::mat4(1.0f);
5      translate = glm::translate(translate, -Position);
6
7      return rotate * translate;
8  }
```

9.5 Mesh

Per generar la geometria, guardar els seus atributs i textures a la targeta gràfica, i per mostrar l'objecte per pantalla fem ús de l'objecte **Mesh**. El constructor rep un llistat de **vèrtex**, d'**indexs** i textures i construeix una nova malla a l'origen de l'escena:

```
Mesh::Mesh(std::vector<Vertex> vertices, std::vector<unsigned int> indices, std::↩
    vector<Texture> textures) {
    this->vertices = std::move(vertices);
    this->indices = std::move(indices);
    this->textures = std::move(textures);
    this->position = glm::vec3(0);
    this->scale = glm::vec3(1);
    this->axis = glm::vec3(1);
    this->angle = 0;

    setupMesh();
}
```

Per guardar els diferents atributs a la targeta gràfica ho fem amb el següent mètode. La metodologia seguida s'ha explicat en detall al **Capítol 7: OpenGL**.

```
void Mesh::setupMesh() {
    // create buffers/arrays
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
    glGenBuffers(1, &EBO);

    glBindVertexArray(VAO);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);

    glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(Vertex), &vertices[0], ↩
        GL_DYNAMIC_DRAW);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(unsigned int), &↩
        indices[0], GL_DYNAMIC_DRAW);

    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)0);

    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof↩
        (Vertex, Normal));

    glEnableVertexAttribArray(2);
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof↩
        (Vertex, TexCoords));

    glBindVertexArray(0);
}
```

Per carregar les textures al **shader**, ho fem amb el següent mètode. La metodologia seguida s'ha explicat en detall al **Capítol 7: OpenGL**.

```
void Mesh::bindTextures(Shader &shader) {
    unsigned int diffuseNr  = 0;
    unsigned int i = 0;

    for(i; i < textures.size(); i++)
    {
        glActiveTexture(GL_TEXTURE0 + i);

        std::string number = "1";
        std::string name = textures[i].type;
        if(name == "texture_diffuse")
            number = std::to_string(diffuseNr++);

        if (name != "material")
        {
            shader.setInt("material.hasTexture",1);
            glUniform1i(glGetUniformLocation(shader.ID, (name +"["+= number+="]").↩
                c_str()), i);
            glActiveTexture(GL_TEXTURE0+i);
            glBindTexture(GL_TEXTURE_2D, textures[i].id);
        }
    }
}
```

Finalment, per dibuixar l'objecte per pantalla, associem el **VAO**, construïm la matriu **model** aplicant les transformacions d'escalat, rotació i translació, i dibuixem l'objecte per pantalla escollint la primitiva en funció del paràmetre **patches**:

```
void Mesh::Draw(Shader &shader, bool patches) {
    glBindVertexArray(VAO);

    glm::mat4 meshModel = glm::mat4(1.0f);

    meshModel = glm::scale(meshModel,scale);
    meshModel = glm::rotate(meshModel,angle,axis);
    meshModel = glm::translate(meshModel,position);
    shader.setMat4("model", meshModel);

    if (patches)
        glDrawElements(GL_PATCHES, indices.size(), GL_UNSIGNED_INT, 0);
    else
        glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);

    glBindVertexArray(0);
    glActiveTexture(GL_TEXTURE0);
}
```

9.6 Shader

Per llegir i compilar **shaders** disposem de dos constructors. El primer rep dues cadenes de caràcters que especifiquen la ruta del **Vertex Shader** i **Fragment Shader** respectivament, obre aquests fitxers, els llegeix i converteix el codi en una cadena de caràcters. Seguidament, crea un objecte a la targeta gràfica de tipus **shader**, associa el codi pertinent, el compila i comprova si hi ha errors. Finalment, es crea un únic programa amb els dos **shaders** i s'associa un identificador únic:

```
Shader::Shader(const char *vertexPath, const char *fragmentPath) {
    std::string vertexCode;
    std::string fragmentCode;
    std::ifstream vShaderFile;
    std::ifstream fShaderFile;
    vShaderFile.exceptions (std::ifstream::failbit | std::ifstream::badbit);
    fShaderFile.exceptions (std::ifstream::failbit | std::ifstream::badbit);
    try
    {
        vShaderFile.open(vertexPath);
        fShaderFile.open(fragmentPath);
        std::stringstream vShaderStream, fShaderStream;
        vShaderStream << vShaderFile.rdbuf();
        fShaderStream << fShaderFile.rdbuf();
        vShaderFile.close();
        fShaderFile.close();
        vertexCode   = vShaderStream.str();
        fragmentCode = fShaderStream.str();
    }
    catch (std::ifstream::failure& e)
    {
        std::cout << "ERROR::SHADER::FILE_NOT_SUCCESFULLY_READ::" <<vertexPath<<"::↩
            "<<fragmentPath
        << std::endl;
    }
    const char* vShaderCode = vertexCode.c_str();
    const char * fShaderCode = fragmentCode.c_str();
    unsigned int vertex, fragment;
    vertex = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertex, 1, &vShaderCode, NULL);
    glCompileShader(vertex);
    checkCompileErrors(vertex, "VERTEX");
    fragment = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragment, 1, &fShaderCode, NULL);
    glCompileShader(fragment);
    checkCompileErrors(fragment, "FRAGMENT");
    ID = glCreateProgram();
    glAttachShader(ID, vertex);
    glAttachShader(ID, fragment);
    glLinkProgram(ID);
    checkCompileErrors(ID, "PROGRAM");
    glDeleteShader(vertex);
    glDeleteShader(fragment);
}
```

El segon constructor fa el mateix que el primer, però especificant també la ruta pel **Tesselation Control** i **Tesselation Evaluation shaders**:

```
Shader::Shader(const char *vertexPath, const char *fragmentPath, const char *tcsPath, const char *tesPath) {
    // 1. retrieve the vertex/fragment source code from filePath
    std::string vertexCode;
    std::string fragmentCode;
    std::string tcsCode,pgCode,tesCode;
    std::ifstream vShaderFile;
    std::ifstream fShaderFile;
    std::ifstream tcShaderFile,tesShaderFile;
    // ensure ifstream objects can throw exceptions:
    vShaderFile.exceptions (std::ifstream::failbit | std::ifstream::badbit);
    fShaderFile.exceptions (std::ifstream::failbit | std::ifstream::badbit);
    tcShaderFile.exceptions (std::ifstream::failbit | std::ifstream::badbit);
    tesShaderFile.exceptions (std::ifstream::failbit | std::ifstream::badbit);
    try
    {
        vShaderFile.open(vertexPath);
        fShaderFile.open(fragmentPath);
        tcShaderFile.open(tcsPath);
        tesShaderFile.open(tesPath);
        std::stringstream vShaderStream, fShaderStream,tcShaderStream, tesShaderStream;
        vShaderStream << vShaderFile.rdbuf();
        fShaderStream << fShaderFile.rdbuf();
        tcShaderStream << tcShaderFile.rdbuf();
        tesShaderStream << tesShaderFile.rdbuf();
        vShaderFile.close();
        fShaderFile.close();
        tcShaderFile.close();
        tesShaderFile.close();
        vertexCode   = vShaderStream.str();
        fragmentCode = fShaderStream.str();
        tcsCode = tcShaderStream.str();
        tesCode = tesShaderStream.str();
    }
    catch (std::ifstream::failure& e)
    {
        std::cout << "ERROR::SHADER::FILE_NOT_SUCCESFULLY_READ::" <<vertexPath<<"::"<<fragmentPath
                  << std::endl;
    }
    const char* vShaderCode = vertexCode.c_str();
    const char * fShaderCode = fragmentCode.c_str();
    const char * tcsShaderCode = tcsCode.c_str();
    const char * tesShaderCode = tesCode.c_str();
    unsigned int vertex, fragment,tcs,tes;
    vertex = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertex, 1, &vShaderCode, NULL);
    glCompileShader(vertex);
    checkCompileErrors(vertex, "VERTEX");
    tcs = glCreateShader(GL_TESS_CONTROL_SHADER);
    glShaderSource(tcs, 1, &tcsShaderCode, NULL);
    glCompileShader(tcs);
    checkCompileErrors(tcs, "TCS");
    tes = glCreateShader(GL_TESS_EVALUATION_SHADER);
    glShaderSource(tes, 1, &tesShaderCode, NULL);
    glCompileShader(tes);
```

```
55      checkCompileErrors(tes, "TES");
56      fragment = glCreateShader(GL_FRAGMENT_SHADER);
57      glShaderSource(fragment, 1, &fShaderCode, NULL);
58      glCompileShader(fragment);
59      checkCompileErrors(fragment, "FRAGMENT");
60      ID = glCreateProgram();
61      glAttachShader(ID, vertex);
62      glAttachShader(ID, fragment);
63      glAttachShader(ID,tcs);
64      glAttachShader(ID,tes);
65      glLinkProgram(ID);
66      checkCompileErrors(ID, "PROGRAM");
67      glDeleteShader(vertex);
68      glDeleteShader(fragment);
69      glDeleteShader(tcs);
70      glDeleteShader(tes);
71  }
```

Seguidament, disposem dels següents mètodes per carregar les diferents **uniforms** al **shader**:

- Permet carregar reals al **shader**:

```
1  void Shader::setFloat(const std::string &name, float value) const {
2      glUniform1f(glGetUniformLocation(ID, name.c_str()), value);
3  }
```

- Permet carregar matrius de 4x4 al **shader**:

```
1  void Shader::setMat4(const std::string &name, glm::mat4 value) const {
2      glUniformMatrix4fv(glGetUniformLocation(ID, name.c_str()),1, GL_FALSE, ↩
           glm::value_ptr(value));
3  }
```

- Permet carregar vectors de 3 elements al **shader**:

```
1  void Shader::setVec3(const std::string &name, glm::vec3 value) const {
2      glUniform3f(glGetUniformLocation(ID, name.c_str()),value.x,value.y,value.↩
           z);
3  }
```

Finalment, quan volem utilitzar un **shader** en un moment determinat de l'execució, ho fem amb:

```
1  void Shader::use() {
2      glUseProgram(ID);
3  }
```

9.7 Entitats

Les diferents entitats que implementem a l'aplicació es defineixen amb una classe base que decideix els mètodes i atributs obligatoris que han de tenir les classes derivades:

```
1  class Entity {
2  public:
3      int id;
4      virtual ~Entity();
5      virtual glm::vec3 getPosition();
6      virtual glm::vec4 getRotation();
7      virtual glm::vec3 getScale();
8      int getType();
9
10     virtual void draw(Shader &shader, bool outlined = false, int depthMap = -1) = ↩
           0;
11     virtual std::string toString() = 0;
12     virtual void setPosition(glm::vec3 position);
13     virtual void setRotation(glm::vec4 rotation);
14     virtual void setScale(glm::vec3 scale);
15 protected:
16     glm::vec3 _position;
17     glm::vec4 _rotation;
18     glm::vec3 _scale;
19     Mesh *entityMesh;
20     Shader entityShader;
21
22     int type;
23 };
```

On cada entitat tindrà un identificador únic i un tipus específic. El tipus potser 2, que ens diu que és tracta d'una llum, o 3, que ens diu que es tracta d'un planeta.

9.7.1 Planeta

Per la construcció del planeta, ho dividim en 3 etapes. La primera, ens descriu com construïm la geometria de l'objecte, com apliquem les textures al terreny i com s'aplica il·luminació.

La segona expliquem com implementem l'atmosfera, explicant detalladament el funcionament del **shader** que s'encarrega de crear l'efecte.

La tercera que descriu l'esfera aquàtica que compon els diferents cossos aquàtics que trobem al planeta.

9.7.1.1 Geometria

Construïm el planeta des del **parser** amb un radi i un nombre de subdivisions determinat, un booleà que ens diu si disposa d'atmosfera, un vector amb la ruta de les textures que componen el planeta, un vector de 3 elements que descriu la posició, un booleà que determina si tindrà o no cossos aquàtics i una funció de soroll configurada:

```
Planet::Planet(float radius, int nSeg, bool hasAtmos, float maxHeight,const std::vector<std::string> &path,
               glm::vec3 position, bool hasWater, FastNoiseLite Noise):
               radius(radius), nSeg(nSeg), hasAtmos(hasAtmos),
               maxHeight(maxHeight), path(path),hasWater(hasWater)
{
    _position = position;
    this->type = 3;
    drawEffects = true;
    bindPlanetTextures();
    this->noise = Noise;
    this->noise.SetSeed(seed);
    setupMesh();
}
```

Seguidament, construïm la geometria, on definim un objecte **CubeSphere**, que és una classe que ens ajuda a generar els vèrtexs i indexes, obtenim el punt més alt i més baix del terreny, que ens ajudarà a aplicar les textures al terreny i construïm una nova **Mesh**:

```
void Planet::setupMesh() {
    auto *cubeSphere = new Cubesphere(radius,nSeg,true,minValue);
    cubeSphere->setupNoise(maxHeight,noise,minValue);
    highestPoint = cubeSphere->getHPoint();
    lowestPoint = cubeSphere->getLPoint();
    entityMesh = new Mesh(cubeSphere->getVertexList(),cubeSphere->getIndices(),textures);
    entityMesh->position = _position;
    delete cubeSphere;
}
```

La classe **CubeSphere** encapsula els diferents mètodes per construir el llistat de vèrtex i indexes. El constructor determina quants vèrtexs hi haurà per fila en funció del nombre de subdivisions i com un cub consta de 6 cares, quants vèrtexs tindrà per cara. També inicialitza els valors de **lPoint** (punt més baix) i **hPoint**, punt més alt:

```
Cubesphere::Cubesphere(float radius, int sub, bool smooth, float minValue) : radius(radius), subdivision(sub), smooth(smooth), interleavedStride(32)
{
    vertexCountPerRow = (unsigned int)pow(2, sub) + 1;
    vertexCountPerFace = vertexCountPerRow * vertexCountPerRow;
    std::cout<<vertexCountPerRow<<std::endl;
```

```
6     lPoint = 0;
7     hPoint = 0;
8     this->minValue = minValue;
9 }
```

Seguidament, és crida a la funció **setupNoise**, que donat una alçada màxima i una funció de soroll, i comença a construir els vèrtexs:

```
1 void Cubesphere::setupNoise(float height, FastNoiseLite noise, float minValue) {
2     clearArrays();
3     this->noise = noise;
4     this->height = height;
5     this->minValue = minValue;
6     buildVerticesSmooth();
7 }
```

Per construir tots els vèrtexs, inicialment definim les diferents cares que formaran el cub. +X i -X construiran les 2 cares laterals del cub, +Y i -Y les 2 cares verticals, i +Z i -Z les cares frontals. Construïm el llistat de posicions del vèrtex de la cara +X definint el mètode **getUnitPositiveX** que construeix els vèrtexs fent la intersecció entre 2 plans esfèrics. Veure figura 9.4:

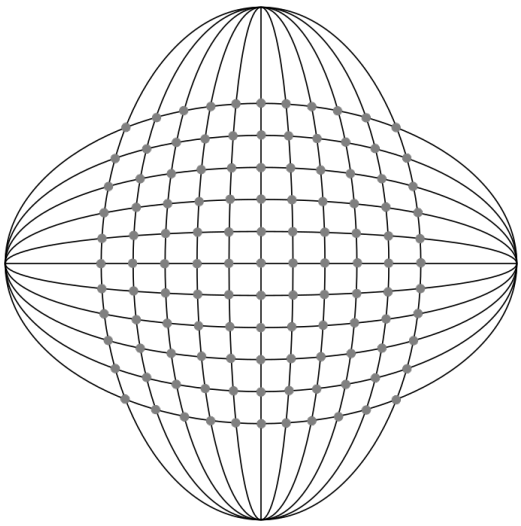


Figura 9.4: Primera cara de l'esfera cúbica [Ahn 021]

```
1 std::vector<float> Cubesphere::getUnitPositiveX(unsigned int pointsPerRow)
2 {
3     const float DEG2RAD = acos(-1) / 180.0f;
4
5     std::vector<float> vertices;
6     float n1[3];        // normal of longitudinal plane rotating along Y-axis
7     float n2[3];        // normal of latitudinal plane rotating along Z-axis
8     float v[3];         // direction vector intersecting 2 planes, n1 x n2
```

```
9     float a1;           // longitudinal angle along y-axis
10    float a2;           // latitudinal angle
11    float scale;
12
13    // rotate latitudinal plane from 45 to -45 degrees along Z-axis
14    for(unsigned int i = 0; i < pointsPerRow; ++i)
15    {
16        // normal for latitudinal plane
17        a2 = DEG2RAD * (45.0f - 90.0f * i / (pointsPerRow - 1));
18        n2[0] = -sin(a2);
19        n2[1] = cos(a2);
20        n2[2] = 0;
21
22        // rotate longitudinal plane from -45 to 45 along Y-axis
23        for(unsigned int j = 0; j < pointsPerRow; ++j)
24        {
25            // normal for longitudinal plane
26            a1 = DEG2RAD * (-45.0f + 90.0f * j / (pointsPerRow - 1));
27            n1[0] = -sin(a1);
28            n1[1] = 0;
29            n1[2] = -cos(a1);
30
31            // find direction vector of intersected line, n1 x n2
32            v[0] = n1[1] * n2[2] - n1[2] * n2[1];
33            v[1] = n1[2] * n2[0] - n1[0] * n2[2];
34            v[2] = n1[0] * n2[1] - n1[1] * n2[0];
35
36            // normalize direction vector
37            scale = Cubesphere::computeScaleForLength(v, 1);
38
39            v[0] *= scale;
40            v[1] *= scale;
41            v[2] *= scale;
42
43            vertices.push_back(v[0]);
44            vertices.push_back(v[1]);
45            vertices.push_back(v[2]);
46        }
47    }
48
49    return vertices;
50 }
```

Amb el llistat de reals obtingut, acabem de construir la cara +X, definint **s** i **t** que determinen les textures de coordenades on, per cada fila **i**, cada columna **j** i el nombre total de vèrtex per fila k, definim s i t com a:

$$s = \frac{j}{(k-1)*k} \tag{9.1}$$

$$t = \frac{i}{(k-1)*k} \tag{9.2}$$

Per calcular les posicions x,y i z de cada vèrtex, multipliquem aquestes pel radi i sumem el producte d'un vector direccional entre l'origen de l'esfera i el vèrtex amb la funció de soroll computada en aquell vèrtex:

```
void Cubesphere::buildVerticesSmooth()
{
    // generate unit-length verties in +X face
    std::vector<float> unitVertices = Cubesphere::getUnitPositiveX(↩
        vertexCountPerRow);
    std::cout<<"Vertex count per row: "<<vertexCountPerRow<<std::endl;
    // clear memory of prev arrays
    clearArrays();

    float x, y, z, s, t;
    int k = 0, k1, k2;

    // build +X face
    for(unsigned int i = 0; i < vertexCountPerRow; ++i)
    {
        k1 = i * vertexCountPerRow;     // index for curr row
        k2 = k1 + vertexCountPerRow;    // index for next row
        t = (float)i / (vertexCountPerRow - 1) * vertexCountPerRow;

        for(unsigned int j = 0; j < vertexCountPerRow; ++j, k += 3, ++k1, ++k2)
        {
            x = unitVertices[k];
            y = unitVertices[k+1];
            z = unitVertices[k+2];

            s = ((float)j / (vertexCountPerRow - 1)) *vertexCountPerRow ;
            Vertex auxVertex;
            glm::vec3 vertexPos = glm::vec3(x*radius,y*radius,z*radius);
            float minNoise = noise.GetNoise(x*radius,y*radius,z*radius);
            float noiseLevel = minNoise * height;
            if (noiseLevel > hPoint)
                hPoint = noiseLevel;
            if (noiseLevel < lPoint)
                lPoint = noiseLevel;
            glm::vec3 dir = glm::normalize(glm::vec3(0) - vertexPos);
            auxVertex.Position = vertexPos + dir*noiseLevel;
            auxVertex.Normal = glm::vec3(x,y,z);
            auxVertex.TexCoords = glm::vec2(s,t);

            vertexList.push_back(auxVertex);
            addVertex(x*radius, y*radius, z*radius);
            addNormal(x, y, z);
            addTexCoord(s, t);

            // add indices
            if(i < (vertexCountPerRow-1) && j < (vertexCountPerRow-1))
            {
                addIndices(k1, k2, k1+1);
                addIndices(k1+1, k2, k2+1);

            }
        }
    }
```

Construïda la primera cara, construïm la resta en funció d'aquesta per evitar repetir operacions, de manera que:

- Construïm la cara -X negant les components x i z de la cara +X.
- Construïm la cara +Y intercanviant les components x per y, y per -z i z per

-x de la cara +X.

- Construïm la cara -Y intercanviant les components x per -y, y per z i z per -x de la cara +X.
- Construïm la cara +Z intercanviant les components x per z i z per -x de la cara +X.
- Construïm la cara -Z intercanviant les components x per -z i z per +x de la cara +X.

```
    unsigned int startIndex;
    int vertexSize = (int)vertices.size();
    int indexSize = (int)indices.size();
    int lineIndexSize = (int)lineIndices.size();

    std::vector<int> index1 = {0,1,2};
    std::vector<Vertex> vertices0;
    std::vector<unsigned int> indices0;
    buildFace(glm::vec3(-1,1,-1), index1,vertexSize,indexSize,vertices0,indices0);;
    finishFaceComputation(vertices0);

    std::vector<int> index2 = {2,0,1};
    std::vector<Vertex> vertices1;
    std::vector<unsigned int> indices1;
    buildFace(glm::vec3(-1,1,-1), index2,vertexSize,indexSize,vertices1,indices1);
    finishFaceComputation(vertices1);

    std::vector<int> index3 = {2,0,1};
    std::vector<Vertex> vertices2;
    std::vector<unsigned int> indices2;
    buildFace(glm::vec3(-1,-1,1), index3,vertexSize,indexSize,vertices2,indices2);
    finishFaceComputation(vertices2);

    std::vector<Vertex> vertices3;
    std::vector<unsigned int> indices3;
    std::vector<int> index4 = {2,1,0};
    buildFace(glm::vec3(-1,1,1), index4,vertexSize,indexSize,vertices3,indices3);
    finishFaceComputation(vertices3);

    std::vector<int> index5 = {2,1,0};
    std::vector<Vertex> vertices4;
    std::vector<unsigned int> indices4;
    buildFace(glm::vec3(1,1,-1), index5,vertexSize,indexSize,vertices4,indices4);
    finishFaceComputation(vertices4);

    indices.insert(indices.end(),indices0.begin(),indices0.end());
    indices.insert(indices.end(),indices1.begin(),indices1.end());
    indices.insert(indices.end(),indices2.begin(),indices2.end());
    indices.insert(indices.end(),indices3.begin(),indices3.end());
    indices.insert(indices.end(),indices4.begin(),indices4.end());

    buildInterleavedVertices();
    computeNormals();
}
```

Construïdes les 6 cares, construïm els vèrtexs que intercalen les cares:

```
void Cubesphere::buildInterleavedVertices()
{
    std::vector<float>().swap(interleavedVertices);

    std::size_t i, j;
    std::size_t count = vertices.size();
    for(i = 0, j = 0; i < count; i += 3, j += 2)
    {
        Vertex auxVert;
        interleavedVertices.push_back(vertices[i]);
        interleavedVertices.push_back(vertices[i+1]);
        interleavedVertices.push_back(vertices[i+2]);
        auxVert.Position = glm::vec3(vertices[i],vertices[i+1],vertices[i+2]);
        interleavedVertices.push_back(normals[i]);
        interleavedVertices.push_back(normals[i+1]);
        interleavedVertices.push_back(normals[i+2]);
        auxVert.Normal = glm::vec3(normals[i],normals[i+1],normals[i+2]);
        interleavedVertices.push_back(texCoords[j]);
        interleavedVertices.push_back(texCoords[j+1]);
        auxVert.TexCoords = glm::vec2(texCoords[j],texCoords[j+1]);
        vertexList.push_back(auxVert);
    }
}
```

Finalment, calculem les normals fent ús del vector d'índexs, on per cada triangle, obtenim un vector direccional entre els seus vèrtexs i fem el producte vectorial per obtenir un vector perpendicular, veure figura 9.5:

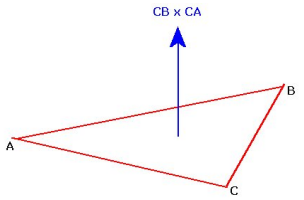


Figura 9.5: Vector perpendicular a la cara d'un triangle

```
void Cubesphere::computeNormals() {
    int count = 0;
    for (int i = 0; i < indices.size();i+=3)
    {
        unsigned int iA = indices[i];
        unsigned int iB = indices[i+1];
        unsigned int iC = indices[i+2];
        auto edgeAB = vertexList[iB].Position - vertexList[iA].Position;
        auto edgeAC = vertexList[iC].Position - vertexList[iA].Position;
        auto norm = glm::normalize(glm::cross(edgeAB,edgeAC));
        vertexList[iA].Normal = norm;
        vertexList[iB].Normal = norm;
        vertexList[iC].Normal = norm;
    }
}
```

9.7.1.2 Textures i il·luminació

Per dibuixar el planeta per pantalla, primer habilitem el **shader** i posem les **uniforms** necessàries per permetre interpolar les textures per latitud i per alçada. Fet això, dibuixem la geometria per pantalla, i en cas que els efectes estiguin habilitats i el planeta tingui efectes, els mostrem per pantalla:

```
1  void Planet::draw(Shader &shader, bool outlined, int depthMap) {
2
3      shader.use();
4
5      shader.setFloat("hPoint",highestPoint + hPointOffset);
6      shader.setFloat("lPoint",lowestPoint - lPointOffset);
7      shader.setFloat("pRadius",radius);
8
9      shader.setFloat("depthBlend",depthBlend);
10     shader.setVec3("pPosition",_position);
11     shader.setInt("nBiomes",biomes.size());
12     shader.setInt("nTextures",path.size());
13
14     for (int i = 0; i < biomes.size();i++)
15     {
16         shader.setFloat("biomes["+std::to_string(i)+"].latStart",biomes[i].latStart↩
               );
17         shader.setFloat("biomes["+std::to_string(i)+"].latEnd",biomes[i].latEnd);
18         shader.setInt("biomes["+std::to_string(i)+"].nTextBiome",biomes[i].↩
               textHeight.size());
19         for (int j = 0; j < biomes[i].textHeight.size();j++)
20         {
21             shader.setInt("biomes["+std::to_string(i)+"].textIndex["+std::to_string↩
                   (j)+"].index",biomes[i].textHeight[j].index);
22             shader.setFloat("biomes["+std::to_string(i)+"].textIndex["+std::↩
                   to_string(j)+"].hStart",biomes[i].textHeight[j].hStart);
23             shader.setFloat("biomes["+std::to_string(i)+"].textIndex["+std::↩
                   to_string(j)+"].hEnd",biomes[i].textHeight[j].hEnd);
24         }
25     }
26
27     shader.setInt("nTextures",path.size());
28     entityMesh->Draw(shader,true,true);
29
30     if (drawEffects)
31     {
32         if (hasWater)
33             water->draw(shader,outlined,depthMap);
34         if (hasAtmos)
35             skyDome->draw(shader,outlined,-1);
36     }
37 }
```

El **shader** del planeta disposa de 2 parts que descriurem a continuació:

Terrain Vertex Shader

On especifiquem la versió, obtenim els diferents atributs que ens arriben del **VAO** de la **Mesh**, especifiquem les diferents sortides, calculem la posició del fragment multiplicant la matriu model amb la posició del vèrtex i calculem la posició global multiplicant la matriu de projecció, amb la matriu de vista i amb la posició del fragment:

```
#version 410
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoords;

out vec3 FragPos;
out vec3 Normal;
out vec2 TexCoords;
out vec3 LocalPos;
out vec4 worldPos;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    FragPos = vec3(model * vec4(aPos, 1.0));
    LocalPos = aPos;
    Normal = aNormal;
    TexCoords = aTexCoords;
    worldPos = projection * view * vec4(FragPos, 1.0);
    gl_Position = worldPos;
}
```

Terrain Fragment Shader

Inicialment definim les estructures que ens permetran definir el color final del fragment. L'estructura **DirLight**, explicat a Capítol 8: Light i les estructures **TextHeight** i **Biome**, explicades al **Capítol 8: Planet**.

```
#version 410 core

#define maxText 8
#define maxBiomes 6

struct DirLight {
    vec3 direction;

    vec3 ambient;
    vec3 diffuse;
};

struct TextHeight {
    int index;//4
```

```
15      float hStart;//4
16      float hEnd;//4
17  };
18
19  struct Biome {
20      float latStart;//4  0
21      float latEnd;//4    4
22      int nTextBiome;//4  8
23      TextHeight[maxText] textIndex;//12*16=192    200
24  };
```

Seguidament, definim les entrades que ens arriben del **Vertex Shader** i les diferents **uniforms** que descriuen el nombre de textures i biomes, la llum direccional, el vector de textures i biomes, el radi del planeta, un factor que determina la profunditat en la que s'interpolen les textures, el punt més alt i el punt més baix, i la posició origen del planeta:

```
1  in vec3 FragPos;
2  in vec3 Normal;
3  in vec2 TexCoords;
4  in vec3 LocalPos;
5  in vec4 worldPos;
6
7  uniform int nTextures;
8  uniform int nBiomes;
9  uniform DirLight dirLight;
10 uniform sampler2D texture_diffuse[maxText];
11 uniform Biome biomes[maxBiomes];
12 uniform float pRadius;
13 uniform float depthBlend;
14 uniform float hPoint;
15 uniform float lPoint;
16 uniform vec3 pPosition;
17
18 out vec4 FragColor;
19
20 const float PI = 3.14159265359;
```

El programa principal carrega les textures, calcula l'alçada i latitud a la qual es troba el fragment, interpola les textures i calcula la quantitat de llum que rep el fragment:

```
1  void main()
2  {
3      vec3 norm = Normal;
4      vec4 text = vec4(0,0,0,1);
5      vec4 loadedTextures[maxText];
6      for (int i = 0; i < nTextures; i++)
7      {
8          loadedTextures[i] = textureNoTile(texture_diffuse[i],TexCoords);
9      }
10
11     vec3 dir = pPosition-FragPos;
12     float absHeigth = (abs(lPoint) + abs(hPoint));
13     float height = abs((length(dir) - pRadius) - lPoint) / absHeigth;
```

```
14
15      vec3 pos = (dir)/pRadius;
16      float lat = atan(pos.y,sqrt(pos.x*pos.x+pos.z*pos.z)) + PI/2.0;//convert to ↩
            positive 0..3.14
17      lat = lat / PI;
18
19      for (int i = 0; i < nBiomes; i++)
20      {
21          text += biomeInterpolation(loadedTextures,i,lat,height);
22      }
23
24      vec3 result = vec3(0.0);
25      result += CalcDirLight(dirLight, norm,text);
26      FragColor = vec4(_ACESFilmicToneMapping(result), 1.0);
27  }
```

La funció **textureNoTile** rep una textura i les seves coordenades, i ens permet crear un patró de repetició més natural quan repetim una textura sobre el terreny, transformant les coordenades de textura per tal d'aplicar rotacions diferents a cada repetició, veure figura 9.6:

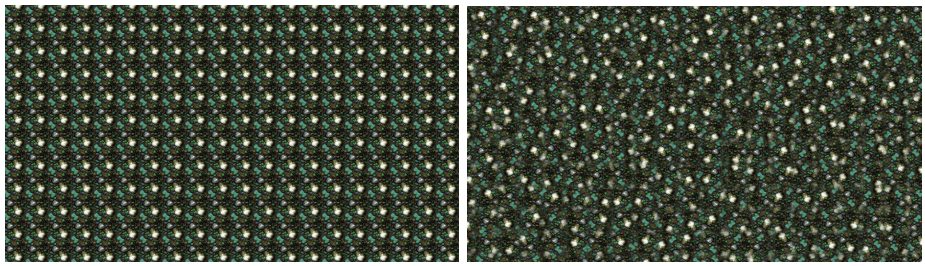


Figura 9.6: Patró de repetició per defecte / aplicant l'algoritme [Quilez]

```
1  vec4 textureNoTile( sampler2D samp, in vec2 uv )
2  {
3      ivec2 iuv = ivec2( floor( uv ) );
4      vec2 fuv = fract( uv );
5      vec4 ofa = hash4( iuv + ivec2(0,0) );
6      vec4 ofb = hash4( iuv + ivec2(1,0) );
7      vec4 ofc = hash4( iuv + ivec2(0,1) );
8      vec4 ofd = hash4( iuv + ivec2(1,1) );
9      vec2 ddx = dFdx( uv );
10     vec2 ddy = dFdy( uv );
11
12     ofa.zw = sign( ofa.zw-0.5 );
13     ofb.zw = sign( ofb.zw-0.5 );
14     ofc.zw = sign( ofc.zw-0.5 );
15     ofd.zw = sign( ofd.zw-0.5 );
16
17     vec2 uva = uv*ofa.zw + ofa.xy, ddxa = ddx*ofa.zw, ddya = ddy*ofa.zw;
18     vec2 uvb = uv*ofb.zw + ofb.xy, ddxb = ddx*ofb.zw, ddyb = ddy*ofb.zw;
19     vec2 uvc = uv*ofc.zw + ofc.xy, ddxc = ddx*ofc.zw, ddyc = ddy*ofc.zw;
20     vec2 uvd = uv*ofd.zw + ofd.xy, ddxd = ddx*ofd.zw, ddyd = ddy*ofd.zw;
```

```
21
22      vec2 b = smoothstep( 0.25,0.75, fuv );
23
24      return mix( mix( textureGrad( samp, uva, ddxa, ddya ),
25      textureGrad( samp, uvb, ddxb, ddyb ), b.x ),
26      mix( textureGrad( samp, uvc, ddxc, ddyc ),
27      textureGrad( samp, uvd, ddxd, ddyd ), b.x), b.y );
28  }
```

Amb la funció **biomeInterpolation**, que rep com a paràmetre un vector amb les textures que formen el terreny, l'índex del bioma concret i la latitud i alçada a la que es troba el vèrtex, comprovem si aquell bioma es troba dins de l'espai restringit per la latitud i apliquem l'algoritme d'interpolació per alçada:

```
1  vec4 biomeInterpolation(vec4 loadedTextures[maxText], int index, float lat, float ↩
       height)
2  {
3      vec4 result = vec4(0.0);
4      if (lat <= biomes[index].latStart && lat > biomes[index].latEnd)
5      {
6          float a = smoothstep(biomes[index].latStart, biomes[index].latEnd,lat);
7          if (index == 0)
8              result += createTerrainTextHeight(loadedTextures,biomes[0], height);
9          else
10             result += vec4(blend(createTerrainTextHeight(loadedTextures,biomes[↩
                   index - 1], height), 1-a, createTerrainTextHeight(loadedTextures,↩
                   biomes[index], height), a), 0.0);
11     }
12     return result;
13 }
```

La funció **createTerrainTextHeight**, que rep el llistat de textures, el bioma en el qual ens trobem i l'alçada del vèrtex i interpola les textures segons l'alçada a la qual es troben:

```
1  vec4 createTerrainTextHeight(vec4 loadedTextures[maxText],Biome biome, float height↩
       ){
2      vec4 terrainColor = vec4(0.0, 0.0, 0.0, 1.0);
3      int nText = biome.nTextBiome;
4      for (int i = 0; i < nText; i++)
5      {
6          float a = smoothstep(biome.textIndex[i].hStart,biome.textIndex[i].hEnd,↩
               height);
7          if (height >= biome.textIndex[i].hStart && height <  biome.textIndex[i].↩
               hEnd)
8          {
9              if (i == 0)
10                 terrainColor += vec4(blend(loadedTextures[biome.textIndex[0].index↩
                       ],1-a,loadedTextures[biome.textIndex[0].index],a),0.0);
11             else
12                 terrainColor += vec4(blend(loadedTextures[biome.textIndex[i - 1].↩
                       index],1-a,loadedTextures[biome.textIndex[i].index],a),0.0);
13         }
14     }
15     return terrainColor;
16 }
```

La funció **blend** rep dues textures i, per cada textura, un real que determina la seva opacitat. Primer, en funció de la transparència de la textura, calculem l'opacitat màxima i per cada textura, calculem l'opacitat individual. El factor **depthBlend** ens permet modificar l'opacitat màxima, veure figura 9.7.1.2.

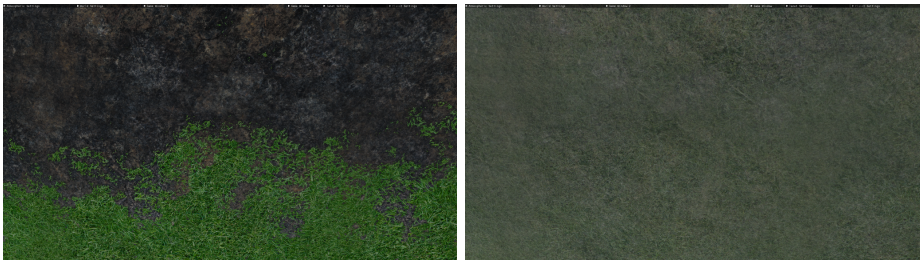


Figura 9.7: Barreja de textures amb depthBlend = 0 i depthBlend = 1 [Mishkinis 2013]

```
vec3 blend(vec4 texture1, float a1, vec4 texture2, float a2)
{
    float ma = max(texture1.a + a1, texture2.a + a2) - depthBlend;
    float b1 = max(texture1.a + a1 - ma, 0);
    float b2 = max(texture2.a + a2 - ma, 0);

    return (texture1.rgb * b1 + texture2.rgb * b2) / (b1 + b2);
}
```

9.7.1.3 Atmosfera

Construïm l'atmosfera assignant tots els paràmetres des del **parser**, inicialitzant la geometria fent servir una **Cubesphere** i el seu **shader** corresponent:

```
Atmosphere::Atmosphere(float planetRadius, float atmosRadius, Camera *cam, float kR, float kM, float e, float h,float l, float atmosColor[3], float gM, float numOutScatter, float numInScatter, float scale,glm::vec3 position)
    : planetRadius(planetRadius), atmosRadius(atmosRadius), cam(cam), k_r(kR), k_m(kM),e(e),H(h),L(l),g_m(gM), numOutScatter(numOutScatter), numInScatter(numInScatter), scale(scale) {
    for (int i = 0; i < 3; i++)
        this->atmosColor[i] = atmosColor[i];
    this->planetRadius = planetRadius;
    this->atmosRadius = atmosRadius;
    this->cam = cam;
    Cubesphere cubesphere(atmosRadius,4,true);
    cubesphere.setupNoise(0,NULL);
    entityMesh = new Mesh(cubesphere.vertexList,cubesphere.getIndices(),"","");
    entityShader = Shader("../Shaders/skydomeVertex.shader", "../Shaders/skydomeFragment.shader");
    _position = position;
}
```

Per dibuixar l'atmosfera, especifiquem a **OpenGL** que es tracta d'un objecte amb transparència amb **GL_BLEND** i li especifiquem que obtingui aquest valor de la font. Seguidament, habilitem el **shader** i especifiquem les **uniforms** que permetran crear l'efecte atmosfèric. Finalment, renderitzem 2 vegades la geometria, una per cara, per obtenir un efecte més visual des de l'espai i des de l'interior del planeta:

```
void Atmosphere::draw(Shader &shader) {
    renderGui();
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA,GL_SRC_ALPHA);
    glm::mat4 view = this->cam->GetViewMatrix();
    glm::mat4 projection;
    projection = glm::perspective(glm::radians(this->cam->Zoom), (float)setting_scrWidth/setting_scrHeight, setting_near, setting_far);
    glm::mat4 cameraModel(1.0f);
    entityShader.use();
    entityShader.setMat4("view",view);
    entityShader.setMat4("projection",projection);
    entityShader.setMat4("model", cameraModel);
    entityShader.setVec3("cameraPosition",cam->Position);
    entityShader.setVec3("planetPosition",_position);
    entityShader.setFloat("planetRadius",planetRadius);
    entityShader.setFloat("atmosRadius",atmosRadius);
    entityShader.setVec3("sunDir",sun->getDirVector());
    entityShader.setFloat("H",H);
    entityShader.setFloat("L",L);
    entityShader.setFloat("K_R",k_r);
    entityShader.setFloat("K_M",k_m);
    entityShader.setFloat("E",e);
    entityShader.setVec3("C_R",glm::normalize(glm::vec3(atmosColor[0],atmosColor
```

```
        [1],atmosColor[2])));
    entityShader.setFloat("G_M",g_m);
    entityShader.setFloat("MAX",viewDistance);
    entityShader.setFloat("fNumOutScatter",numOutScatter);
    entityShader.setFloat("fNumInScatter",numInScatter);
    glFrontFace(GL_CW);
    entityMesh->Draw(atmosShader,outlined,depthMap,false,false);
    glFrontFace(GL_CCW);
    entityMesh->Draw(atmosShader,outlined,depthMap,false,false);
    glDisable(GL_BLEND);
    shader.use();
}
```

Skydome Vertex Shader

Equivalent al **Terrain Vertex Shader** que hem explicat anteriorment:

```
#version 410
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoords;

out vec3 FragPos;
out vec4 worldPos;
out vec3 Normal;
out vec2 TexCoords;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    FragPos = vec3(model * vec4(aPos, 1.0));
    Normal = aNormal;
    worldPos = projection * view * vec4(FragPos, 1.0);
    TexCoords = aTexCoords;
    gl_Position = worldPos;
}
```

Skydome Fragment Shader

Inicialment definim les diferents **uniforms** que ens permetran definir i modificar en temps d'execució la dispersió de la llum en el fragment en el qual ens trobem:

```
#version 410 core
uniform vec3 cameraPosition;
uniform vec3 planetPosition;
uniform vec3 sunDir;
uniform vec3 C_R;

uniform float K_R;
uniform float K_M;
uniform float E;
uniform float G_M;
uniform float MAX;
uniform float H;
uniform float L;
uniform float planetRadius;
uniform float atmosRadius;

uniform float fNumOutScatter;
uniform float fNumInScatter;

in vec4 worldPos;
in vec4 FragPos;
in vec3 Normal;

out vec4 FragColor;

const float PI = 3.14159265359;
```

Al programa principal, calculem un vector direccional entre la càmera i la posició del fragment i calculem el punt d'intersecció amb la capa superior i inferior de l'atmosfera. Amb aquesta informació, calculem el color que tindrà el fragment en funció de la dispersió:

```
void main (void)
{
    vec3 dir = normalize(cameraPosition - (FragPos.xyz));
    vec2 e = rayIntersection(cameraPosition, dir, atmosRadius);
    vec2 f = rayIntersection(cameraPosition, dir, planetRadius);

    e.y = min(e.y, f.x);

    vec3 I = inScatter(cameraPosition, dir, e, -sunDir);
    FragColor = vec4(I, 1.0);
}
```

Per calcular la intersecció, la funció **rayIntersection** rep la posició de la càmera, la direcció entre el fragment i la càmera i un radi determinat i ens dona la distància al punt d'entrada i sortida de l'atmosfera:

```
vec2 rayIntersection(vec3 p, vec3 dir, float radius ) {
    float b = dot( p, dir );
    float c = dot( p, p ) - (radius * radius);

    float d = b * b - c;
    if ( d < 0.0 ) {
        return vec2( MAX, -MAX );
    }
    d = sqrt( d );

    float near = -b - d;
    float far = -b + d;

    return vec2(near, far);
}
```

Aquesta funció, donada la posició de la càmera, el vector direccional que forma aquesta amb el fragment, les distàncies entre el punt d'entrada i sortida de l'atmosfera i la direcció de llum, descriu quanta llum és afegida al raig que passa per l'atmosfera en funció de l'efecte de dispersió causat per la llum:

```
vec3 inScatter(vec3 o, vec3 dir, vec2 e, vec3 l) {
    float len = (e.y - e.x) / fNumInScatter;
    vec3 step = dir * len;
    vec3 p = o + dir * e.x;
    vec3 v = p + dir * (len * 0.5);

    vec3 sum = vec3(0.0);
    for(int i = 0; i < fNumInScatter; i++) {
        vec2 f = rayIntersection(v, l, atmosRadius);
        vec3 u = v + l * f.y;
        float n = (optic(p, v) + optic(v, u))*(PI * 4.0);
        sum += density(v)* exp(-n * ( K_R * C_R + K_M ));
        v += step;
    }

    sum *= len * SCALE_L;
    float c = dot(dir, -l);
    float cc = c * c;
    return sum * ( K_R * C_R * rayleighPhase( cc ) + K_M * miePhase( G_M, c, cc ) )↩
         * E;
}
```

Les funcions **rayleighPhase** i **miePhase** decideixen el color final de la llum implementant la funció **Phase**:

$$F(\theta, g) = \frac{3(1-g^2)}{2(2+g)} \frac{1+\dot{cos}^2\theta}{(1+g^2-2gcos\theta)^{\frac{3}{2}}} \tag{9.3}$$

On valors negatius de g dispersen més llum cap a la càmera i positius cap a la font de llum. Per la dispersió **RayLeigh**, $g = 0$, simplificant considerablement l'equació:

```
float miePhase( float g, float c, float cc ) {
    float gg = g * g;

    float a = ( 1.0 - gg ) * ( 1.0 + cc );

    float b = 1.0 + gg - 2.0 * g * c;
    b *= sqrt( b );
    b *= 2.0 + gg;

    return ( 3.0 / 8.0 / PI ) * a / b;
}

float rayleighPhase( float cc ) {
    return 3/2 * ( 1.0 + cc );
}
```

9.7.1.4 Esfera aquàtica

Construïm l'objecte que defineix l'esfera aquàtica assignant els diferents atributs, definint dues textures, una per l'efecte de distorsió i l'altre que apliquem a distàncies llunyanes quan no hi ha efecte de reflexió, i generem la geometria partint d'una **CubeSphere** com hem fet anteriorment:

```
WaterSphere::WaterSphere(float planetRadius, float waterRadius, Camera *cam, glm::↩
    vec3 position) {
    this->planetRadius = planetRadius;
    this->cam = cam;
    entityShader = Shader("../Shaders/waterShaderVertex.shader","../Shaders/↩
        waterShaderFragment.shader");
    this->position = position;
    Cubesphere cubesphere(waterRadius,6,true);
    cubesphere.setupNoise(0,NULL);
    std::vector<Texture> textv;
    Texture text1;
    text1.type = "texture_diffuse";
    text1.id = TextureFromFile("/Planet/dudv.png","../Textures/",false,true);
    text1.path = "/Planet/dudv.png";
    Texture text2;
    text2.type = "texture_diffuse";
    text2.id = TextureFromFile("/Planet/waterTile2.jpg","../Textures/",false,true);
    text2.path = "/planet/reflexion1.jpg";
    textv.push_back(text1);
    textv.push_back(text2);
    entityMesh = new Mesh(cubesphere.vertexList,cubesphere.getIndices(),textv);
    entityMesh->position = position;
}
```

Una vegada el **Renderer** hagi creat la textura amb l'efecte de reflexió, li fem arribar a l'**entityMesh** amb el següent mètode:

```
void WaterSphere::setWaterTexture(unsigned int waterText) {
    if (!isset)
    {
        Texture text;
        text.path ="";
        text.type ="texture_diffuse";
        text.id = waterText;
        entityMesh->textures.push_back(text);
        isset = true;
    }
}
```

Per renderitzar l'objecte per pantalla, indiquem a **OpenGL** que es tracta d'un objecte transparent, habilitem el **shader**, carreguem les **uniforms** i dibuixem la **Mesh** i la font de llum:

```
void WaterSphere::draw(Shader &shader, bool outlined, int depthMap) {
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
```

```
4     entityShader.use();
5     glm::mat4 view = this->cam->GetViewMatrix();
6     glm::mat4 projection;
7     projection = glm::perspective(glm::radians(this->cam->Zoom), (float)↩
          setting_scrWidth/setting_scrHeight, setting_near, setting_far);
8     glm::mat4 cameraModel(1.0f);
9     entityShader.setMat4("view",view);
10    entityShader.setMat4("projection",projection);
11    entityShader.setMat4("model", cameraModel);
12    entityShader.setVec3("viewPos",cam->Position);
13    entityShader.setFloat("shininess",12.80f);
14    entityShader.setVec3("planetOrigin",position);
15    if (moveFactor < 1.0)
16        moveFactor += waveSpeed * cam->deltaTime;
17    else
18        moveFactor = 0.0;
19    entityShader.setFloat("waveSpeed",moveFactor);
20    entityMesh->Draw(entityShader,false,false);
21    sun->draw(entityShader,outlined,depthMap);
22    glDisable(GL_BLEND);
23 }
```

El **WaterSphere Vertex Shader** és equivalent al de l'atmosfera. Per tant, passem a definir directament el **WaterSphere Fragment Shader**.
Inicialment, definim els elements d'entrada del **Vertex Shader**, un llistat de textures que tindrà la textura de l'escena per la reflexió, la textura per aplicar l'efecte de distorsió i la textura a mostrar quan no hi ha efecte de reflexió, la llum direccional, la posició de la càmera, un real que determina la velocitat de les onades i un real que determina com de brillant és l'efecte **specular**:

```
1  #version 430 core
2
3  in vec2 TexCoord;
4  in vec4 worldPos;
5  in vec3 Normal;
6  in vec3 FragPos;
7
8  uniform sampler2D texture_diffuse[3];
9
10 struct DirLight {
11     vec3 direction;
12     vec3 ambient;
13     vec3 diffuse;
14     vec3 specular;
15 };
16
17 uniform DirLight dirLight;
18 uniform vec3 viewPos;
19 uniform float waveSpeed;
20 uniform float shininess;
21 out vec4 FragColor;
22 const float waveStrength = 0.2;
```

El programa principal inicialment calcula el vector direccional i la distància entre la càmera i el fragment. Seguidament, calculem les coordenades de textures **NDC** que ens permet aplicar la textura a l'àrea que ocupa la geometria dins de la projecció de la càmera.

Fet això, carreguem els components vermell i verd de la textura de distorsió, ho multipliquem per la constant **waveStrength** i incrementem les coordenades de reflexió i les coordenades de la textura llunyana amb aquest valor.

Amb les coordenades de textura definides per les 2 textures, les carreguem, les interpolem en funció de la distància entre la càmera i el fragment, calculem la llum i definim el color amb la transparència en funció de la distància:

```
1  void main()
2  {
3      vec3 viewDir = normalize(viewPos - FragPos);
4      float dist = distance(FragPos,viewPos);
5
6      vec2 ndc = ((worldPos.xy/worldPos.w)/2.0)+0.5;
7      vec2 distorsion = (texture2D(texture_diffuse[0],vec2(TexCoord.x + waveSpeed,↩
           TexCoord.y)*5).rg * 2.0 - 1.0)*waveStrength;
8      vec2 reflectCoord = vec2(ndc.x, 1.0 - ndc.y);
9      vec2 diffCoord = TexCoord/2;
10
11     reflectCoord += distorsion;
12     diffCoord += distorsion;
13
14     vec4 waterTextDiff = textureNoTile(texture_diffuse[1], diffCoord);
15     vec4 waterTextRefl = texture2D(texture_diffuse[2], reflectCoord);
16     float mixAmount = dist/5000;
17
18     result += CalcDirLight(dirLight, Normal, viewDir,waterText);
19     float alphaAmount = dist/2000;
20
21     FragColor = vec4(result,clamp(1.0,0.8,alphaAmount));
22 }
```

CAPÍTOL 10

Resultats

10.1 Captures de pantalla

Els resultats obtinguts els mostrem creant una escena amb 3 planetes als que apliquem diferents tècniques de soroll i efectes atmosfèrics propis. També definim 3 qualitats en funció del nombre de subdivisions amb el que s'ha construït l'esfera.

La qualitat baixa dels planetes equival a 8 subdivisions, mitjana a 9 subdivisions i alta a 10 subdivisions. a la Figura 10.1 podem observar els 3 planetes en qualitat baixa:

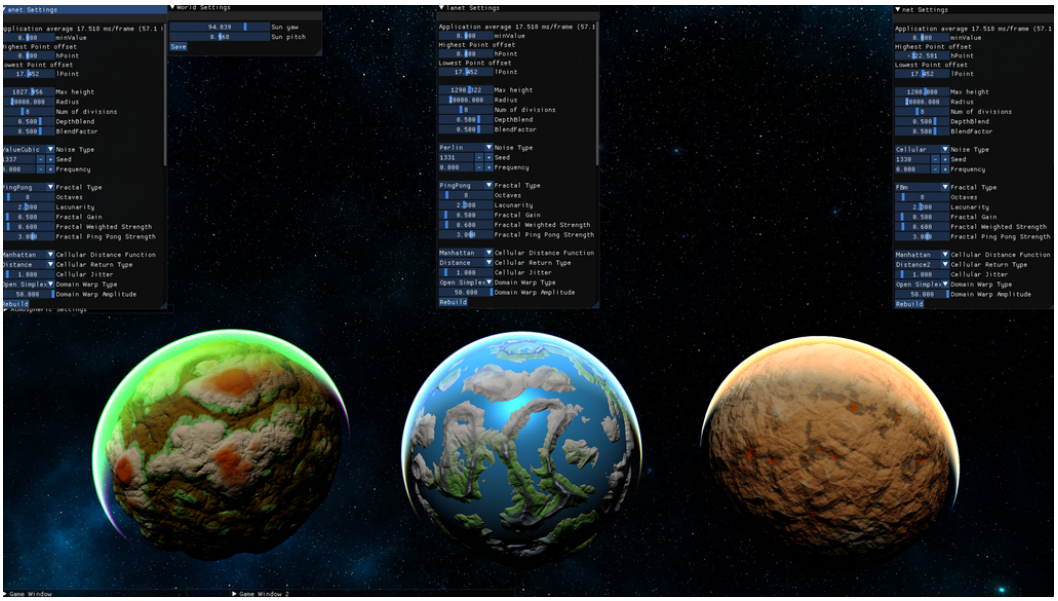


Figura 10.1: Tres planetes en qualitat baixa

a la Figura 10.2 podem observar els 3 planetes en qualitat alta:

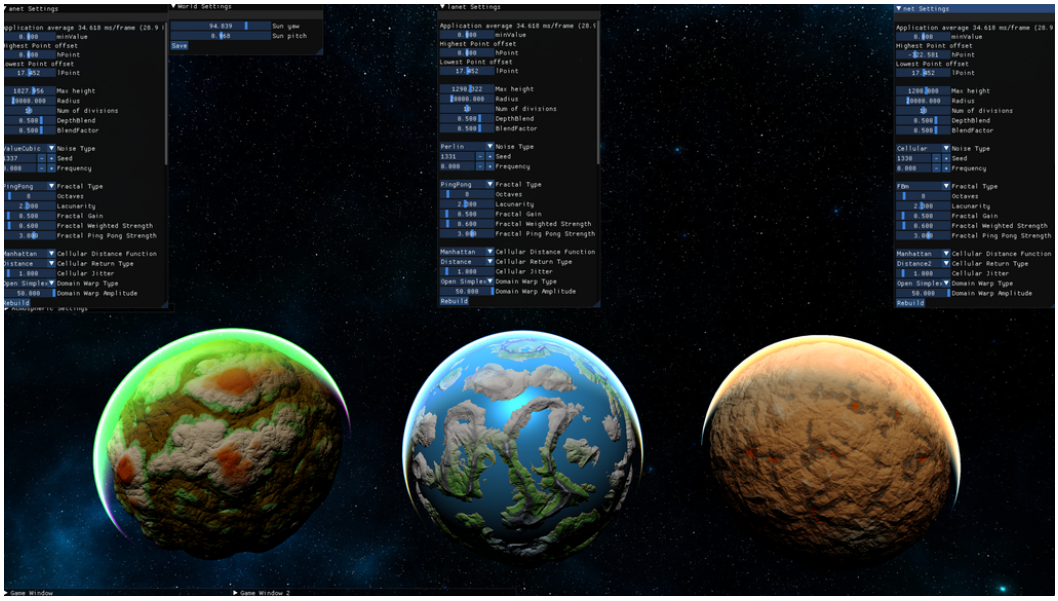


Figura 10.2: Tres planetes en qualitat alta

A continuació, mostrem els planetes des de la superfície. a la Figura 10.3 observem el planeta amb atmosfera verda (alienígena) en qualitat baixa:

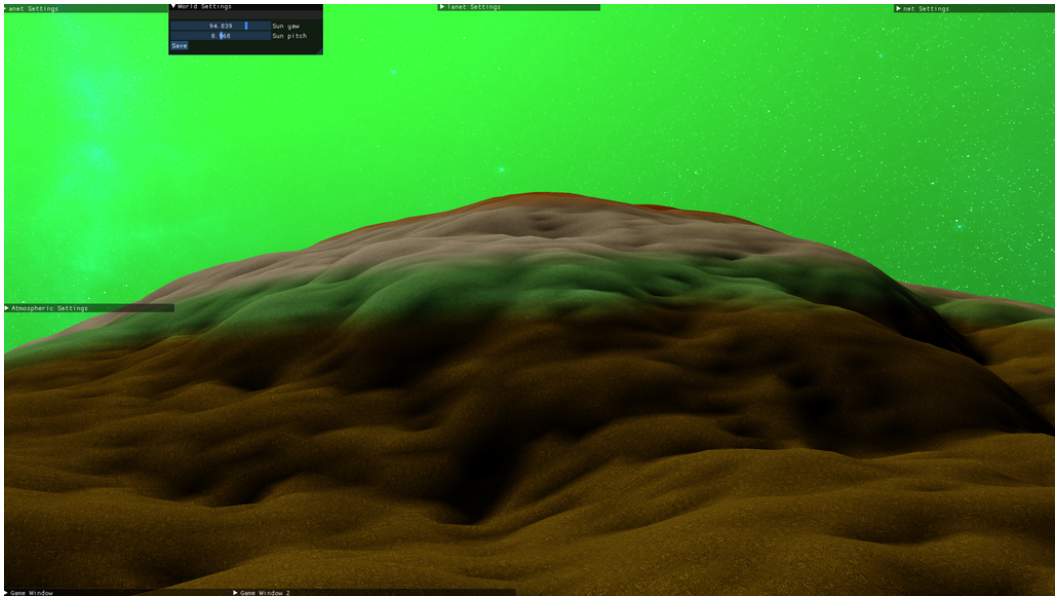


Figura 10.3: Planeta alienígena i qualitat baixa

A la 10.4 podem observar el planeta alienígena en qualitat mitjana:

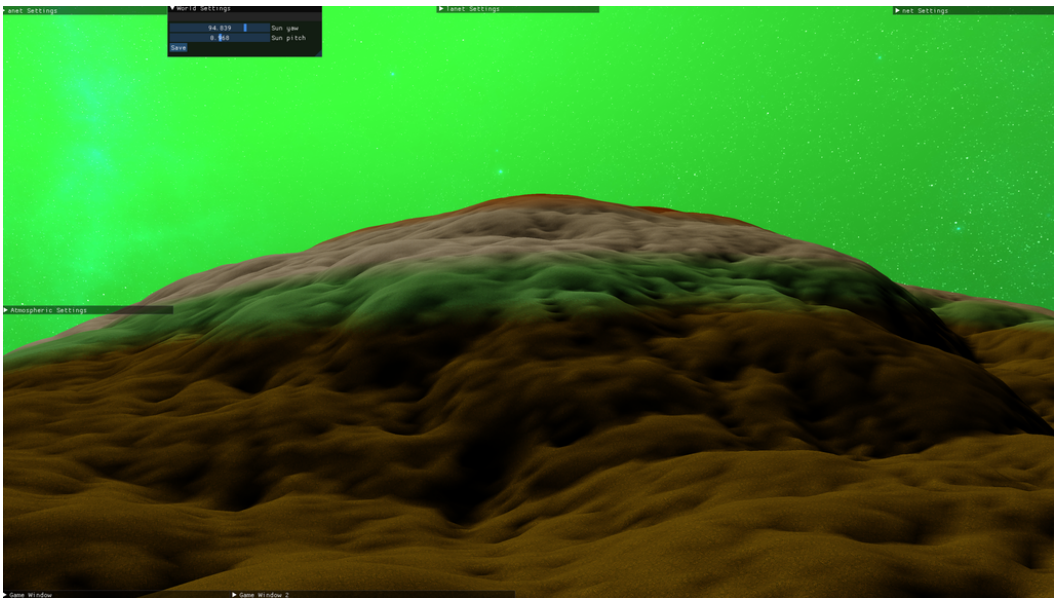


Figura 10.4: Planeta alienígena i qualitat mitjana

A la 10.5 podem observar el planeta alienígena en qualitat alta:

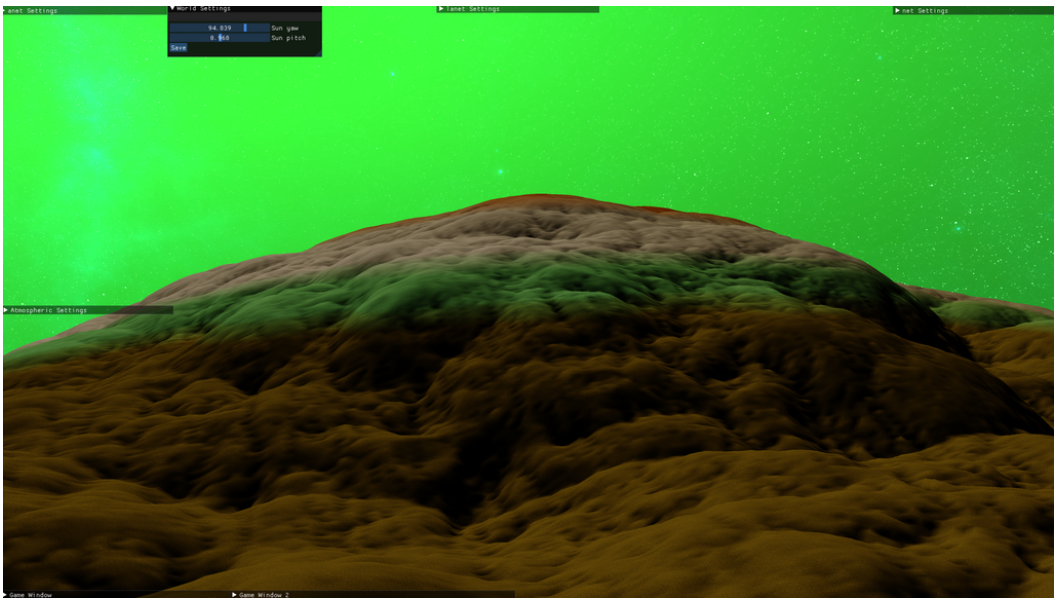


Figura 10.5: Planeta alienígena i qualitat alta

Seguim amb el planeta amb atmosfera blava (terrícola). a la Figura 10.6, podem observar aquest en qualitat baixa.

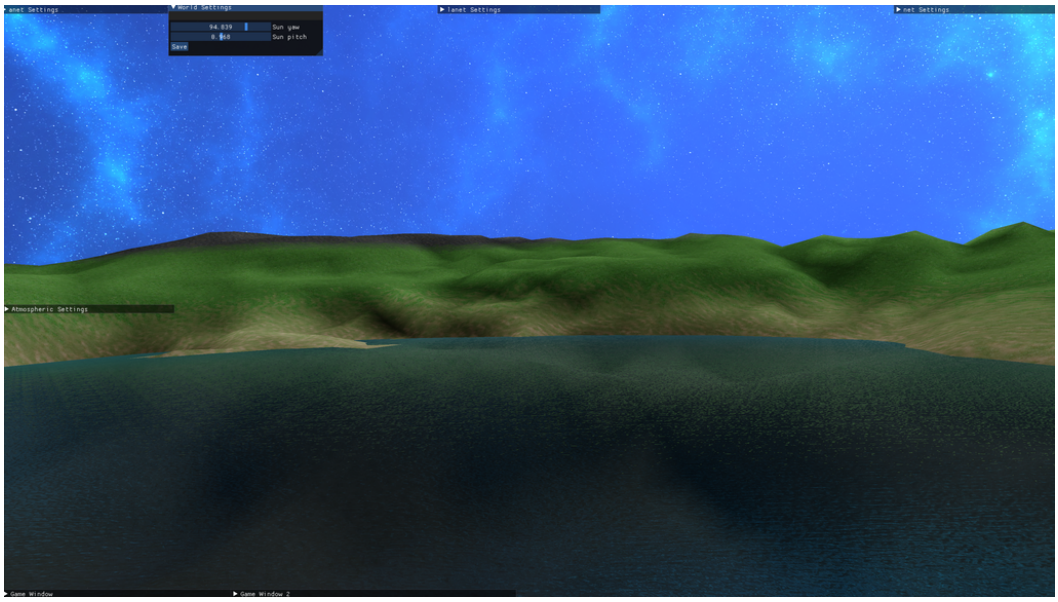


Figura 10.6: Planeta terrícola i qualitat baixa

A la Figura 10.7, podem observar el planeta terrícola en qualitat mitjana.

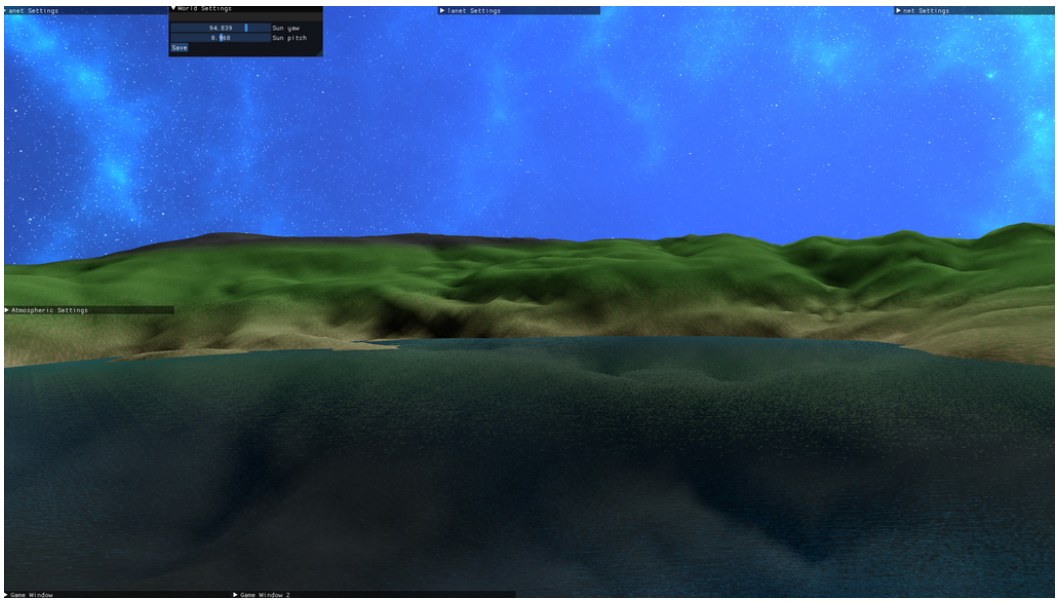


Figura 10.7: Planeta terrícola i qualitat mitjana

A la Figura 10.8, podem observar el planeta terrícola en qualitat alta.

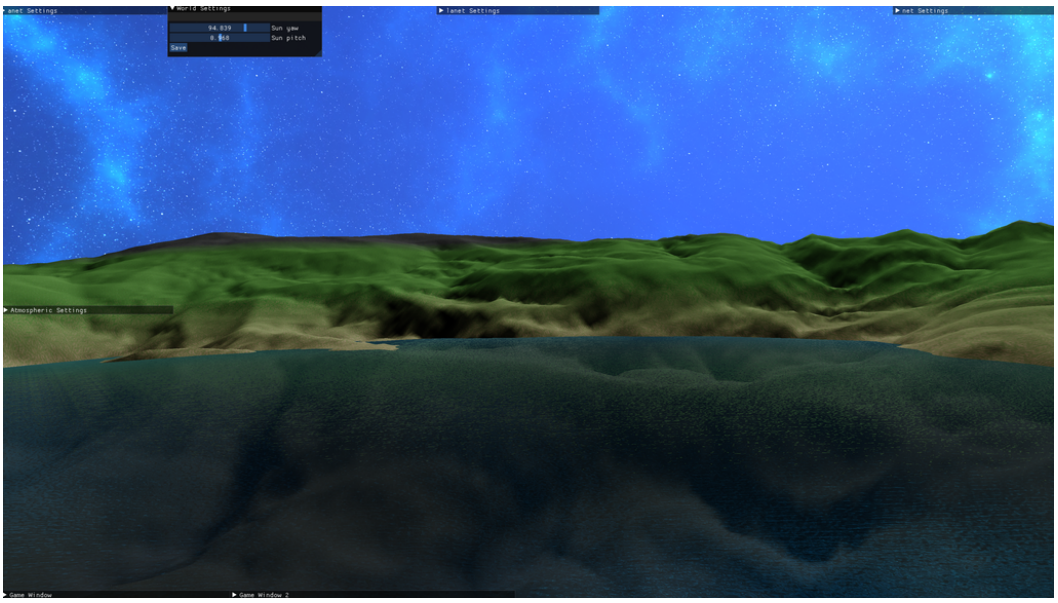


Figura 10.8: Planeta terrícola i qualitat alta

Continuem amb el planeta amb atmosfera vermella (marcià). a la Figura 10.9, podem observar aquest en qualitat baixa:

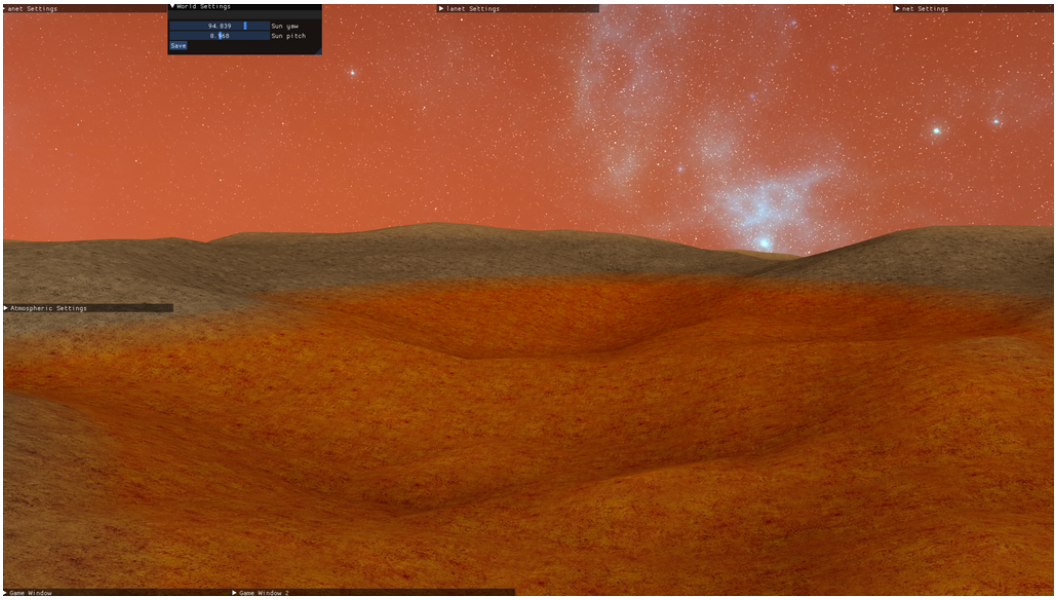


Figura 10.9: Planeta marcià i qualitat baixa

A la 10.10, podem observar el planeta marcià en qualitat mitjana.

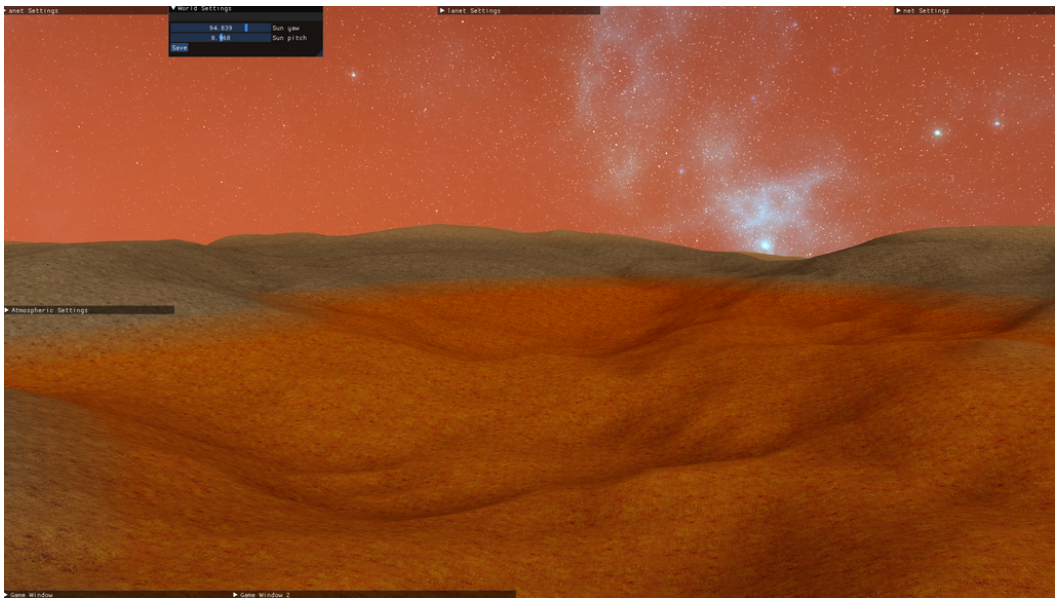


Figura 10.10: Planeta terrícola i qualitat mitjana

A la 10.11, podem observar el planeta marcià en qualitat alta.

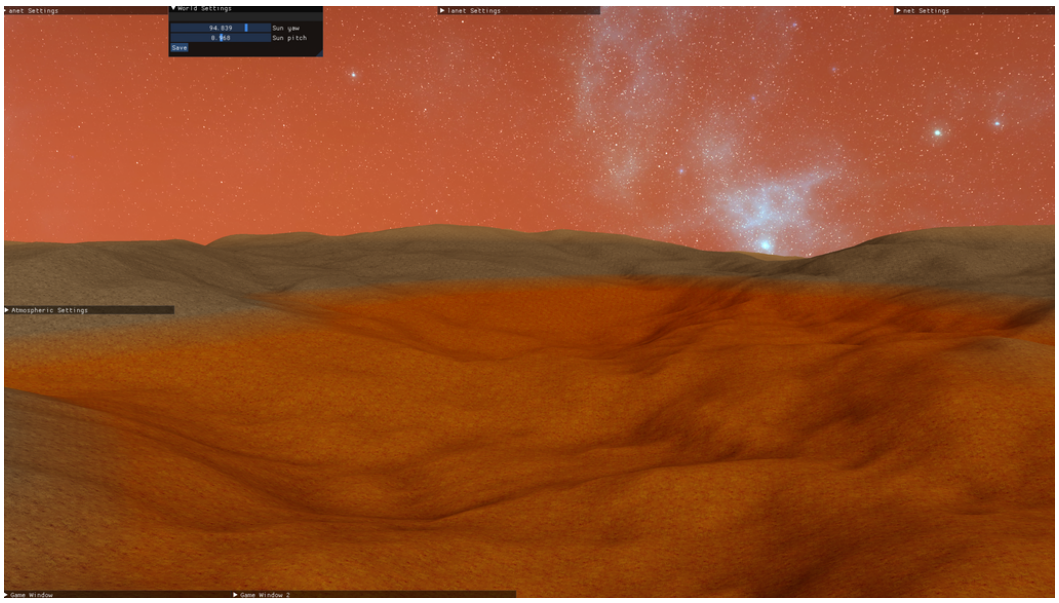


Figura 10.11: Planeta marcià i qualitat alta

A continuació mostrem el que podem modificant la funció de soroll. Partim del planeta marcià, a la Figura 10.12 configurem la funció de soroll amb **Cellular** i **CellularReturn Type CellValue**.

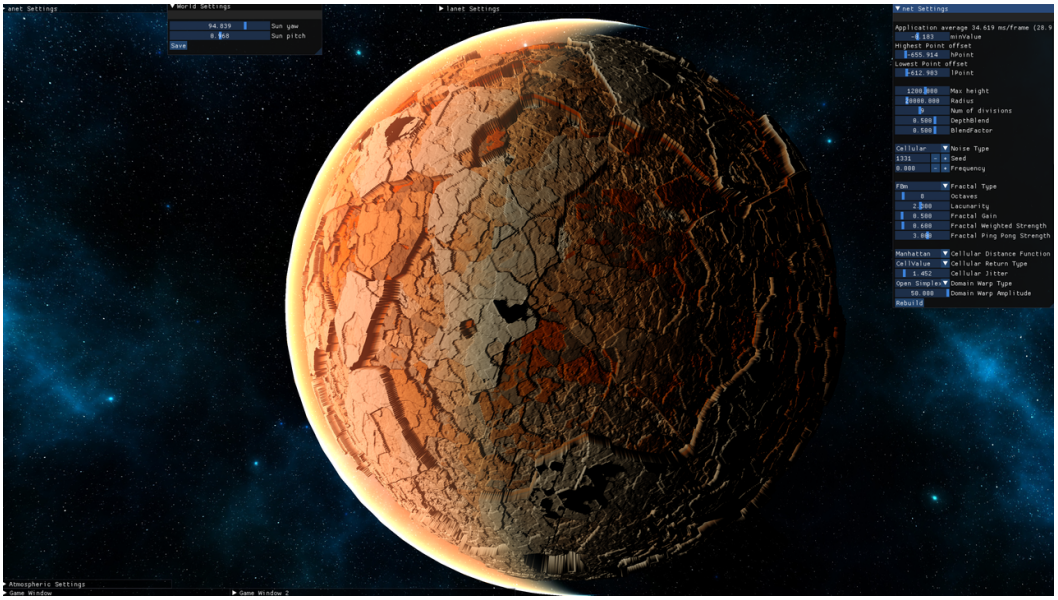


Figura 10.12: funció de soroll amb **Cellular** i **CellValue**

A la Figura 10.15 configurem la funció de soroll amb **Cellular** i **CellularReturnType Distance2Div**.

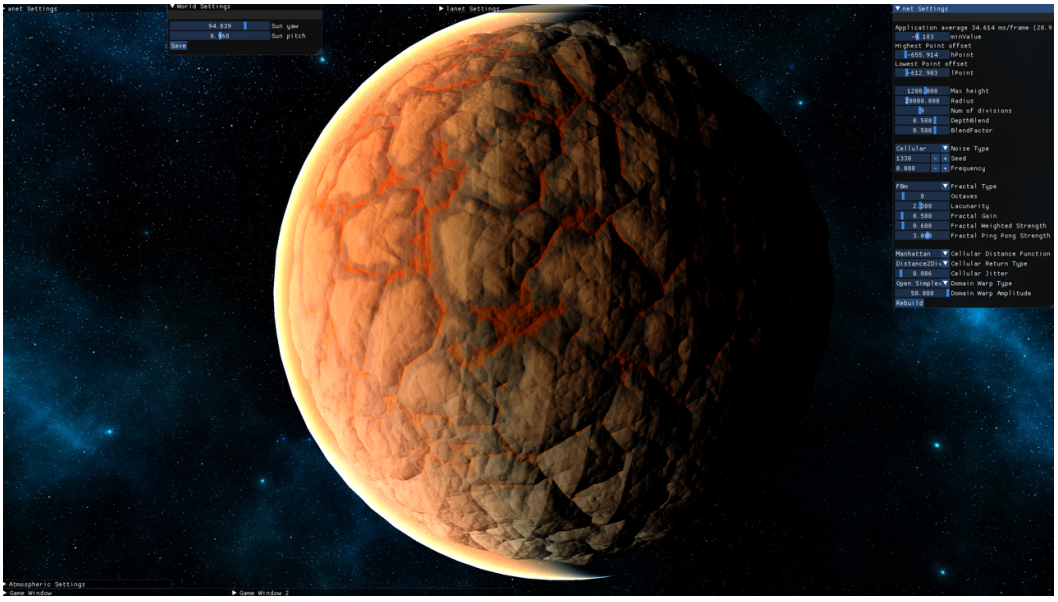


Figura 10.13: funció de soroll amb **Cellular** i **Distance2Div**

A la Figura 10.14 configurem la funció de soroll amb **Perlin** i **FractalType Ridged**.

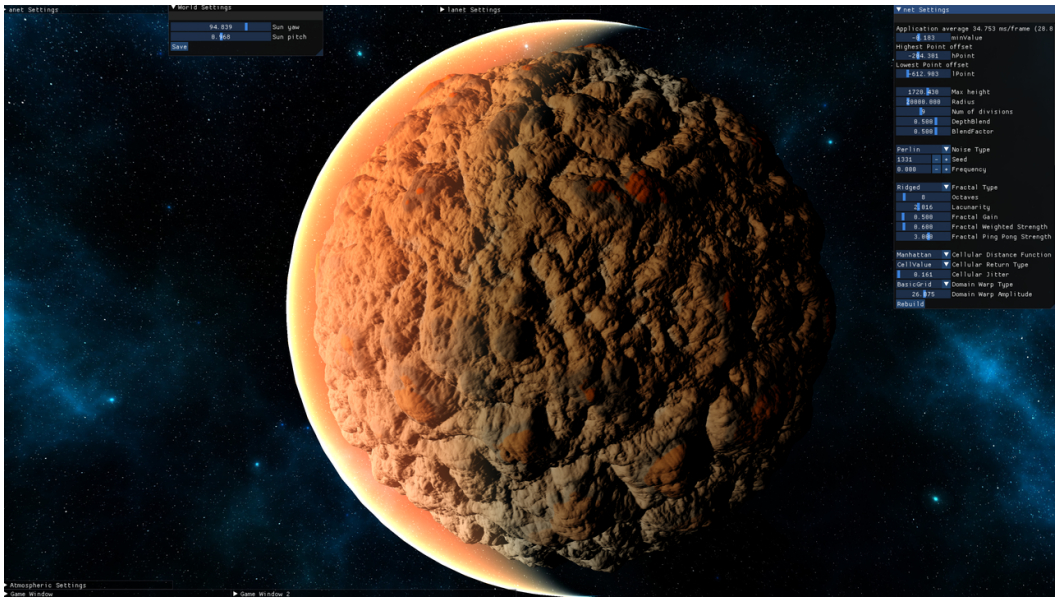


Figura 10.14: funció de soroll amb **Perlin** i **Ridged**

A la Figura 10.15 configurem la funció de soroll amb **Perlin** i **FractalType Ridged** amb 16 **octaves**:

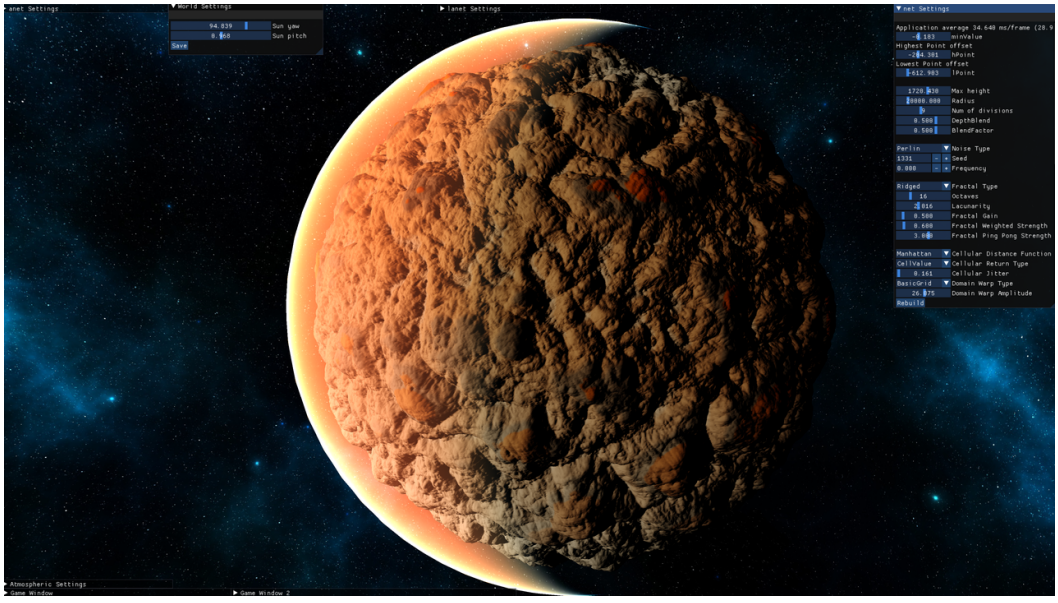


Figura 10.15: funció de soroll amb **Perlin**, **Ridged** i 16 **octaves**

A la Figura 10.16 configurem la funció de soroll amb **OpenSimplex** i **FractalType Ridged**:

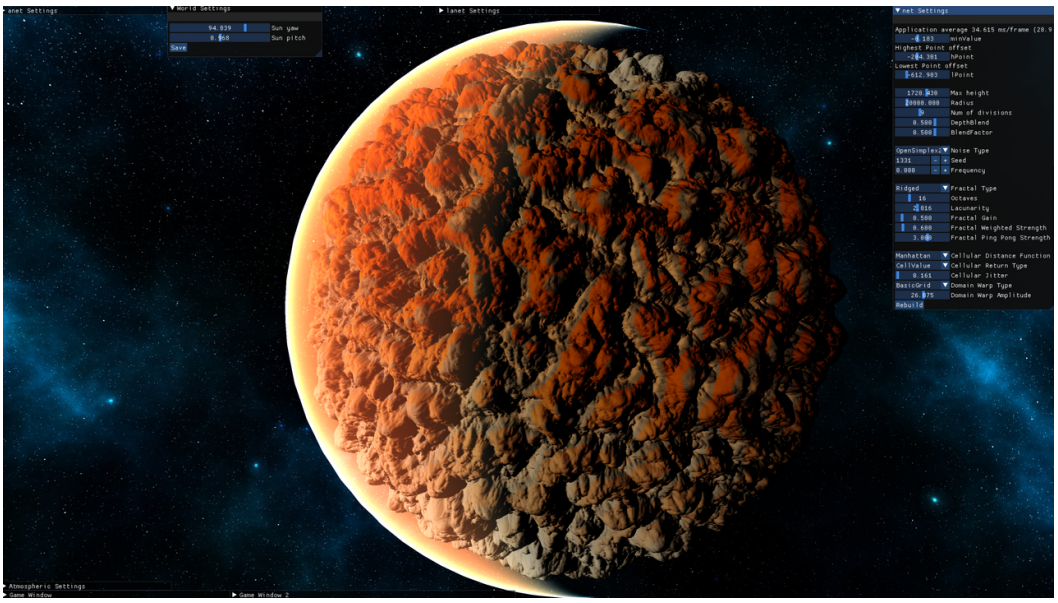


Figura 10.16: funció de soroll amb **OpenSimplex** i **Ridged**

A la Figura 10.17 configurem la funció de soroll amb **OpenSimplex** i **FractalType Ridged** amb freqüència reduïda:

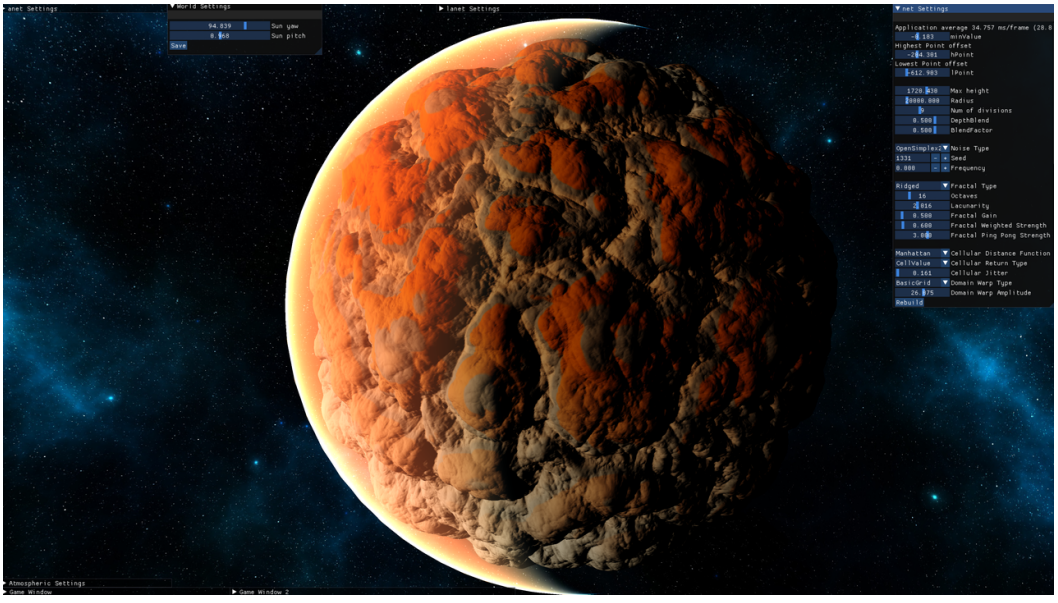


Figura 10.17: funció de soroll amb **OpenSimplex** i **Ridged** amb freqüència reduïda.

La configuració de l'atmosfera permet modificar la forma en què la llum interactua amb aquesta. Partim del planeta terrícola on a la Figura 10.18 em configurat l'atmosfera jugant amb els diferents atributs fins a aconseguir l'efecte visual desitjat.

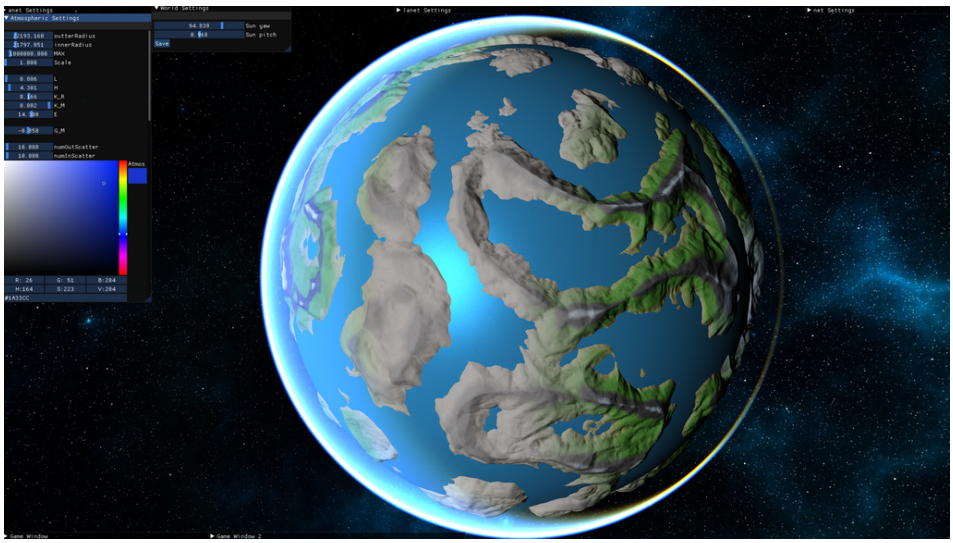


Figura 10.18: Atmosfera per defecte

Si incrementem l'atribut **H**, a la Figura 10.19 observem que la densitat de l'atmosfera es redueix.

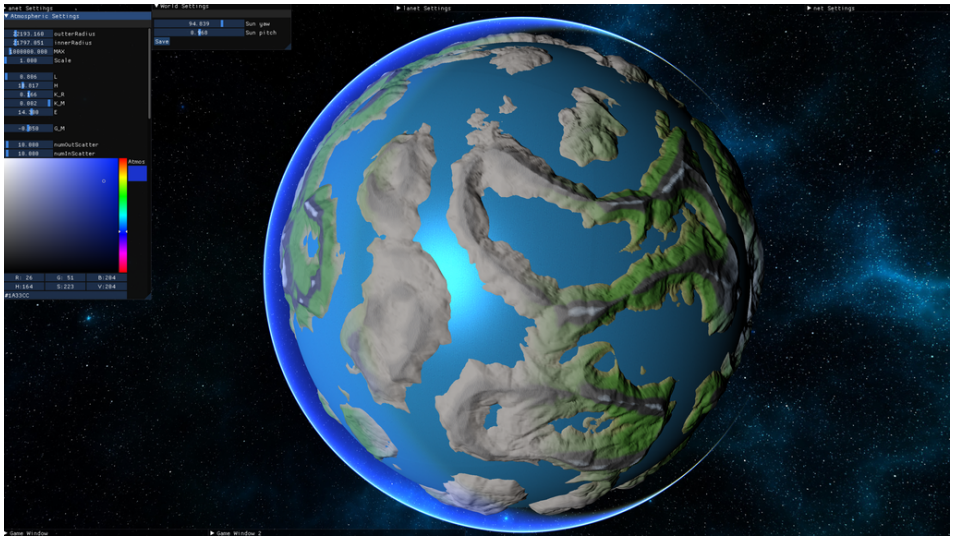


Figura 10.19: Atmosfera amb **H** incrementat

A la Figura 10.20 observem que incrementa la llum blava incrementant l'atribut **RayLeight**.

Figura 10.20: Atmosfera amb **RayLeigh** incrementat

A la Figura 10.21 observem que incrementa la lluminositat incrementant l'atribut **L**.

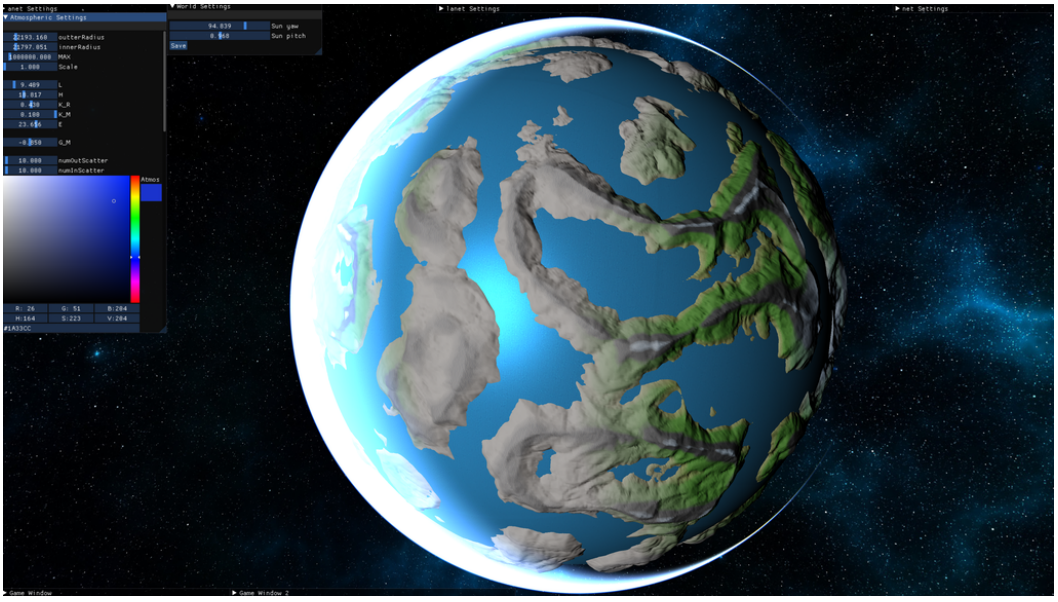


Figura 10.21: Atmosfera amb **L** incrementat

10.2 Demo

Per la demostració, el programa principal llegeix els fitxers a la carpeta **Scene** i mostra les rutes per pantalla, veure Figura 10.22:

```
Selecciona una escena:
[0]"../Scenes/Scene1.xml"
[1]"../Scenes/Scene2.xml"
[2]"../Scenes/Scene4.xml"
[3]"../Scenes/Scene3.xml"
```

Figura 10.22: Menú de la demostració

Això ens permet escollir entre 4 escenes diferents. L'escena número 0 consta d'un planeta i una lluna, veure figura 10.23:

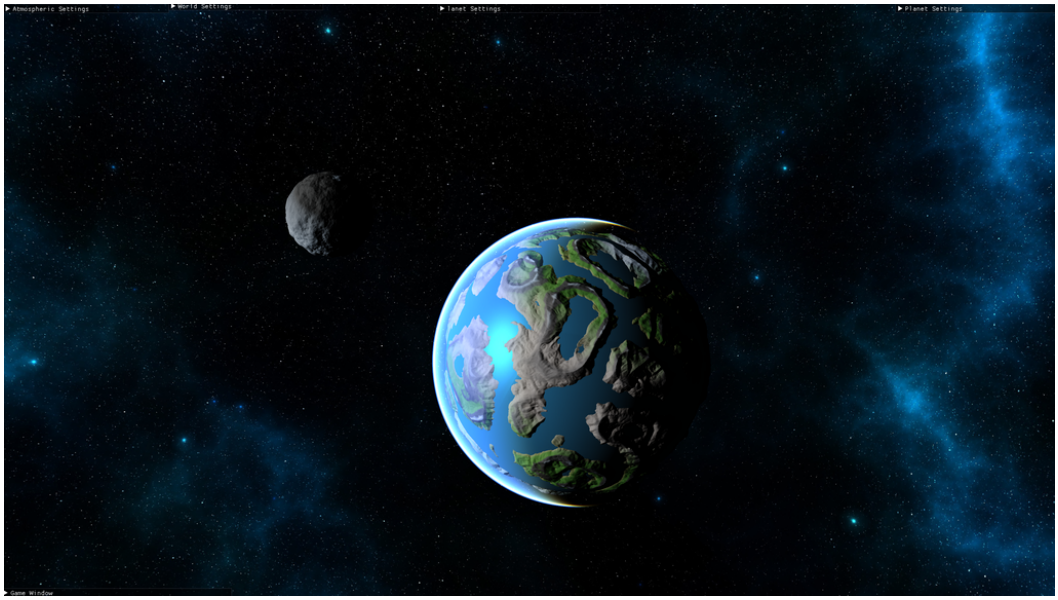


Figura 10.23: Escena 0

L'escena número 1 consta de dos planetes, un d'ells amb una lluna, veure Figura 10.24:

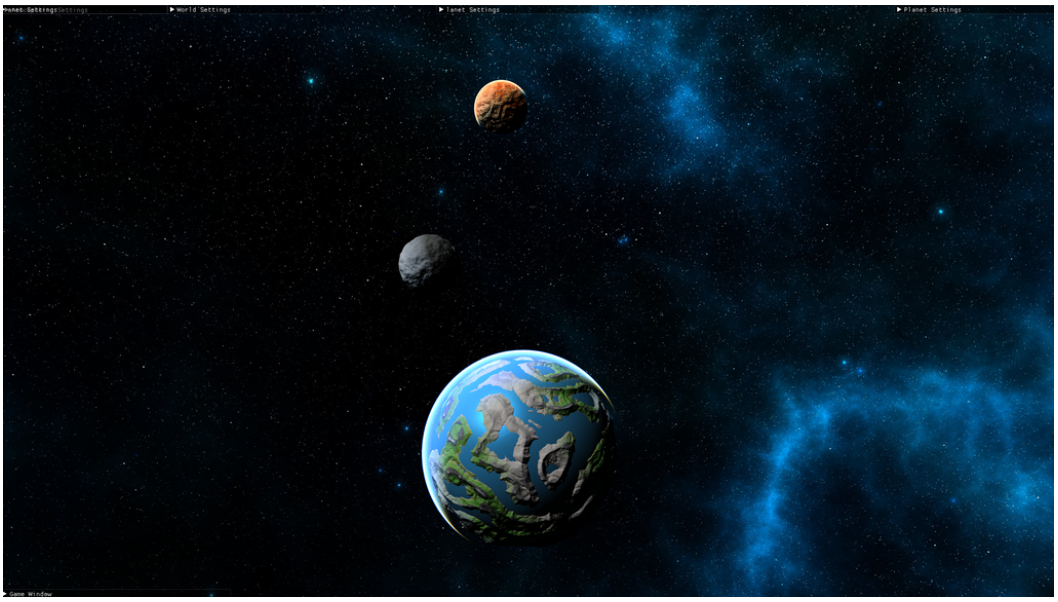


Figura 10.24: Escena 1

L'escena número 2 consta de tres planetes alineats, veure Figura 10.25:

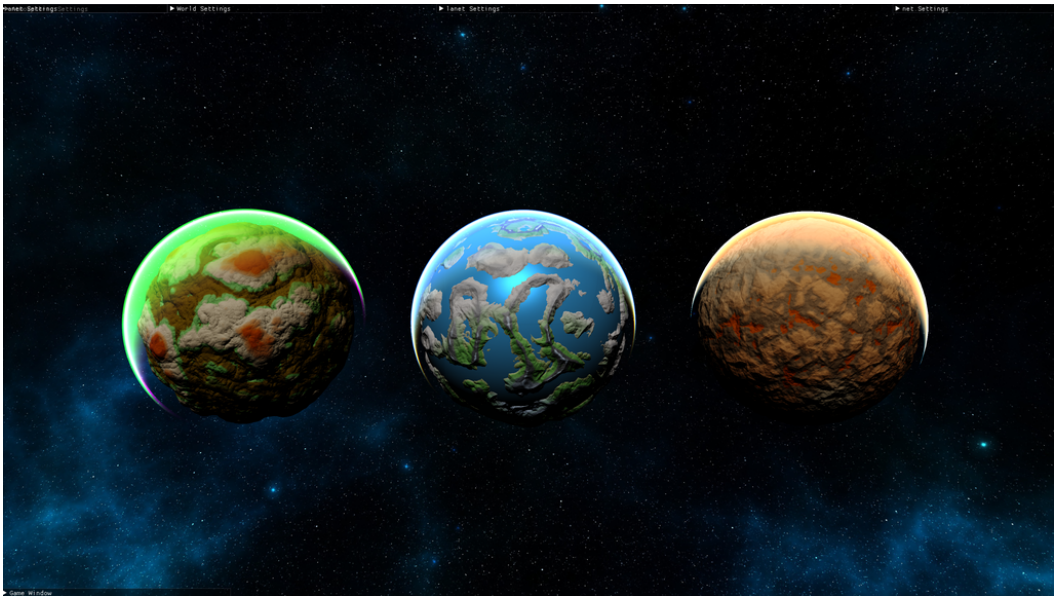


Figura 10.25: Escena 2

L'escena número 3 consta de tres planetes, un d'ells amb una lluna, veure Figura 10.26:

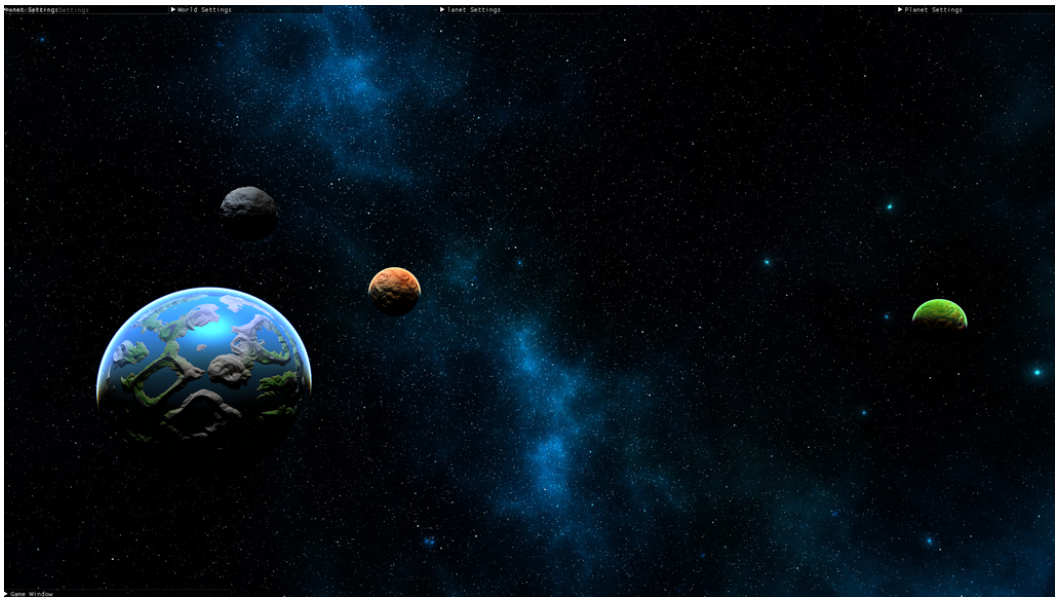


Figura 10.26: Escena 3

CAPÍTOL 11

Conclusions

Concloem aquest projecte amb tots els objectius plantejats complerts. L'estudi de com funciona **OpenGL** ens va proporcionar una base sòlida on hem pogut desenvolupar un entorn el més semblant possible a un *mapa* d'un videojoc.

L'estudi d'**OpenGL** també ens ha proporcionat un coneixement més elevat de les **API**'s gràfiques, fàcilment extrapolable a altres entorns més avançats com **Vulkan** o **Direct3D**, mantenint l'estructura del motor.

També hem aprés com treballar amb textures, algoritmes per interpolar-les, per millorar la qualitat i les limitacions i impacte que tenen sobre el rendiment. Gran part del temps invertit al projecte l'hem dedicat a millorar el rendiment dels algoritmes d'interpolació de textures.

La construcció de la geometria amb la funció de soroll ens ha permès crear un planeta el més natural possible, on podem observar tots els atributs característics de terreny real, com muntanyes, valls... La llibreria **FastNoise Lite** ens ha permès implementar això amb relativa facilitat.

Els efectes atmosfèrics i aquàtics afegeixen qualitat i realisme a l'escena final. La implementació d'efectes atmosfèrics va suposar elevar el coneixement que teníem sobre **shaders** i aprendre les limitacions que té **OpenGL** treballant amb materials transparents. Per altra banda, amb la implementació d'efectes aquàtics, vam millorar el coneixement que teníem sobre **FrameBuffers** i aprendre a crear efectes de distorsió.

Finalment, concloure amb la valoració bona dels coneixements adquirits desenvolupant aquest projecte, considero que ha estat el més important i que em servirà com a base per treballs futurs.

CAPÍTOL 12

Treball futur

El treball futur d'aquest projecte consisteix a millorar la qualitat i el rendiment de l'escena. Per fer això, existeixen tècniques conegudes:

- Dividir la geometria en **Octrees**, permeten crear blocs de terreny que podem modificar en temps d'execució i millorar la qualitat en funció de la distància.
- Tenir l'escena dividida en **Octrees** també ens permetria aplicar la tècnica de **Frustrum Culling**, on només enviem a **OpenGL** els objectes que es troben dins de la projecció de la càmera, evitant crides innecessàries al **Renderer**.
- La tècnica de **Frustrum Culling** també ens permetria afegir objectes sobre la geometria, com arbres, pedres, herba... que aportarien molt de realisme a l'escena.
- Afegir una distorsió real a la geometria de l'aigua creant onades, on es pogués especificar una direcció del vent i el moviment de la geometria és mogues acorde.
- Afegir més restriccions a les textures, per exemple, per inclinació del terreny per crear muntanyes més realistes.
- Afegir núvols als efectes atmosfèrics, que es moguin acorde a una direcció del vent específica amb efectes climàtics diferents.
- Crear un sistema de partícules interactiu per afegir efectes a l'escena.
- Millorar la interfície d'usuari, afegint una jerarquia de menús on l'usuari té l'habilitat d'afegir/eliminar objectes de l'escena.
- Millorar els efectes d'il·luminació, per exemple, aplicant tècniques més avançades com **RayTracing**.
- Millorar la interacció de l'usuari amb l'escena, on l'usuari tingui l'habilitat de seleccionar objectes amb tècniques com **RayCast**.

Bibliografia

[Ahn 021] Song Ho Ahn. *OpenGL Sphere*, (2018 - 2021). Available at http://www.songho.ca/opengl/gl_sphere.html. (Cited on pages 61 and 99.)

[Auburn] Auburn. *FastNoise Lite*. Available at https://github.com/Auburn/FastNoiseLite. (Cited on page 36.)

[Cornut] Omar Cornut. *Dear ImGUI*. Available at https://github.com/ocornut/imgui. (Cited on page 39.)

[de Vries 014] Joey de Vries. *LearnOpenGL*, (June 2014). Available at https://learnopengl.com/. (Cited on page 25.)

[Kafitz 2020] Jens Kafitz. *Noise terminology*, 2020. (Cited on page 36.)

[MboSoftWorks] MboSoftWorks. *Texture interpolation tutorial*. Available at https://www.mbsoftworks.sk/tutorials/opengl4/018-heightmap-pt3-multiple-layers/. (Not cited.)

[Mishkinis 2013] Andrey Mishkinis. *Advanced terrain texture splatting*, 2013. (Cited on page 109.)

[O'Neil 2005] Sean O'Neil. *Accurate Atmospheric Scattering*, 2005. (Cited on page 63.)

['provod' Avdeev 2018] Ivan 'provod' Avdeev. *Improving texture repetition*, 2018. (Not cited.)

[Quilez] Inigo Quilez. *Improving texture repetition*. Avaiable at https://www.iquilezles.org/www/articles/texturerepetition/texturerepetition.html. (Cited on page 107.)

[ThinMatrix 2015] ThinMatrix. *OpenGL Water Tutorial*, 2015. Available at https://youtu.be/HusvGeEDU_U?list=PLRIWtICgwaX23jiqVByUs0bqhnalNTNZh. (Cited on page 65.)