Universitat de Girona
Escola Politècnica Superior

Grau en Disseny i Desenvolupament de Videojocs

PROJECTE FINAL DE GRAU

Creació d'un *voxel engine* amb OpenGL i C++: *MinecraftGL*

Autor:
Rubén López Muñoz

Tutor:
Gustavo Ariel Patow

RESUM

Convocatòria:
Setembre 2023

Departament:
Informàtica, Matemàtica Aplicada i Estadística

Índex

1. Introducció

1.1. Introducció

Des de l'inici de la història dels videojocs, hem vist com les diverses maneres de representar un món en 3D han anat evolucionant amb el temps: des del mètode de *ray casting* del joc *Wolfenstein 3D* al 1992, que representava en 3D un escenari plantejat en 2D al codi, fins a jocs amb milers de polígons que semblen muntar un món aparentment infinit, com *No Man's Sky* (2016), o mapes gegants com el de *The Legend of Zelda: Tears of the Kingdom* (2023). Un dels mètodes més populars avui dia és l'ús de **vòxels** per representar el món.

L'aparició dels *voxels engines* va suposar una alternativa als polígons i models d'objectes convencionals. Aquests tipus de motors permeten la construcció de mons detallats i dinàmics a través dels vòxels, que no són res més que petits elements cúbics que, agrupats, poden arribar a representar qualsevol cosa. La creació i gestió d'un món dinàmic es torna una tasca molt més senzilla d'assolir si fem ús dels vòxels.

El joc més popular que fa ús de vòxels és *Minecraft* (2011), que va suposar la introducció al món dels videojocs a un públic increïblement extens i un èxit de vendes. Potser per aquest mateix motiu van començar a aparèixer còpies del videojoc, com *Manic Digger* (2014). Si bé aquest és un bon exemple de com *Minecraft* va afectar positivament al sector, la immensa majoria de les còpies que van sorgir mancaven algunes de les mecàniques clau, com col·locar i destruir blocs, la sensació d'infinitud del món o, fins i tot, un rendiment acceptable.

1.2. Motivacions

Tot i que actualment es poden fer modificacions al joc base, *Minecraft* no és un joc amb codi obert i les còpies o tampoc ho són, o el codi no és fàcil d'entendre ràpidament per una manca de documentació.

És per aquesta falta de documentació d'un motor gràfic especialitzat en vòxels que en aquest TFG ens centrarem en documentar degudament la creació i implementació d'un *voxel engine* molt semblant a *Minecraft*. Desenvolupar i fer que el motor sigui fàcilment accessible, adaptable i, sobretot, de codi lliure, permetrà a altres persones modificar-lo lliurement sense haver de preocupar-se de la part més complexa de renderització i optimització, apropant aquest món a persones que s'estan iniciant.

Personalment, crec que crear el meu propi motor gràfic, i més fer-ho semblant a un joc tan popular i que m'agrada tant, pot ser l'oportunitat perfecta per demostrar els coneixements que he adquirit al grau.

1.3. Propòsit i objectius del projecte

L'objectiu principal del projecte és desenvolupar un motor gràfic que s'apropi a *Minecraft*, és a dir, que presenti un món en 3D fent ús de vòxels i que permeti al jugador navegar i construir lliurement. A més, ha de permetre a qualsevol usuari interessat la possibilitat de modificar el codi lliurement.

Més concretament, el motor haurà de permetre dibuixar vòxels a l'espai en 3D i *sprites* en 2D pel HUD (*Head-up Display*) i així donar informació al jugador. El HUD no haurà d'interactuar de qualsevol manera amb el món en 3D i viceversa. Aquest món en 3D serà un món "infinit" creat de manera procedural i donarà la llibertat a l'usuari de poder modificar determinats valors per poder modificar com es crearà. El jugador podrà afegir o destruir vòxels en temps real. Tot això s'ha de poder fer mantenint un rendiment bo, sense baixades sobtades de frames.

Tenint aquests propòsits en ment, hem decidit utilitzar la API gràfica OpenGL per C++ per renderitzar el món i el HUD. Per tant, un altre dels objectius del projecte és el d'aprendre a fer servir la llibreria i tècniques d'optimització per un *voxel engine*. Com estem fent servir OpenGL, també haurem d'aprendre a utilitzar altres llibreries com GLFW, GLM o *freetype*, així com GLSL, que és el llenguatge de programació de *shaders*.

1.4. Quadre d'autoavaluació

Ja que es tracta d'un projecte enfocat a la creació d'un motor gràfic, el quadre queda de la següent manera:

Estètica	5%
Narrativa	0%
Mecàniques	15%
Tecnologia	80%

El pes d'estètica és referent a la creació dels *shaders* que veurem més endavant i les decisions estètiques d'apartats com el HUD, la generació de món i la decoració (com arbres, flors o arbustos). Dintre de les mecàniques trobarem la part referent al jugador: que es pugui moure, interactuar amb un inventari, amb el món, etc. I tractant-se d'un motor, no he vist cabuda per una narrativa. Clarament el que més pes té és la tecnologia, ja que hem de poder visualitzar el món sencer, el HUD, poder renderitzar text, etc.

2. Estudi de viabilitat

Abans de començar el projecte, cal analitzar si és viable i assegurar-nos que es pot implementar sense problemes. Hem d'estudiar altres projectes similars, definir el perfil del jugador i tenir en compte les eines necessàries, com els recursos humans, el programari que s'ha d'utilitzar... En el nostre cas, ja tenim la infraestructura per desenvolupar el projecte eficaçment, amb costos d'infraestructura mínims.

2.1. Recursos humans

Si volguéssim realitzar el projecte en tota la seva plenitud, amb apartat gràfic inclòs, hauríem de tenir en compte que necessitem diferents perfils de treballador, i que el sou d'aquests difereix segons la seva categoria:

- **Dissenyador**: 20€/h. S'encarrega de conceptualitzar el projecte: posar en clar quins són els objectius a assolir, les mecàniques...
- **Artista**: 15€/h. S'encarrega de crear les imatges i textures necessàries.
- **Programador**: 15€/h. Responsable de la programació del projecte, assegurant el correcte desenvolupament de tot el que ha proposat el dissenyador.

Si haguéssim de relacionar cada perfil amb una tasca, tenint en compte les hores que haurien de fer cadascun per poder completar el projecte, obtindríem el següent quadre de costos:

Tasca	Perfil	Hores	Cost
Investigació	Dissenyador	60	1200€
Disseny d'algorismes	Dissenyador	80	1600€
Documentació	Dissenyador	80	1600€
Implementació	Programador	300	4500€
Proves	Programador	200	3000€
Disseny de gràfics	Artista	100	1500€
Disseny del HUD	Artista	40	600€

El total és de 860 hores, amb un cost total de 14000€. En el nostre cas, una sola persona és l'encarregada de suplir tots els rols.

2.2. Recursos tecnològics

Hem de tenir en compte també la maquinària emprada per poder realitzar el projecte, així com el *software* que haurem de utilitzar per desenvolupar-lo correctament.

2.2.1. *Hardware*

El *hardware* que s'ha fet servir és un ordinador portàtil que ja es trobava a la nostra disposició amb les següents especificacions:

- **Sistema Operatiu:** Windows 11
- **CPU**: AMD Ryzen 7 5800H
- **RAM**: 16GB
- **GPU**: NVIDIA GeForce RTX 3060 6GB

Per suposat, el projecte es pot realitzar en un entorn amb especificacions menors i en apartats posteriors demostrarem que és possible córrer el motor en màquines inferiors. El cost de l'ordinador en cas de no disposar-lo és aproximadament de 1000€.

2.2.2. *Software*

A continuació disposem de tot el *software* que hem utilitzat per portar a terme el projecte:

- **Visual Studio**: un IDE per poder desenvolupar en C++. Ens decidim per aquest IDE en comptes de Visual Studio Code per la flexibilitat a l'hora de compilar i gestionar el projecte i tots els arxius, classes...
- **GIMP**: es tracta d'un programa de tractament d'imatges que permet crear gràfics fàcilment.
- **GitHub**: un sistema de control de versions molt necessari pel volum de treball que suposa el projecte, a més de proporcionar una eina per poder veure tasques que falten per fer.

Tot el programari esmentat és gratuït, així que el cost d'aquest apartat és nul. Les llibreries emprades també són gratuïtes.

2.3. Estudi de mercat

A continuació farem una llista de productes que s'assemblen al que volem aconseguir amb el nostre projecte:

2.3.1. *Minecraft*

Òbviament el primer que hem d'esmentar és *Minecraft*. És el joc més popular i complet fet amb un *voxel engine*. Permet diferents modes de joc, com supervivència, on el jugador ha de recollir recursos per poder afrontar el dia a dia al joc, tractant de sobreviure, o el mode creatiu, on el jugador té llibertat per construir el que vulgui sense restriccions. Es tracta d'un *sandbox*, o caixa de sorra: el joc dona llibertat per construir en un món aparentment infinit (veure Figura 2.1).

Si bé és veritat que es poden fer modificacions al joc amb programes externs, no és un joc de codi obert, i per tant no és tan fàcilment adaptable. El nostre projecte cobriria la necessitat d'un joc semblant però més accessible a nivell de programació.

Figura 2.1: el món de Minecraft

2.3.2. *Manic Digger*

Un joc basat en vòxels en 3D molt semblant a *Minecraft*. En comparació, manca un món dinàmic que es va construint segons el jugador es va movent. A la pròpia pàgina de *GitHub* mencionen que el món es immens (9984x9984x128), però no infinit. Aquesta mancança la podem suplir amb el nostre projecte incloent la generació dinàmica de terreny. Tot i que és un joc en codi obert, no hi ha gairebé documentació. A la Figura 2.2 podem veure la seva similitud amb *Minecraft*:

Figura 2.2: captura de pantalla del joc Manic Digger

En conclusió, ens podem fer un lloc en el mercat si aconseguim balancejar les mancances que hem esmentat.

3. Planificació

Veient la mida del projecte i la quantitat de treball que hi ha per una persona, s'ha dividit tot aquest en diferents tasques que s'han anat assolint al llarg dels mesos. Tot i que no es segueixin per aquest ordre, o si al final s'ha fet una tasca abans que una altra, la llista a continuació mostra com s'ha repartit la feina per tal de fer un projecte viable:

- **Investigació**: investigar com funciona un *voxel engine*, fer "treball de camp" obrint Minecraft i investigant com podem adaptar les mecàniques al nostre projecte. És una tasca constant: sempre hem d'estar trobant millors optimitzacions i maneres d'implementar-les per poder decidir després què és el que volem.

- **Practicar**: aprendre a utilitzar OpenGL i la resta de llibreries. Abans de començar a desenvolupar el projecte, és necessari saber el que estem fent. Durant aquesta tasca hem d'aprendre a obrir una finestra i poder visualitzar un polígon al que li podem canviar el color, transformar-lo, etc.

- **Inicialització**: començar el projecte. Inicialitzem un programa senzill de shaders, ens assegurem que s'obri la finestra correctament, etc. Quan s'acabi aquesta tasca, ens fixem l'objectiu de poder visualitzar en una finestra un conjunt de vòxels (chunk) i poder orbitar al voltant d'aquest. Això significa que necessitem de la implementació d'una càmera i de les primitives necessàries per poder visualitzar els vòxels.

- **Conjunt de chunks**: o el que és el mateix, generar i trobar la manera de gestionar un món. Al final d'aquesta tasca no fa falta que s'hagi aconseguit un món "infinit", però sí que el jugador pugui moure's lliurement per un terreny finit. L'usuari haurà de poder determinar el tamany d'aquest terreny, així com el de cada chunk. A més, han d'estar ben optimitzats.

- **Eliminar i col·locar vòxels**: hem de poder destruir i construir al món per tal de donar al jugador un grau de llibertat.

- **Texturització de vòxels**: poder distingir els vòxels els uns dels altres visualment mitjançant una textura.

- **Il·luminació**: alguns vòxels emeten il·luminació pròpia. Farem una simulació de llum.

- **Creació de la base de dades**: si tenim uns quants vòxels no fa falta, però la idea és poder expandir el projecte amb tots els vòxels que vulguem. Hem de poder mantenir d'alguna manera una base de dades on apareguin tots els tipus de vòxels que hi ha.

- **Món "infinit"**: per tal de desmarcar-nos d'altres projectes, hem de deixar que el món es vagi construint dinàmicament i presentar al jugador un món gairebé interminable.

- **Implementació de HUD:** d'aquesta manera podrem permetre al jugador escollir el tipus de vòxel que vol col·locar. Distingim entre diverses tasques:
    - *Implementació d'inventari:* deixarem que el jugador tingui una col·lecció de vòxels per poder construir.
    - *Implementació de text:* útil per saber el vòxel que ha escollit o altre informació.
    - *Debug:* una mica de text accessible a través d'una tecla per tal d'obtenir informació com la posició del jugador o el vòxel que està observant.

- **Generació de món**: distingim entre diverses tasques:
    - *Generació de vegetació*: flors, arbustos, etc.
    - *Generació d'arbres*: estructures amb fusta i fulles.
    - *Generació de terreny*: aigua, muntanyes, etc.

- ***Frustum culling***: una manera d'obtenir un increment de rendiment realment significatiu. Veurem de què tracta a l'Apartat 5.8.17.

- **Col·lisions del jugador:** per si el jugador no vol anar tan lliurement pel món, hem d'activar gravetat i poder col·lisionar amb vòxels.

- **Decoracions**: núvols, detalls de shaders que donen molta més vida al motor, com estar sota l'aigua, boira, etc.

- **Documentació**: la part de documentació del projecte.

En aquesta llista hem descrit els objectius principals que volem assolir per una demostració d'una part del que seria el projecte sencer, però no difereix gaire de com començaria un *voxel engine* qualsevol.

Hem creat un diagrama de Gantt que reflexa l'evolució del projecte al llarg del temps:

Tasca	Octubre	Novembre	Desembre	Gener	Febrer	Març	Abril	Maig	Juny	Juliol	Agost
Investigació											
Pràctica											
Inici											
Conjunt de chunks											
Modificar terreny											
Texturització											
Il·luminació											
Base de dades											
Món infinit											
Generació vegetació											
Generació arbres											
Generació de terreny											
Text											
Inventari											
Informació *debug*											
Frustum culling											
Col·lisions											
Decoracions											
Documentació											

4. Marc de treball

4.1. Llibreries

A continuació veurem quines llibreries hem utilitzat en aquest projecte en més detall, explicarem per què les fem servir i com.

4.1.1. OpenGL

OpenGL és una API per crear gràfics en 2D i 3D. Proporciona un conjunt de funcions i comandes que permeten als desenvolupadors controlar i renderitzar gràfics en temps real. És multiplataforma i compatible amb una varietat de sistemes operatius i dispositius gràfics, la qual cosa la fa molt versàtil. Hem escollit OpenGL perquè a la carrera ja s'ha treballat una mica amb la seva API germana, WebGL, a més de estar molt ben documentada. També proporciona la capacitat de programar shaders, així com d'altres funcions que ens seran útils més endavant. Farem servir la seva versió per C++.

4.1.2. GLFW

GLFW és una llibreria que treballa conjuntament amb OpenGL, ja que tot i que aquesta darrera és l'encarregada de dibuixar i renderitzar models, manca una característica clau: no pot crear finestres. Per aquest motiu, GLFW s'utilitza per generar finestres i controlar l'entrada de l'usuari. El farem servir per controlar al jugador: moure la càmera, detectar el moviment i els clics del ratolí, etc.

4.1.3. GLAD

GLAD és un generador de funcions d'OpenGL. A causa de la naturalesa de l'API gràfica, si no tinguéssim GLAD, hauríem de carregar i administrar nosaltres mateixos les funcions d'OpenGL. GLAD ens treu aquest treball de sobre.

4.1.4. GLM

GLM és una llibreria que ens facilita funcions matemàtiques enfocat a OpenGL i altres llibreries que utilitzin GLSL (el llenguatge de programació de shaders). Proporciona classes com *vec3* (vector de 3 elements) i funcions per aquestes, com normalitzar, fer producte vectorial, etc. És molt útil també per fer operacions amb matrius, que més endavant veurem que són essencials per la càmera del jugador.

4.1.5. FreeType

FreeType és una llibreria que ens allibera de la càrrega de treball d'haver de gestionar fonts de lletra. OpenGL no proporciona cap mena de funcions per treballar amb text, així que hauríem de ser nosaltres els que carreguéssim una font, analitzar cada caràcter, etc. FreeType ens ajuda proporcionant-nos una classe que s'encarrega d'aquesta tediosa tasca. Només li hem de donar una font que pugui reconèixer i la llibreria farà la resta.

4.1.6. nlohmann/json

Com ja hem expressat al capítol 3, una de les tasques és veure com podem gestionar una base de dades. Veurem que aquesta llibreria, que permet fer ús d'arxius en format JSON en C++, fa que aquesta feina sigui molt més senzilla.

4.1.7. FastNoise

FastNoise és una llibreria que permet generar mapes i gradients de soroll. Podem generar diferents tipus de soroll: Perlin, Simplex, Celular, etc. Aquesta llibreria és important perquè ens dona accés a nivells de soroll amb moviment brownià. A l'apartat 4.2.7 explicarem amb més detall què és el soroll i per què el farem servir al projecte.

4.1.8. stb_image

L'arxiu stb_image.h és part d'una llibreria multifuncional més gran anomenada stb. Amb stb_image podem carregar imatges PNG, JPG, etc. Gràcies a això, podrem carregar textures i visualitzar millor vòxels, a més de poder fer els nostres propis sprites.

4.2. Conceptes previs

Abans d'endinsar-nos a explicar com s'ha implementat el projecte i les parts d'aquest, hem de tenir clar alguns conceptes claus que s'esmentaran al llarg del document.

4.2.2. Vòxel

Ja hem explicat molt breument anteriorment què és un vòxel, però en aquesta secció veurem amb més detall de què es tracta. En el nostre cas, definim un vòxel com la unitat mínima modificable al nostre món. Per fer-ho encara més senzill: un vòxel és un polígon de 6 cares o, per fer la correlació amb una figura coneguda, un **cub**. A la figura 4.1 podem veure un exemple de vòxel i com visualment s'assembla a un cub.

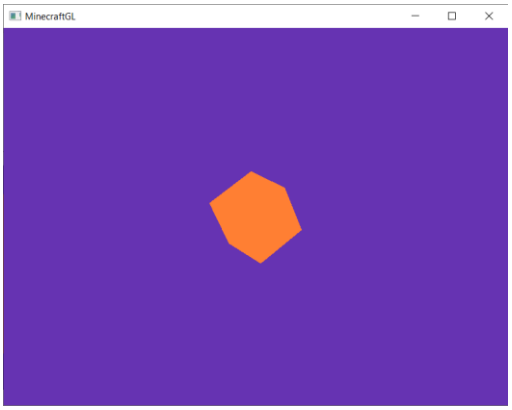


Figura 4.1: exemple de vòxel taronja en un fons violeta

Tot i que vòxel és un sinònim de cub per les seves similituds, al nostre projecte els vòxels no es construeixen com un simple cub per motius d'optimització i texturització. En comptes de fer un cub sencer, el motor separarà cada cara i la tractarà individualment. Això ens permetrà posar una textura diferent a cada cara, ja que potser un vòxel no es texturitza de la mateixa forma per tots els costats, tot i que el motiu principal és millorar el rendiment. Veurem quina és aquesta millora a l'apartat següent.

Veurem que el motor distingirà entre diferents tipus de vòxels depenent de la situació, com que siguin transparents o que representin vegetació. Tindrem vòxels que representin terra, gespa, pedra, aire, etc.

4.2.3. Chunk

Tenir un vòxel està bé, però un món no es construeix només amb un de sol. Necessitem moltíssims més cubs si volem que el jugador pensi que està davant d'un món infinit. La qüestió és que, si tenim milions de cubs formant un món, està clar que necessitaríem una memòria RAM gairebé infinita per poder gestionar-los tots i guardar la informació de cadascun. Òbviament això és impossible, així que hem de trobar una manera més eficaç de gestionar els vòxels, tot mantenint la mentalitat de que hem de poder accedir fàcilment a qualsevol vòxel que es trobi al món.

La solució és agrupar els vòxels en grups i visualitzar només els grups que vulguem. És molt més senzill tenir 16 grups de 30 vòxels cadascun que només un de 480. Cadascun d'aquests grups s'anomena chunk.

El món es composarà de chunks que s'aniran generant dinàmicament. Un chunk tindrà 4 adjacents: un a l'esquerra, un altre a la dreta, enfront i al darrera. Si pensem en aquesta representació, el món és un conjunt en 2 dimensions de chunks (n chunks per m chunks), que són grups en 3 dimensions de vòxels (x vòxels per y vòxels per z vòxels). A la Figura 4.2 hi ha representat un chunk de 2*4*2 vòxels.

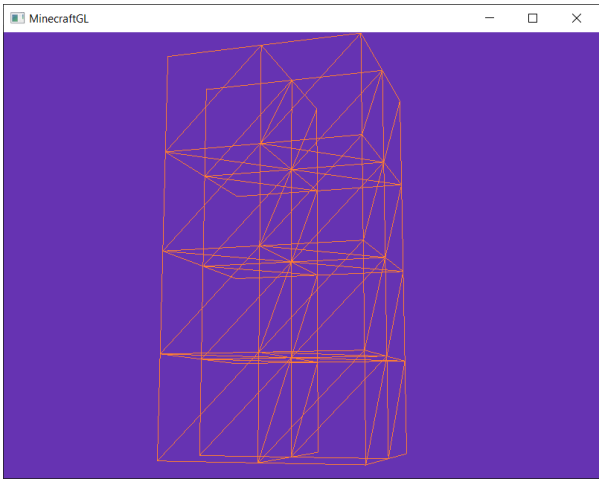


Figura 4.2: un petit chunk renderitzat en wireframe

Abans hem explicat que els vòxels no els tractarem com un cub sencer pels motius abans esmentats. Observem la Figura 4.3:

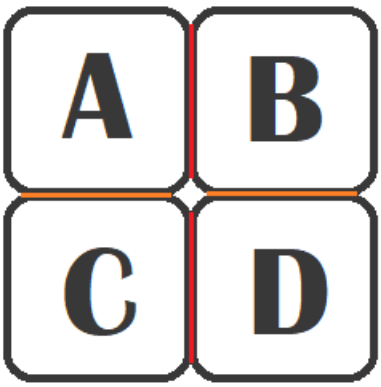


Figura 4.3: un conjunt de quatre vòxels

Es tracten de quatre vòxels, A, B, C i D, vists des d'una perspectiva aèria. Podem observar que estan connectats entre sí: A amb B i C, B amb A i D... La Figura 4.2 també és un bon exemple, tot i que una mica més difícil de veure.

Si observem el conjunt, ens podem fixar que les cares exteriors, les que no estan connectades amb cap altre vòxel, haurien de ser visibles. Però les cares interiors (subratllades amb colors a la figura), que és precisament on es connecten els vòxels, mai les podrem veure. No fa falta renderitzar aquestes cares, que estan consumint recursos innecessaris i no aporten cap informació visual al jugador.

Si eliminéssim aquestes cares, que no el vòxel sencer, ens quedaria el resultat que mostra la Figura 4.4:

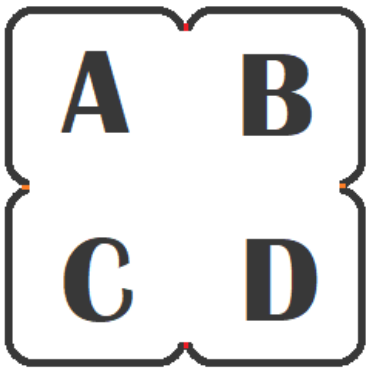
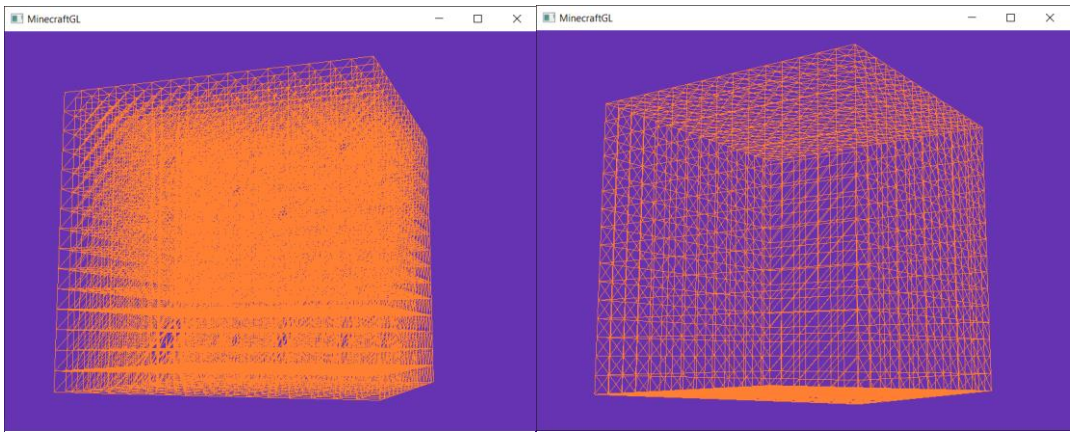


Figura 4.4: un chunk optimitzat

Potser amb 4 vòxels i una imatge en 2D no s'aprecia bé el gran canvi que això suposa. A continuació la comparació d'un chunk de 16*16*16 abans i després d'aplicar aquesta optimització:



a) Abans d'optimitzar *b) Després d'eliminar cares no visibles*

Figura 4.5: comparació de chunks

Visualment es pot veure el canvi i a més el podem calcular nosaltres mateixos. En el cas de la Figura 4.5, hem passat de 24576 cares a només 1536. Podem aplicar encara una altra tècnica de *culling* (eliminar cares que no podem visualitzar) que ja ve donada per OpenGL: el *face culling*. Només hem d'utilitzar una línia de codi, `glEnable(GL_CULL_FACE)`, i el resultat de la figura 4.5.b es converteix en la Figura 4.6:

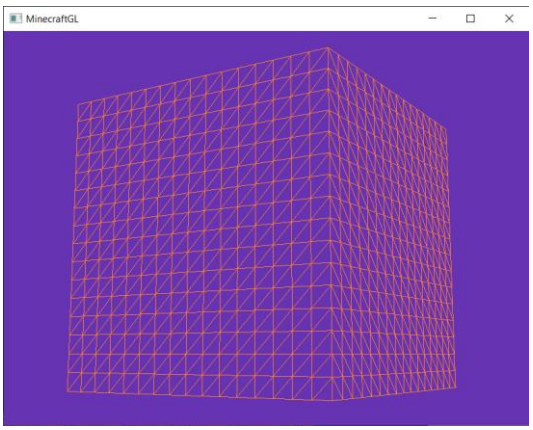


Figura 4.6: chunk optimitzat conjuntament amb la tècnica d'OpenGL

Quan el jugador canvia els chunks afegint o destruint vòxels, el motor ha de mantenir aquests canvis d'alguna manera per tal que aquests no desapareguin. Al nostre motor mantindrem un `map` amb els canvis, però això significa que, en tancar el joc, desapareixeran. Per poder mantenir-los definitivament, hauríem de guardar aquests canvis en arxius.

4.2.4. Vèrtex

Fixem-nos que la Figura 4.2 mostra un chunk sense optimitzar però que utilitza la tècnica d'OpenGL: les cares superiors, inferiors, les de l'esquerra i les del fons del tot no es veuen. Això és per com es renderitzen les cares.

Cada cara està formada per dos triangles. Aquests triangles, a la vegada, per 3 vèrtexs. Per tant, una sola cara té 6 vèrtexs. Cada vèrtex ens informa de la seva posició, és a dir, ens diu on l'hem de renderitzar. Al contrari del que es pot pensar, un vèrtex no només informa de la posició, sinó que també podem saber la normal o fins i tot el color. Podríem fer que cada vèrtex d'un vòxel tingués un color diferent, però per uniformitat farem que tots els vèrtexs amb informació comuna del vòxel tinguin la mateixa informació, com el color i el tipus.

Tornant al *face culling*, aquesta tècnica examina l'ordre dels vèrtexs de cada triangle. Si mirem la Figura 4.7, no és el mateix fer 1 -> 2 -> 3 que 3 -> 2 -> 1. El primer ordre és en el sentit de les agulles del rellotge, i l'altre és a la inversa. Això avisa a OpenGL que el primer ordre, si estiguéssim mirant tal i com es mostra a la figura, serveix per muntar un triangle que està mirant a la **mateixa direcció** que nosaltres, mentre que l'altre ordre està mirant cap a **nosaltres**. Si féssim tots els triangles amb la mateixa direcció de vèrtexs, no podríem veure els que hi ha a una part dels vòxels.

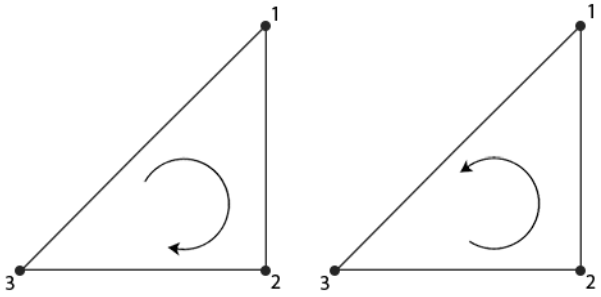


Figura 4.7: orientació de dos triangles

Un cop examinat l'ordre d'un triangle, **si aquest no està mirant cap a nosaltres**, no es renderitza. Per això a la Figura 4.6 només es veuen dos costats del chunk: són els que, des de la nostra perspectiva, tenen l'ordre al contrari de les agulles del rellotge.

4.2.5. VAO i VBO

Mirant la documentació d'OpenGL, sovint es poden confondre dos conceptes clau a l'hora de renderitzar: el VAO i el VBO. Comencem per VBO, o *vertex buffer object*.

Com ja hem explicat abans, tenim una sèrie de vèrtexs que, junts formen triangles. Aquests triangles formen cares i així aconseguim un vòxel. Aquests vèrtexs, però, s'han de guardar en algun lloc i s'han d'enviar a la GPU d'alguna manera. Aquí es on entren els VBO. Com el seu propi nom indica, es tracta d'un *buffer* on podrem desar

informació dels nostres vèrtexs, i permet desar molta informació a la vegada, útil per no haver d'enviar els vèrtexs un a un. El gran avantatge dels VBO és que, un cop a la GPU li arriba el VBO, el *vertex shader* (veure Apartat 4.1.6) té accés gairebé instantani als vèrtexs, fent la lectura increïblement ràpida. La part més lenta d'aquest procés és el pas de CPU a GPU dels vèrtexs, però com podem veure, un cop allà no ens hem de preocupar.

A la Figura 4.8 podem veure 3 vèrtexs. Cada un té 3 valors, però podríem posar encara més informació, com color o posició de les UV.

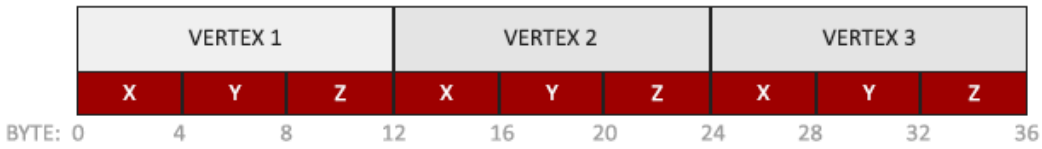


Figura 4.8: exemple de VBO.

Un VAO, o *vertex array object*, és un altre objecte d'OpenGL que serveix per "explicar" com està fet el VBO. Per exemple, podria ser que cada vèrtex es guardés la posició i el color que té. El VAO s'encarrega de dir-li a la GPU on ha de mirar al vèrtex per obtenir cada informació. A aquest punter se li denomina *attribute pointer*, i podem utilitzar la funció `glVertexAttribPointer` per definir-los.

A l'Apartat 4.1.6 veurem que els *shaders* fan ús d'aquests punters. Tècnicament no fa falta un VAO si el VBO només es guarda un tipus d'informació, però OpenGL ens demana tenir-ne un com a mínim.

Cada chunk tindrà el seu propi VBO, i el món un VAO. Tots els VBO estaran connectats a aquest VAO. En comptes d'anar enviant tots els vèrtexs del món a la GPU constantment, només hem d'actualitzar un VBO cada cop que eliminem o col·loquem un vòxel. Més endavant veurem que els chunks tindran en realitat dos VBO per un problema de transparència als vòxels.

Nosaltres en un sol vèrtex ens guardarem:

- Posició del vòxel
- Quantitat de llum
- UV
- Tipus del vòxel
- Quin costat del vòxel representa
- Color

Tots aquests representats amb bytes. Podríem fer que ocupessin bastant menys si féssim servir operacions de bits (potser un byte és massa pel costat que representa, per exemple, el podríem empaquetar millor), però això només afecta a la memòria utilitzada i no al rendiment com a tal, així que de moment ho deixarem com bytes. Cal notar que la posició són 3 bytes, les UV 2, etc. No fa falta que cada informació tingui el mateix nombre de bytes que les altres, ja que la funció d'abans permetrà definir quants bytes ha de mirar un punter.

4.2.6. *Shader*

Un *shader* és un programa petit escrit en GLSL (el llenguatge de programació de *shaders*) que, en compte de córrer en la CPU, ho fa en la GPU. D'aquesta manera permet controlar diferents parts del procés de renderitzat. Hi ha diferents tipus de *shaders* segons l'etapa que vulguem modificar. Els que nosaltres veurem i desenvoluparem són *vertex shaders* i *fragment shaders*.

El *vertex shader*, com el nom indica, s'executa sobre cada vèrtex que hem enviat a la GPU. La funció principal és transformar el vèrtex d'un espai 3D a l'espai 2D de la pantalla.

Per altra banda, el *fragment shader* opera sobre "fragments". Un cop el *vertex* acaba l'execució, un altre *shader* anomenat *geometry shader* genera la forma de l'objecte (aquest és opcional, ja s'encarrega d'aquest pas una part d'OpenGL). Després el rasteritzador s'encarrega de dividir les parts visuals de l'objecte en fragments de la mida d'un píxel. Aquí és on comença el treball del *fragment*, que modificarà el color i el retornarà perquè finalment es visualitzi a la pantalla. Per tant, el que podem programar en un *fragment shader* és com es veurà cada píxel a la pantalla.

Tot aquest procés és el que realitza la *graphics pipeline* d'OpenGL per poder veure tota la informació que li hem donat i retornar-nos una imatge en 2D (veure Figura 4.9).

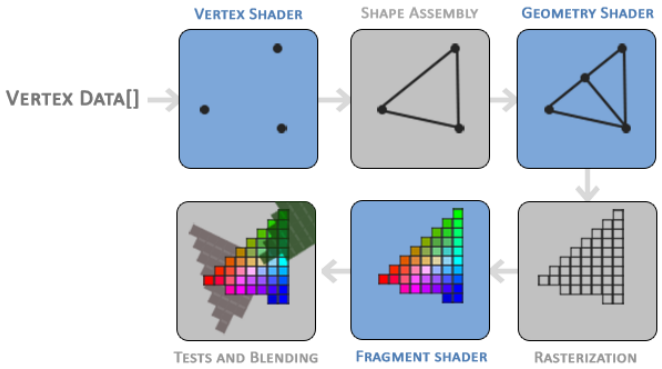


Figura 4.9: el procés que fa la graphics pipeline

A la Figura 4.10 podem veure un exemple del *vèrtex shader* que es troba al projecte.

```
1  #version 330 core
2
3  layout (location = 0) in vec3 aPosition;
4  layout (location = 1) in vec2 aTexCoord;
5
6  out vec2 TexCoord;
7
8  uniform mat4 model;
9  uniform mat4 view;
10 uniform mat4 projection;
11
12 void main()
13 {
14     gl_Position = projection * view * model * vec4(aPosition, 1.0f);
15     TexCoord = aTexCoord;
16 }
```

Figura 4.10: exemple de vertex shader

De moment no ens preocuparem pel que fa aquest *shader*, però veurem com està estructurat.

Els *shaders* comencen amb una línia que expressa la versió de GLSL que es farà servir. Just després podem veure dues línies que comencen de la mateixa manera. La línia `layout (location = 0) in vec3 aPosition` és la forma d'expressar del VAO que la primera cosa que trobarem al vèrtex és la posició. `location = 0` indica que és el primer. Si en volem més, com a la línia 4, anem incrementant. Això s'ha de veure reflectit al VAO. La resta de la línia significa que la informació que arriba és un vector de 3 *floats* anomenat *aPosition*.

La línia 6 mostra la manera de comunicar-nos amb el *fragment shader*. La paraula clau `out` expressa que el vector TexCoord li arribarà al *fragment* quan s'executi. Pel contrari, la paraula clau `uniform` a la línia 8 és la manera que tenim per enviar-li **informació que no es troba als vèrtexs** al *shader*.

A la línia 12 podem veure com el llenguatge GLSL es basa en C: `void main()` és la funció d'entrada, és a dir, des d'on comença a executar-se. La variable `gl_Position` és el resultat que ha de donar el *shader* per poder saber on col·locar el vèrtex a la pantalla. A la línia de sota observem que el que està fent és enviar-li al *fragment* la informació de coordenades de textura (UV) que li ha arribat en un vèrtex.

A continuació, a la Figura 4.11, veurem un altre exemple. Aquesta vegada és un *fragment shader*, però no és la continuació de l'anterior, sinó un altre diferent:

```
1  #version 330 core
2
3  out vec4 color;
4
5  in vec3 vertexColor;
6
7  void main()
8  {
9      color = vec4(vertexColor, 1.0);
10 }
```

Figura 4.11: exemple de fragment shader

L'estructura és la mateixa: comencem amb la versió de GLSL que volem utilitzar, declarem variables i implementem la funció `main()`. La diferència amb el *vertex* que té aquest *shader* és que `out` hauria de ser un vector de 4 *floats* que representa un color. Si tornem a la Figura 4.9, veurem que el que retorna el *fragment* és el color d'un fragment, per tant té sentit que el que ha de sortir sigui un vector amb 4 components: el color vermell, verd, blau i l'alfa (transparència). En aquest exemple, veiem a la línia 9 que el *shader* retorna el mateix color que li ha arribat del seu corresponent *vertex* (línia 5) amb màxima transparència.

Per veure un exemple real de com funciona un *shader*, a continuació modificarem el *fragment shader* que controla els núvols. Com no es un *shader* petit, especificarem més endavant el que fa detalladament quan parlem dels núvols. Ara mateix ens enfocarem en què podem fer si canviem el que retorna el *fragment.*

```
1  const vec4 fogcolor = vec4(0.7, 0.8, 1.0, 1.0);
2  const vec4 colorNuvols = vec4(1.0f);
3  ...
4
5  color = colorNuvols * colorText;
6  color = mix(fogcolor, color, fog);
```

Figura 4.12: retall del fragment shader dels núvols

A la Figura 4.12 trobem un retall del que hi ha al *fragment* que hem esmentat. Les dues primeres línies es troben al principi del *shader*, i les dues finals al final del `main()`. No fa falta saber què es colorText, només ens hem de centrar en que el *shader* retorna un color pels núvols i un altre per la boira. Ho veurem més clar a la Figura 4.13.

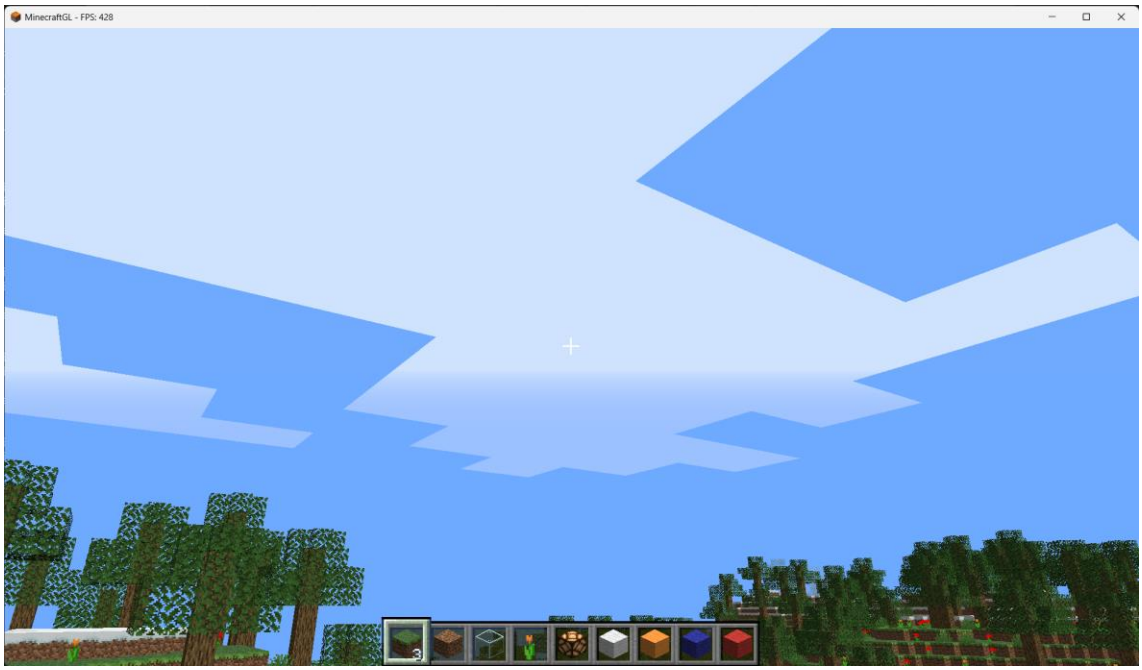


Figura 4.13: resultat del shader de la Figura 4.12

Els núvols obtenen dos colors diferents, però el *shader* només retorna un. Això és per com està fet el *shader* que, depenent de la distància, aplicarà un color o un altre. De fet és el que està fent a la línia 6. `mix()` és una funció que decideix quin color fer servir segons un tercer valor (`fog`). Si canviem la variable colorNuvols per un 0.0f en comptes d'un 1.0f, obtenim el resultat que es pot apreciar a la Figura 4.14.

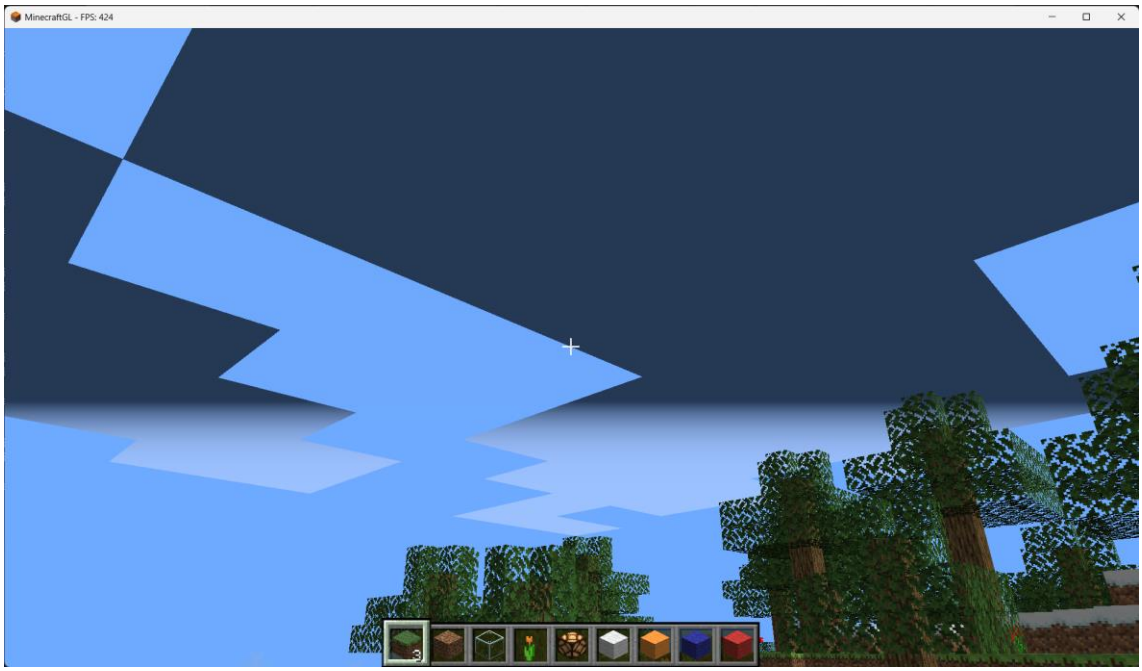


Figura 4.14: la variable colorNuvols retorna 4 zeros, que equival a negre

Encara podem obtenir resultats molt més diferents i llamatius (veure Figures 4.15 i 4.16).

```
1  const vec4 fogcolor = vec4(1.0f, 0.8, 0.0, 1.0);
2  const vec4 colorNuvols = vec4(1.0f, .5f, 0.0f, 1.0f);
3
4  ...
5
6  color = colorNuvols * colorText;
7  color = mix(fogcolor, color, fog);
```

Figura 4.15: el mateix shader, amb diferents valors

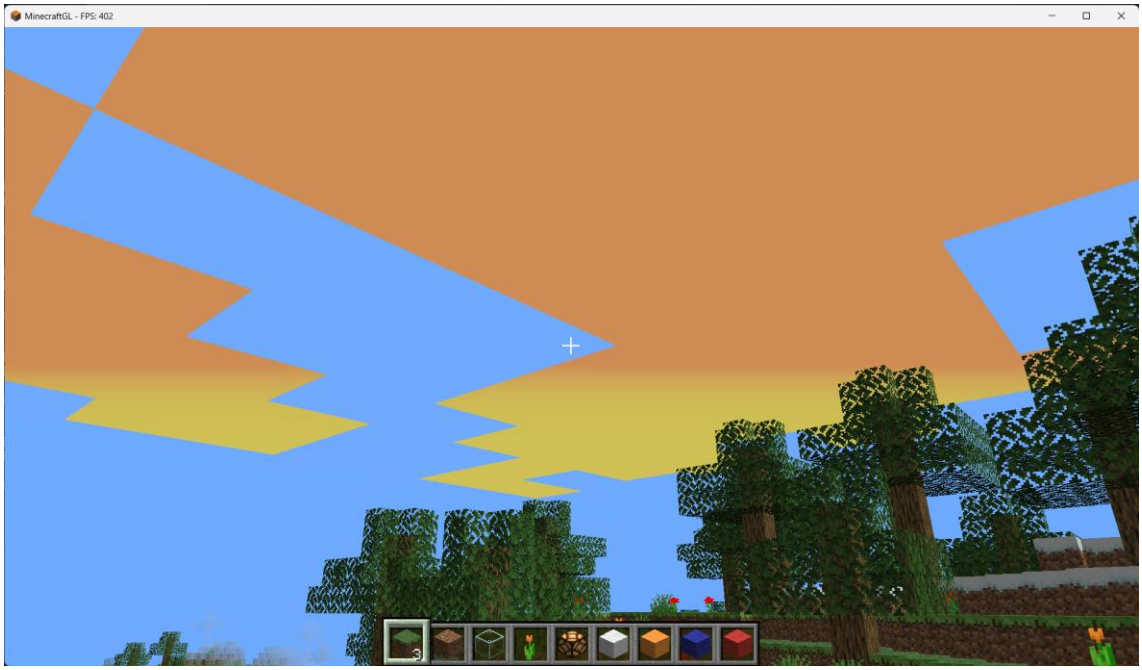


Figura 4.16: el resultat d'executar el shader de la Figura 4.15

Aquesta flexibilitat permetrà donar-li a l'usuari un grau de llibertat per tal que pugui confeccionar el món al seu gust.

4.2.7. Soroll

El soroll, en el context de generació d'efectes visuals i generació de terreny, és una senyal aleatòria que obtenim a partir d'un procés. Podem representar el soroll como una funció matemàtica que, donats un o més paràmetres, retorna un valor usualment comprés entre 0 i 1. Podem utilitzar aquest valor per saber l'altura que li hem de donar a un terreny, per exemple. Aquest tipus d'aplicació s'anomena *height map*.

El soroll es genera a partir d'una **llavor**. Aquesta llavor farà que una senyal de soroll sempre doni els mateixos resultats. En canviar la llavor, els resultats donats per la funció seran totalment diferents.

Hi ha diferents tipus de soroll: el més bàsic és el soroll blanc (*white noise*), que s'assembla al que apareix a les televisions antigues quan no rebien senyal (Figura 4.17). El podem obtenir si a cada parella `(x,y)` li donem un valor aleatori.

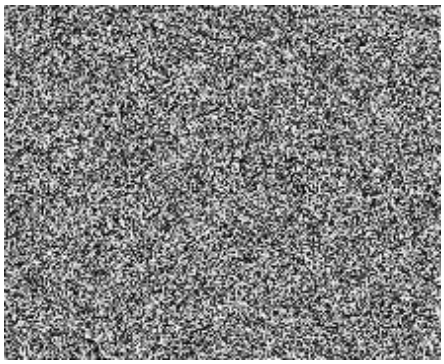


Figura 4.17: soroll blanc

El problema del soroll blanc és que cada valor és totalment aleatori, sense tenir en compte els seus veïns. Si generem terreny fent ús de soroll blanc, el resultat és el que trobem a la Figura 4.18.

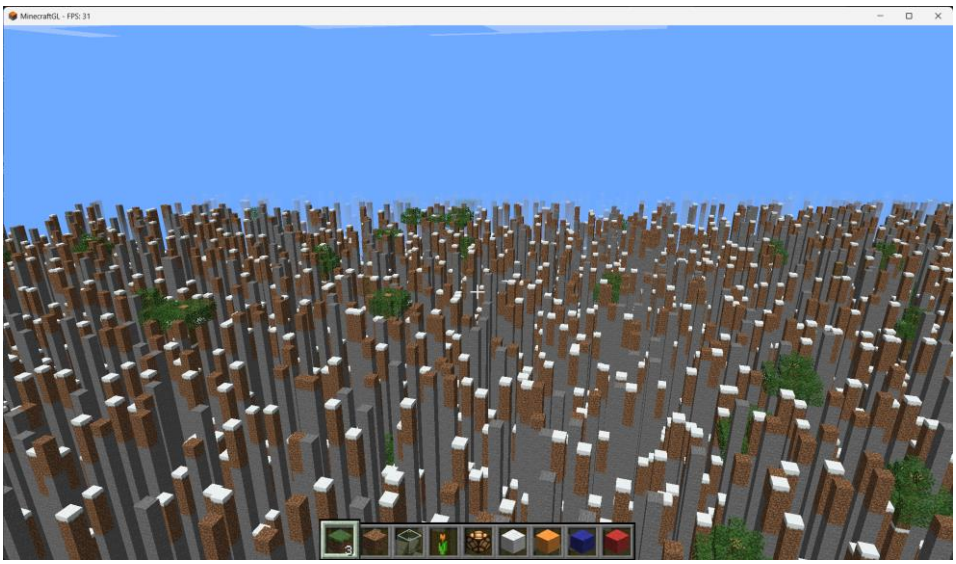


Figura 4.18: terreny generat amb soroll blanc

Obtenim un terreny molt abrupte que no s'assembla al món real, per això aquest tipus de soroll gairebé no s'utilitza amb aquests objectius en ment. El que necessitem és un **gradient de soroll**, que interpola entre els valors veïns, obtenint un soroll molt més suau i coherent. Tipus de gradients de soroll comuns son el *Perlin noise* (veure Figura 4.19) o *Simplex noise.*

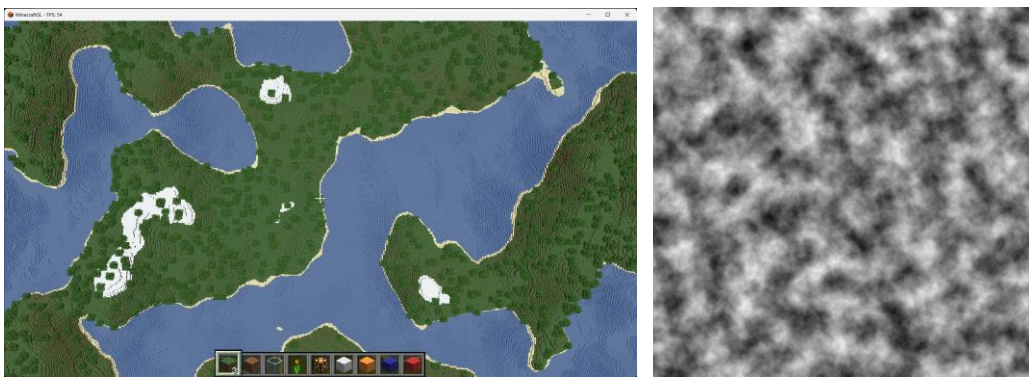


Figura 4.19: Aplicació en terreny de Perlin noise

Els núvols, per exemple, venen donats gràcies a una imatge formada a partir de soroll. Valors a partir d'un cert llindar generen un núvol i valors per sota retornen transparència.

4.2.8. Sprite i textura

Textura és sinònim de *bitmap* bidimensional, és a dir, una imatge. Un *sprite* és simplement una textura que és fàcilment transformable: es pot traslladar, escalar, rotar, etc. Aquests termes són importants, ja que qualsevol cosa que visualitzem al motor tindrà una textura associada. Si no, tindrien un color sòlid o com a molt un gradient.

Un *sprite* pot fer servir una textura sencera o potser una part d'ella. Si es tracta d'una part d'ella, usualment és perquè la textura es tracta d'un **mapa de textures**: un conjunt de textures més petites col·locades en una sola. A la Figura 4.20 podem veure el mapa de textures que fem servir al projecte per pintar els vòxels. Cada sub-textura és de 16*16, i tenim també 16*16 sub-textures, donant la possibilitat de tenir 256 textures en una sola imatge.

Figura 4.20: mapa de textures

Si tinguéssim totes les textures separades (que, en realitat, és el que fa *Minecraft* avui dia), hauríem de carregar-les totes. Amb un mapa de textures només hem de carregar una textura. El desavantatge d'això és que les sub-textures han de ser petites, però per nosaltres no és cap problema ja que visualment no queden malament (veure Figura 4.21).

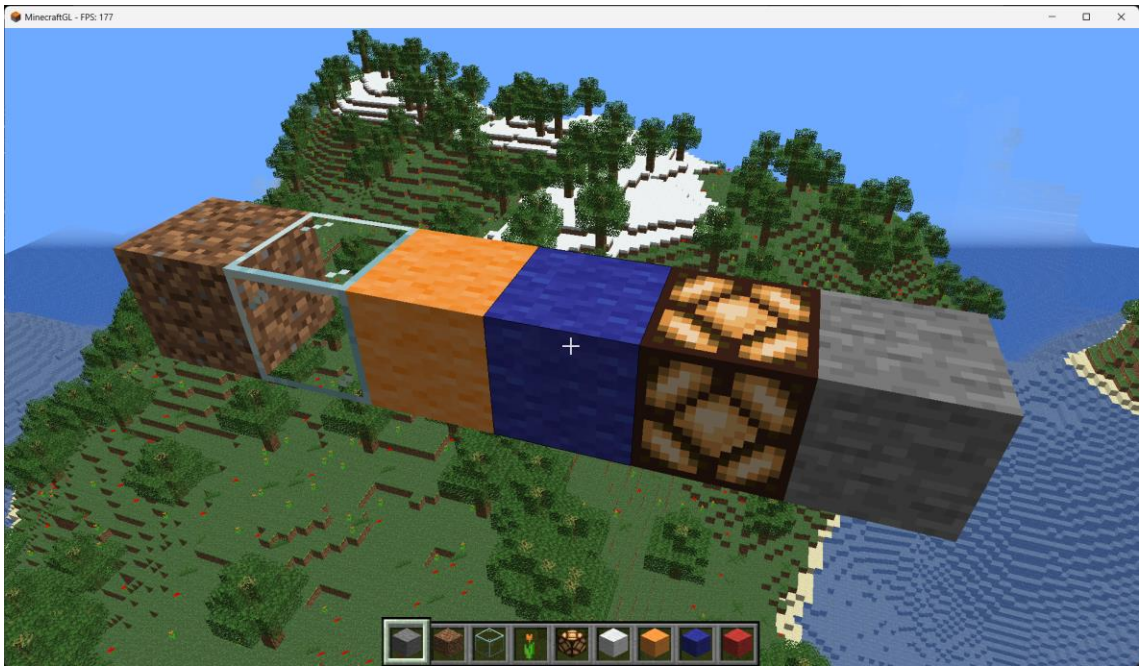


Figura 4.21: sis vòxels col·locats per un jugador vists des de prop

Permetre tenir un mapa de textures editable fa que l'usuari tingui la llibertat de modificar cada vòxel visualment com més li agradi, així com fer servir altres mapes de textures d'Internet.

4.2.9. *Thread*

Una de les raons d'escollir C++ és el rendiment. Tot i així, si fem moltes coses al bucle principal del programa, podem acabar amb baixades importants d'FPS (*frames per second*) i obtenir un rendiment pobre, semblant al de les còpies de les que ens volem allunyar. Per no sobrecarregar el bucle i per poder tenir un món dinàmic estable, s'han de fer servir *threads* (o fils en català).

Podem definir un *thread* com un tros de codi que és independent del codi principal i que el processador pot executar a la vegada. Per posar-ho en context del motor: mentre el fil principal (d'on surten la resta de fils) es dedicarà a gestionar els inputs i renderitzar el món, els altres fils s'encarregaran al mateix temps de gestionar-lo.

És important notar que les funcions d'OpenGL només es poden executar al fil principal i que per això precisament fem que sigui aquest l'encarregat de renderitzar l'escena.

5. Implementació del projecte

Ara que hem posat en clar tots els conceptes necessaris per poder entendre el motor, veurem què fa cada part i com s'ha implementat, explicant les diferents funcions d'OpenGL, les diferents tècniques d'optimització, etc. També veurem com l'usuari pot interactuar i modificar cada classe del motor per personalitzar-lo. S'ha dividit aquesta secció per cada classe que hi ha al projecte. Primer s'oferirà una descripció del propòsit de la classe i després el contingut: atributs i funcions.

Recordem que un dels objectius principals del projecte és documentar clarament el motor per tal que sigui entenedor d'inici a fi, així que veurem totes les funcions en detall.

5.1. Joc

La classe Joc és la classe principal des d'on es gestiona el *loop* principal, l'estat del joc i es tracta l'input del jugador. La funció `main` de C++ el que fa només és crear una instància d'aquesta classe i intentar crear una finestra. Si s'ha pogut crear, aleshores es crida la funció `start` per iniciar el joc. Un cop iniciat, comença la funció `loop`, que s'encarregarà del bucle infinit per renderitzar la resta d'elements. En aquesta classe es troba també la implementació de les tècniques per obtenir el vòxel al que està mirant directament el jugador.

5.1.1. Atributs

Per cada classe, farem esment dels atributs més destacables i quina és la funció de cadascun, així com l'efecte que pot tenir al motor si l'usuari el canviés. Pel Joc:

- `deltaTime` (`float`): informa del temps que ha passat d'un *frame* a un altre.
- `modeInventari` (`bool`): informa si estem a dins de l'inventari creatiu o no.
- `_VSync` (`bool`): activa o desactiva el límit d'FPS. Comença en `false`, però es pot posar a `true` per activar el límit.
- `_Culling` (`bool`): activa o desactiva la tècnica de *culling*. Comença en `true`.
- `mode` (`int`): informa del mode actual del jugador. Comença en mode `ESPECTADOR`.
- `nit` (`bool`): informa si al món es de dia o de nit. Comença en `false`, però es pot posar a `true` per començar de nit.
- `CubActual` (`vec3`): una referència al vòxel al que està mirant el jugador.
- `tecles` (`map`): es tracta d'un `map` que informa si una tecla està sent pressionada contínuament o no. Conté les tecles necessàries pel moviment del jugador, però es poden afegir totes les que es desitgin.

5.1.2. `crearFinestra`

És la primera funció que es crida en tot el programa. Crida a una altra funció de la classe Renderer encarregada de crear la finestra com a tal. Si s'ha pogut crear, farem que la classe Joc tingui una referència a aquesta finestra. En qualsevol cas, el que retorna la funció és si s'ha pogut o no crear la finestra per tal que la funció `main` pugui decidir si acabar el programa o no.

5.1.3. `start`

Com s'ha explicat abans, el que fa la funció és iniciar el joc. Això significa carregar els *shaders* necessaris per poder renderitzar el món. Com que aquests *shaders* són essencials, si no els hem pogut carregar per qualsevol motiu, no deixarem que el joc comenci.

Si s'han pogut carregar, però, hem de tractar primer algunes funcions d'OpenGL i de la llibreria GLFW, com podem veure a la Figura 5.1.

```
1  // Funcions de GLFW per contexte
2  glfwSetKeyCallback(window, key_callback);
3  glfwSetWindowUserPointer(window, reinterpret_cast<void*>(this));
4  glfwSetMouseButtonCallback(window, mouse_click_callback);
5  glfwSetScrollCallback(window, scroll_callback);
6  glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
7
8  glEnable(GL_DEPTH_TEST);
9  glEnable(GL_CULL_FACE);
10 glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
11
12 // Treure el cursor
13 canviarModeMouse(GLFW_CURSOR_DISABLED);
14 glEnable(GL_POLYGON_OFFSET_FILL);
15 glPolygonOffset(1, 0);
16
17 glfwSwapInterval(_VSync);
```

Figura 5.1: funcions inicials per gestionar diferents situacions

Les cinc primeres funcions comencen per "`glfw`", indicant que tenen a veure amb la llibreria GLFW. Recordem que aquesta llibreria proporciona les eines necessàries per construir una finestra i el context d'aquesta.

Aquestes funcions redireccionen esdeveniments de la finestra (que li hem donat el nom `window`) al joc per tal de poder programar-los, d'aquí que el nom de les funcions acabi amb "`callback`". Són bastant autoexplicatives: `glfwSetKeyCallback` permet programar què passa quan es pressiona o es deixa de pressionar una tecla, `glfwSetMouseButtonCallback` el mateix quan és un botó del ratolí, etc. Els paràmetres sempre són els mateixos: primer la finestra de la que volem redirigir l'esdeveniment i la funció que ha de cridar. La única que potser no es pot saber tan bé el que fa és `glfwSetFramebufferSizeCallback`: es crida quan es canvia la mida de la finestra. Veurem què fa cada funció en apartats següents.

La funció de la línia 3, `glfwSetWindowUserPointer`, ens permet "unir" la classe Joc amb la finestra. Hem de pensar que GLFW no sap el que és un Joc, així que a les funcions `callback` no tindrà manera d'accedir a aquest ni a cap funció ni atribut. La funció doncs ens deixa posar un punter a Joc, sempre i quan sigui un `void*` (d'aquí que fem `reinterpret_cast<void*>` per transformar el punter de la classe Joc, `this`, a un `void*`). Per poder obtenir el Joc, haurem de fer servir la funció `glfwGetWindowUserPointer` i tornar-ho a reinterpretar com un `Joc*`.

Les següents funcions són d'OpenGL, i ens deixen habilitar certs modes que seran útils més endavant:

- `GL_DEPTH_TEST`: juntament amb `GL_PLYGON_OFFSET_FILL` i la funció `glPolygonOffset` permetrà conèixer la profunditat d'un fragment.
- `GL_CULL_FACE`: activa la tècnica explicada a l'Apartat 4.1.4.
- `GL_FILL`: el complement és `GL_LINE`. Expressa si els polígons s'han de renderitzar emplenats o en *wireframe*.
- `GLFW_CURSOR_DISABLED`: fa que s'oculti el cursor, molt útil per aquest tipus de jocs en primera persona.

Per últim, `glfwSwapInterval` permetrà que els FPS (*frames per second*) s'adaptin als del monitor o que no hi hagi un límit.

Després d'aquestes inicialitzacions, hem de cridar als constructors de tres classes: `SuperChunk`, `HUD` i `Jugador`. Veurem cadascuna amb més detall més endavant.

5.1.4. `loop`

La funció principal del motor. Aquí es troba el bucle principal, que es repetirà contínuament fins que la finestra s'hagi de tancar. Abans de començar el bucle, aprofitem per iniciar tres matrius necessàries per poder visualitzar el món: model, projecció (que la definim a la funció `canviarProjeccio`, explicada a l'Apartat 5.1.5) i *view*. Ens seran útils quan parlem de la càmera i els *shaders*. També iniciem els núvols, que necessiten de la matriu de projecció.

Aquest és el moment d'iniciar dos *threads* per gestionar els *chunks*. A la Figura 5.2 trobem el codi per iniciar-los.

```
1  // Fem que un thread s'encarregui de la gestió de chunks
2  thread t([&]() {
3      while (!glfwWindowShouldClose(window)) {
4          mon->comprovarChunks(jugador->chunkActual());
5          mon->descarregarChunks();
6          mon->carregarChunks();
7
8      }
9  });
10
11 // En un thread apart fem que s'actualitzi el món
12 thread t2([&]() {
13     while (!glfwWindowShouldClose(window)) {
14         mon->update(jugador->chunkActual());
15     }
16 });
```

Figura 5.2: codi per iniciar dos threads

Podem observar que es tracten també de dos bucles amb la mateixa condició. La funció dins de la condició del `while` és la que s'encarrega de detectar si la finestra ha estat tancada. El primer fil s'encarrega de gestionar la càrrega i descàrrega de *chunks* i el segon d'actualitzar el món. De moment només ens interessa saber que es aquí on es tracta la

part del món dinàmic i més endavant la veurem en detall. Al final de la funció, quan s'acabi el bucle principal, hem de fer servir la funció `join` per poder eliminar els fils definitivament.

Ara podem iniciar el *loop* on renderitzarem l'escena. En aquest bucle hem d'executar les funcions en un ordre molt específic, si no podríem veure errors a l'hora de renderitzar. La primera cosa que fem és calcular el `deltaTime`: agafem el temps actual gràcies a la funció `glfwGetTime` i li restem la anterior, que actualitzem ara amb l'actual. Després, si no som a l'inventari creatiu, deixem que el jugador pugui moure la càmera.

Ara és el torn de renderitzar. Hem de fer-ho en l'ordre indicat a la Figura 5.3: primer actualitzem i renderitzem els núvols i després el món. Si ho fem a la inversa, veuríem els núvols per sobre del món. La variable `sotaAigua` ens dirà si el jugador es troba dins d'un vòxel d'aigua avaluant el vòxel on es troba ara mateix.

```
1  // Canvia el color del fons
2  if(!nit) glClearColor(rgb(colorCelDia->x), rgb(colorCelDia->y), rgb(colorCelDia->z), 1.0f);
3  else glClearColor(rgb(colorCelNit->x), rgb(colorCelNit->y), rgb(colorCelNit->z), 1.0f);
4  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
5
6  nuvols->update(jugador->obtPos2D(),deltaTime);
7  nuvols->render(view);
8
9  bool sotaAigua = mon->obtenirCub(jugador->obtPosBloc(false)) == AIGUA;
10 mon->render(&(jugador->obtCamera()->frustum), sotaAigua);
```

Figura 5.3: ordre de renderitzat

Podem veure també la funció `glClearColor` i `glClear` abans de renderitzar res. El que fem amb aquestes funcions és "netejar" el *frame* anterior. La primera posa un color de fons (a les línies 2 i 3 veiem que, si no és de nit, posa el color de dia i viceversa) i la segona neteja el que havíem renderitzat prèviament. Per això li hem de dir que netegi el *buffer* de color i de profunditat.

Un cop renderitzada l'escena, podem moure el jugador fent servir la funció `moure`, que només crida a la funció `moure` del Jugador, i obtenir el vòxel al que està mirant (veure Apartat 5.1.14). Si es tracta d'un cub vàlid, dibuixarem la seva *bounding box*. Després actualitzarem la matriu `view` fent servir la funció `lookAt` de la càmera.

A la part final del bucle hem de renderitzar el que veurem davant del tot: la interfície d'usuari. En el nostre cas, es tracta de l'inventari i el HUD. La funció `glfwSwapBuffers` s'encarregarà d'enviar tota la informació de color a la finestra, actualitzant el que podem veure. Just abans de tornar a començar, calcularem si ha passat un segon fent la diferència entre el *frame* actual i l'últim *frame* amb el que vam fer aquesta resta. Actualitzem aquest últim i posarem el comptador d'FPS a 0. Això es veurà reflectit al títol de la finestra, que GLFW deixa canviar fàcilment gràcies a la funció `glfwSetWindowTitle`. Podem veure que els *frames* s'aniran acumulant en cas que no hagi passat un segon. Tot això es pot veure a la Figura 5.4.

```
jugador->inventari->render(obtMousePos());
_HUD->render(jugador->obtPosBloc(), CubActual);

glfwSwapBuffers(window); // Volcar l'array de color a la finestra

// Mostrem els frames que hi ha hagut en un segon (fps)
if (currentFrame-ant>=1.0f) { // Si la diferència és 1, és que ha passat un segon
    ant = currentFrame;
    // Mostrem els frames que hem pogut processar
    glfwSetWindowTitle(window, ("MinecraftGL - FPS: " + to_string(fps)).c_str());
    fps = 0; // Resetejem el comptador

}

fps++;
```

Figura 5.4: final del bucle

5.1.5. `canviarProjeccio`

Aquesta funció actualitza la matriu de projecció que es fa servir per visualitzar el món. Per això, cridarem la funció que calcula la matriu a la càmera del jugador. Necessitem saber el nou *aspect ratio*, que ho dona la classe Renderer. Després col·loquem la matriu obtinguda al *shader* principal.

5.1.6. `framebuffer_size_callback`

Es crida quan es canvia la mida de la finestra. El que fem és actualitzar la mida del *viewport* (el que li diu a OpenGL la transformació de coordenades normalitzades, que van de -1 a 1, a coordenades de la finestra) fent servir la funció **`glViewport`** amb la nova altura i la nova amplada. Després cridem la funció de l'anterior Apartat 5.1.5 **`canviarProjeccio`** i actualitzem la classe Recursos amb les noves dades. El codi el podem trobar a la Figura 5.5.

```
glViewport(0, 0, width, height);
Joc* joc = reinterpret_cast<Joc*>(glfwGetWindowUserPointer(window));

joc->canviarProjeccio();
Recursos::width = width;
Recursos::height = height;
```

Figura 5.5: codi de la funció `framebuffer_size_callback`

Notem que hem hagut de reinterpretar, com hem explicat abans, el punter retornat per la funció **`glfwGetWindowUserPointer`** com un punter a una instància de Joc i així poder accedir a la part pública.

5.1.7. Funcions alternables

Entenem per funcions alternables aquelles que canvien l'estat de les opcions del joc, usualment avisant per consola del que s'ha fet. Totes fan funcions similars i la seva explicació és molt breu. A continuació una llista de les funcions que conté la classe Joc:

- **`Culling`**: activa o desactiva la tècnica de *face culling*.

- `VSync`: activa o desactiva el límit d'FPS.
- `Frustum`: activa o desactiva la tècnica de *frustum culling*.
- `HUDDebug`: crida la funció `modeDebug` del HUD.
- `Inventari`: entra o surt de l'inventari de creatiu. Si entra, s'ha de parar al jugador i mostrar el ratolí i, si surt, deixar que el jugador pugui moure's i ocultar el ratolí.
- `CanviaHora`: activa o desactiva el mode nit.
- `CanviarMode`: alterna entre mode `ESPECTADOR` o `CREATIU`.
- `canviarModeMouse`: canvia el mode del ratolí, mostrant-lo o ocultant-lo.

A l'Apartat 5.1.9 es faran servir la majoria d'aquestes funcions.

5.1.8. `obtMousePos`

Retorna la posició del ratolí. Fem servir la funció `glfwGetCursorPos` que proporciona GLFW i posem els dos `floats` `x` i `y` en un sol `vec2` (vector de dos elements).

5.1.9. `key_callback`

Aquesta funció es crida quan l'usuari prem qualsevol tecla. Per paràmetre reben l'acció que fa amb la tecla (`GLFW_PRESS` si la acaba de pressionar o `GLFW_RELEASE` si la ha deixat anar) i quina ha pressionat. Bàsicament el que fa la funció es cridar una altra funció de la classe Joc per no sobrecarregar aquesta, avaluant amb un `switch-case` la tecla que ens arriba. Al manual d'usuari (Capítol 10) explicarem què tecla fa cada cosa, de moment només ens interessa que aquesta funció és una mena d'intermediària entre GLFW i el joc, mantenint les funcions de *callback* senzilles. A la Figura 5.6 trobem un exemple.

```
1 // C: canvia mode espectador <-> creatiu
2 case GLFW_KEY_C:
3     joc->CanviarMode();
4     break;
5 // E: obrir inventari
6 case GLFW_KEY_E:
7     joc->Inventari();
8     break;
```

Figura 5.6: tros del `switch-case` de la funció `key_callback`

D'aquesta manera, facilitem ràpidament a l'usuari una manera de poder configurar les tecles i la seva funció.

Tot i així, hi ha un detall interessant d'explicar, i és la manera amb la que detectem si una tecla està sent pressionada contínuament. Com hem vist a l'Apartat 5.1.1, el `map` `tecles` conté la tecla i un `bool`. Quan pressionem una de les tecles del `map`, posem el seu `bool` a `true`. Quan la deixem de pressionar (`GLFW_RELEASE`), posem el `bool` de nou a `false`. Com aquests esdeveniments es detecten només un cop, evitem que repetidament es posi a `true`.

5.1.10. `scroll_callback`

Es crida cada vegada que el jugador gira la roda del ratolí i informa de quant ha girat mitjançant el paràmetre `yoffset`. El que fa només es cridar una altra funció de l'inventari del jugador que canvia l'objecte seleccionat segons aquest valor.

5.1.11. `mouse_click_callback`

Aquest cop es crida quan el jugador clica amb el ratolí. Podem saber quin botó ha estat clicat i si s'ha pressionat o s'ha soltat. Si s'ha pressionat l'esquerre, farem que es destrueixi un vòxel si no som a dins de l'inventari (Apartat 5.1.12) o que s'agafi un objecte al contrari. Si és el dret el que s'ha polsat, usarem l'objecte que tingui seleccionat el jugador (Apartat 5.1.13).

5.1.12. `DestruirCub`

Destruir un cub primer necessita, precisament, que tinguem un cub vàlid guardat. Per això, farem servir la funció `ObtenirCubMira`, que explicarem a l'Apartat 5.1.14. Si tenim un vòxel vàlid, cridem a la funció `canviarCub` del món i li passarem aquesta posició i el tipus `AIRE`, ja que el que volem és "eliminar" el vòxel. Veurem més endavant que els *chunks* mai eliminen vòxels, així que per donar la il·lusió de que el jugador ha destruït un bloc, el transformarem en un vòxel transparent d'aire.

5.1.13. `Usar`

La funció és molt senzilla: obté l'item que té el jugador seleccionat i, si és un item vàlid i un bloc, el col·locarem com un vòxel nou al món (veure Apartat 5.1.16). Aprofitem per explicar que un item és un objecte que el jugador pot tenir a l'inventari. Aquest pot ser un bloc o un item amb una funcionalitat especial (que en el motor no està implementat). Veurem en apartats futurs que un item té la seva pròpia classe.

5.1.14. `ObtenirCubMira`

D'alguna manera hem de saber a quin vòxel està mirant el jugador o no podríem implementar les mecàniques de destruir i col·locar blocs. Hi ha dues maneres de fer-ho: fent servir *raycast* o desprojectant coordenades de pantalla. Les dues maneres tenen avantatges i desavantatges:

- ***Raycast***: es tracta de crear un raig amb origen a la càmera del jugador i direcció on miri que pararà quan es trobi un vòxel o quan hagi anat `n` vegades endavant. `n` pot ser un nombre qualsevol superior a 0 (si no aniríem cap enrere), però quant més gran sigui, més trigarà en trobar un vòxel. L'avantatge és que dona igual el tipus de vòxel que sigui, mentre es trobi davant del jugador, el trobarà. El desavantatge que trobem, però, és el rendiment. Si bé és veritat que ho podríem implementar en un fil apart i pot ser eficient, en ordinadors més lents fer un *raycast* cada cop que el jugador giri la càmera comporta un cost important.

- **Desprojecció**: si hem renderitzat el món, que és un element en 3D, en una finestra en 2D, d'alguna manera hem de poder fer el procés invers. És a dir, desprojectar una coordenada en pantalla és fer el procediment per obtenir la posició al món d'un vòxel que prèviament hem projectat a la finestra. Pot

semblar un procés complicat, però simplement és fer la inversa de projectar, que és el que fem normalment. L'avantatge principal és la precisió i rapidesa amb la que es fa. Es tracta d'un càlcul molt ràpid de matrius i no té cap límit, al contrari que el *raycast*. Qualsevol píxel visible a la pantalla té el seu corresponent vòxel. I el desavantatge, depenent de com es miri, pot ser decisiu pel jugador o no: si un vòxel és transparent, no el podem seleccionar si no agafem alguna part visible. Per com funciona el mètode, si no veiem informació al píxel, no podem aconseguir el vòxel.

Comparant els dos mètodes, ens decidim per la desprojecció. Implementar-la és increïblement senzill, com podem observar a la Figura 5.7.

```
1  // Desprojectant coordenades de pantalla
2  int ww = Recursos::width;
3  int wh = Recursos::height;
4  float depth;
5  glReadPixels(ww / 2, wh / 2, 1, 1, GL_DEPTH_COMPONENT, GL_FLOAT, &depth); // Llegim la profunditat del pixel al que estem mirant
6  if (depth <= 0 || depth >= 1) {
7      CubActual = glm::vec3(-1, -1, -1);
8      return;
9  }
10
11 glm::vec4 viewport = glm::vec4(0, 0, ww, wh);
12 glm::vec3 wincoord = glm::vec3(ww / 2, wh / 2, depth);
13 glm::vec3 objcoord = glm::unProject(wincoord, jugador->obtCamera()->getView(), jugador->obtCamera()->getProjection(), viewport);
14 CubActual = glm::vec3(floorf(objcoord.x), floorf(objcoord.y), floorf(objcoord.z));
15 if((mon->obtenirCub(CubActual)) == AIGUA) CubActual = glm::vec3(-1, -1, -1);
```

Figura 5.7: implementació de la tècnica de desprojecció

Si recordem les funcions `start` i `loop` dels Apartats 5.1.3 i 5.1.4, vam parlar sobre la profunditat dels píxels. Aquesta característica és molt útil ara: si no estem mirant a cap lloc on hi hagi un píxel com, per exemple, al cel, no hauríem de guardar-nos cap vòxel.

La funció `glReadPixels` proporciona una eina molt potent: llegeix i retorna la informació de qualsevol píxel de la finestra. Li hem d'especificar la posició del píxel, quants píxels volem (en el nostre cas, només un), la informació que necessitem, el tipus d'informació i una variable on guardar-la. La línia 5 té tot això implementat: volem saber la profunditat del píxel que hi ha just al centre de la pantalla i la guardarem a `depth`. Si és negativa o major a 1, sabrem que no és un vòxel vàlid i per tant posarem `CubActual` a -1.

Si és una profunditat vàlida, significa que tenim localitzat un vòxel, però no sabem on és al món, només a la pantalla. Per desprojectar una coordenada farem servir una altra funció molt útil de la llibreria GLM: `unProject`. Aquesta funció necessita saber la coordenada on volem desprojectar i la seva profunditat (a la línia 12), la matriu *view,* la matriu de projecció i la mida del *viewport* (és a dir, la finestra de la que ja hem parlat). Amb tot això, la funció ens retornarà quina coordenada és al món 3D sense haver de preocupar-nos.

Per saber el vòxel, només hem de construir un `vec3` fent `floorf` de cada un dels elements. Això farà que qualsevol coordenada amb un valor de punt flotant es converteixi en un enter. Una de les característiques d'aquest mètode, a més, és que el rang per posar un cub és **infinit** sempre i quan es vegi on es posa.

Tot i escollir la desprojecció, s'ha deixat comentada al codi una possible implementació de *raycast*, però es recomana fer servir la que hem explicat.

5.1.15. `ObtenirCostat`

Ara que tenim el vòxel al que està mirant el jugador, hem de saber el costat. Per destruir el cub no fa falta, simplement amb saber la posició és suficient, però és necessari per col·locar un de nou, ja que no és simplement substituir-lo, sinó posar-lo al costat. Sembla que la tècnica de desprojecció ja no és tan útil, però té una funció que pot donar una idea de com obtenir el costat que volem.

La idea principal és renderitzar el vòxel observat d'una manera diferent. Si cada costat fos d'un color diferent, podríem saber molt fàcilment quin costat mirem pel color. Per desgràcia, quan renderitzem l'escena veiem que un cub té una textura determinada pel mapa de textures vist a la Figura 4.20 i pot ser un color qualsevol. Per tant, el que podem fer és renderitzar el `CubActual` sense tenir en compte aquest mapa ni temes d'il·luminació. Però fer això comporta dos problemes: el jugador veuria aquesta renderització i seria renderitzar tot el món un altre cop.

La solució és fer servir un *framebuffer* definit per nosaltres mateixos on podrem dibuixar només aquest cub per darrera, sense que ho vegi el jugador. El *framebuffer* és un *buffer* especial on podem renderitzar sense haver de bolcar informació de color a la finestra. A les classes Framebuffer i Chunk veurem en més detall com s'implementen les parts més tècniques d'aquesta idea i en aquesta funció `ObtenirCostat`, com ho muntem de manera que funcioni. A la Figura 5.8 podem veure el codi d'aquesta funció.

```
1  // Renderitzem només el cub que estem mirant d'una manera especial
2  renderer.DibuixarDarrera();
3  glClearColor(rgb(0), rgb(0), rgb(0), 1.0f);
4
5  // Canviem el shader que fem servir per un mes senzill
6  renderer.usarShader(1);
7  renderer.colocarMat4("view", jugador->obtCamera()->lookAt());
8  renderer.colocarMat4("projection", jugador->obtCamera()->getProjection());
9
10 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
11
12 bool renderitzat = mon->renderCub(CubActual.x, CubActual.y, CubActual.z);
13 if (!renderitzat) {
14     renderer.DibuixarFront();
15     return glm::vec3(-1);
16 }
17
18 int ww = Recursos::width;
19 int wh = Recursos::height;
20 // Llegim el color del costat que estem mirant
21 glm::vec3 color;
22 glReadPixels(ww / 2, wh / 2, 1, 1, GL_RGB, GL_FLOAT, &color);
23
24 // Tornem a posar el buffer i el shader per defecte i aixi dibuixem l'escena tal qual
25 renderer.DibuixarFront();
26
27 // Mirem quin costat és pel color que hem obtingut abans
28 auto it = colorsCostat.find(color);
29 if (it != colorsCostat.end()) return it->second;
30 return glm::vec3(-1);
```

Figura 5.8: codi de la funció `ObtenirCostat`

La funció **`DibuixarDarrera`** del Renderer és la que permet poder fer ús d'un *framebuffer* apart del principal. Tot el que es dibuixi a partir d'ara no es veurà per pantalla, sinó que es guardarà temporalment fins que es cridi **`DibuixarFront`**.

Com sempre que renderitzem, hem de posar un *shader*. Podríem fer servir el mateix que utilitza el món, però en aquest cas farem servir un de més senzill. En tot cas necessitem un altre cop la matriu *view* i la de projecció, ja que el cub estarà al mateix lloc que abans, en aquest sentit no ha canviat. El fons el posem de color negre. Ara li demanem al món que renderitzi només aquest vòxel i, si s'ha pogut renderitzar, llegim el color de la mateixa coordenada que abans emprant un altre cop la funció **`glReadPixels`**. Podem guardar el color en un **`vec3`**. Podem tornar a dibuixar al *framebuffer* principal, ja que ja hem renderitzat el cub i hem llegit la informació que volíem.

A la classe Joc tindrem un **`map`** amb la correspondència color ⇔ costat anomenat **`colorsCostat`**, així que ara només és qüestió de trobar el color obtingut al **`map`**. Si no existeix, que només passa en el cas que estiguem mirant al fons, retornem una posició invàlida. Altrament, retornem la posició relativa del vòxel colindant al que estava mirant el jugador. A la Figura 5.9 tenim representat el cub tal i com l'hem dibuixat a la línia 12 si no renderitzéssim en un *framebuffer* diferent (amb un fons blau per més claredat).

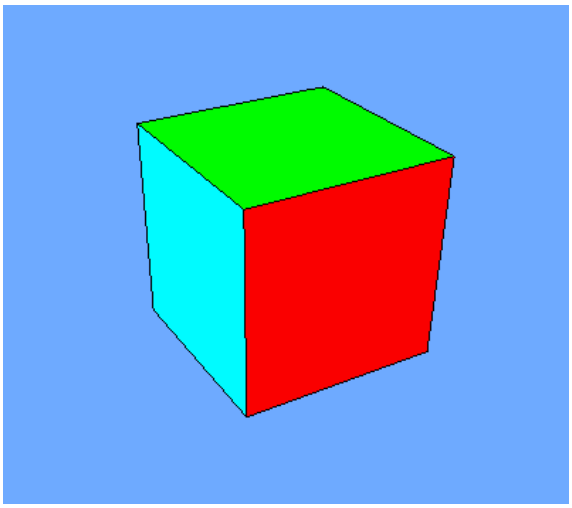


Figura 5.9: un cub renderitzat per poder diferenciar les seves cares

Això és el que veu el *framebuffer* secundari, tot i que el contorn no hi és i el fons és negre, els hem col·locat per una millor visualització. Observem que cada cara és d'un color diferent i que només hi ha un cub.

5.1.16. `PosarCub`

Aquesta funció col·loca un vòxel nou d'un tipus donat per paràmetre. Primer hem de mirar que tenim un `CubActual` vàlid i després obtenim el seu costat amb la funció descrita a l'apartat anterior. Si és un costat vàlid, aleshores calculem ràpidament la nova posició on es vol posar el vòxel i, si el jugador no està obstruint aquesta posició, aleshores fem que el món canviï aquella posició per un vòxel del tipus que ens han expressat. A la Figura 5.10 trobarem el codi de la funció.

```
1  // Si és un cub vàlid
2  if (CubActual.y == -1) return;
3
4  // Obtenim el costat al que estem mirant
5  glm::vec3 Costat = ObtenirCostat();
6  if (Costat.x==-1 && Costat.y==-1) return;
7
8  glm::vec3 posNova = CubActual + Costat;
9  if (jugador->obtPosBloc() == posNova || jugador->obtPosBloc(false) == posNova) return;
10
11 // Canviem el cub
12 mon->canviarCub(posNova.x, posNova.y, posNova.z, tipus, false, true);
```

Figura 5.10: codi de la funció `PosarCub`

5.2. Textura

En els següents apartats parlarem de les classes que serveixen per visualitzar el món. Primer començarem amb les classes més bàsiques i anirem escalant a classes que es relacionen directament amb Joc.

Una de les classes més bàsiques es la classe Textura. Serveix per guardar-nos una imatge i podrem especificar si es tracta d'un mapa de textures o és simplement un *bitmap* sol. Totes les imatges que es vagin a utilitzar com una textura al joc s'han de guardar a la carpeta `/Textures` del projecte.

5.2.1. Atributs

- `nom` (`string`): cada Textura haurà de tenir un nom per tal de poder identificar-la. Podríem haver utilitzat un enter, però amb un nom la podem identificar molt fàcilment, ja que les pròpies imatges ja tenen un nom a l'arxiu.
- `mapa` (`bool`): podem especificar si una Textura es tracta d'un mapa, és a dir, una col·lecció de textures, o no.
- `data` (`char*`): un punter privat a una col·lecció de bytes. Es guarda la informació de la imatge.
- `textura` (`int`): un identificador privat per tal que OpenGL pugui saber de quina imatge es tracta.
- `width`, `height`, `nrChannels` (`int`): informació rellevant de la textura. `nrChannels` ens diu quants canals de color està fent servir. No el farem servir, però és interessant tenir-lo.

5.2.2. Constructor

Tenim un constructor per defecte i un altre que és el que s'hauria de fer servir per poder utilitzar una Textura. El constructor per defecte simplement posarà un nom buit i inicialitzarà tot com si no hi hagués res. L'altre és el que més interessa a l'usuari, ja que construeix la Textura i la registra a OpenGL. Li hem de passar un nom per paràmetre per tal que la puguem identificar més endavant (veure Apartat 5.4). A la Figura 5.11 es troba el codi d'aquest últim.

```
1  Textura::Textura(string _nom)
2  {
3      nom = _nom;
4      string path = "Textures/"+_nom;
5
6      glGenTextures(1,&textura);
7      glBindTexture(GL_TEXTURE_2D, textura);
8
9      glEnable(GL_BLEND);
10     glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
11     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
12     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
13     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
14     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
15
16     data = stbi_load(path.c_str(), &width, &height, &nrChannels, 4);
17     if (data) {
18         glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, data);
19         glGenerateMipmap(GL_TEXTURE_2D);
20
21     }
22     else {
23         cout << "ERROR!!! No s'ha pogut carregar la textura " << path << endl;
24     }
25
26     stbi_image_free(data);
27     glBindTexture(GL_TEXTURE_2D, 0);
28
29 }
```

Figura 5.11: codi del constructor

Com podem observar, el nom que li passem per paràmetre ha de ser el mateix que el nom de l'arxiu que conté la imatge. D'aquesta manera podrem identificar la Textura ràpidament. Totes es guarden a la mateixa carpeta, però l'usuari és lliure de canviar això modificant l'`string` de la línia 4.

Fem servir funcions d'OpenGL per generar el *buffer* on es guarda la informació de la textura. Per això cridem `glGenTextures` i li passem l'identificador sense iniciar, ja que el propi OpenGL el modificarà. Amb `glBindTexture` li diem que estarem treballant a sobre d'aquesta textura. Les funcions de les línies 9 a 14 configuren la textura:

- `GL_BLEND`: permet renderitzar imatges semi-transparents. La següent línia només especifica com s'ha de tractar aquesta transparència, però l'usuari no s'ha de preocupar massa. Només saber que aquestes línies tracten l'alfa de les textures.
- `GL_REPEAT`: permet fer que la textura, en cas que no hi càpiga al polígon designat, es repeteixi de manera equivalent. Hi ha diferents modes, com `GL_MIRRORED_REPEAT`, però no el veurem ja que totes les textures caben perfectament.
- `GL_NEAREST`: aquesta sí que es una opció important pel motor que hem dissenyat. Amb `GL_NEAREST` especifiquem que la textura no tingui cap tipus de filtre. Si estiguéssim fent un motor per un joc diferent, potser ens convindria fer servir `GL_LINEAR`, que fa un filtre per tal que es vegin millor. A la Figura 5.12 podem veure la diferència entre un i altre. Observem que `GL_LINEAR` filtra la imatge, fent que en el nostre cas es vegi borrosa per la mida del mapa de textures.

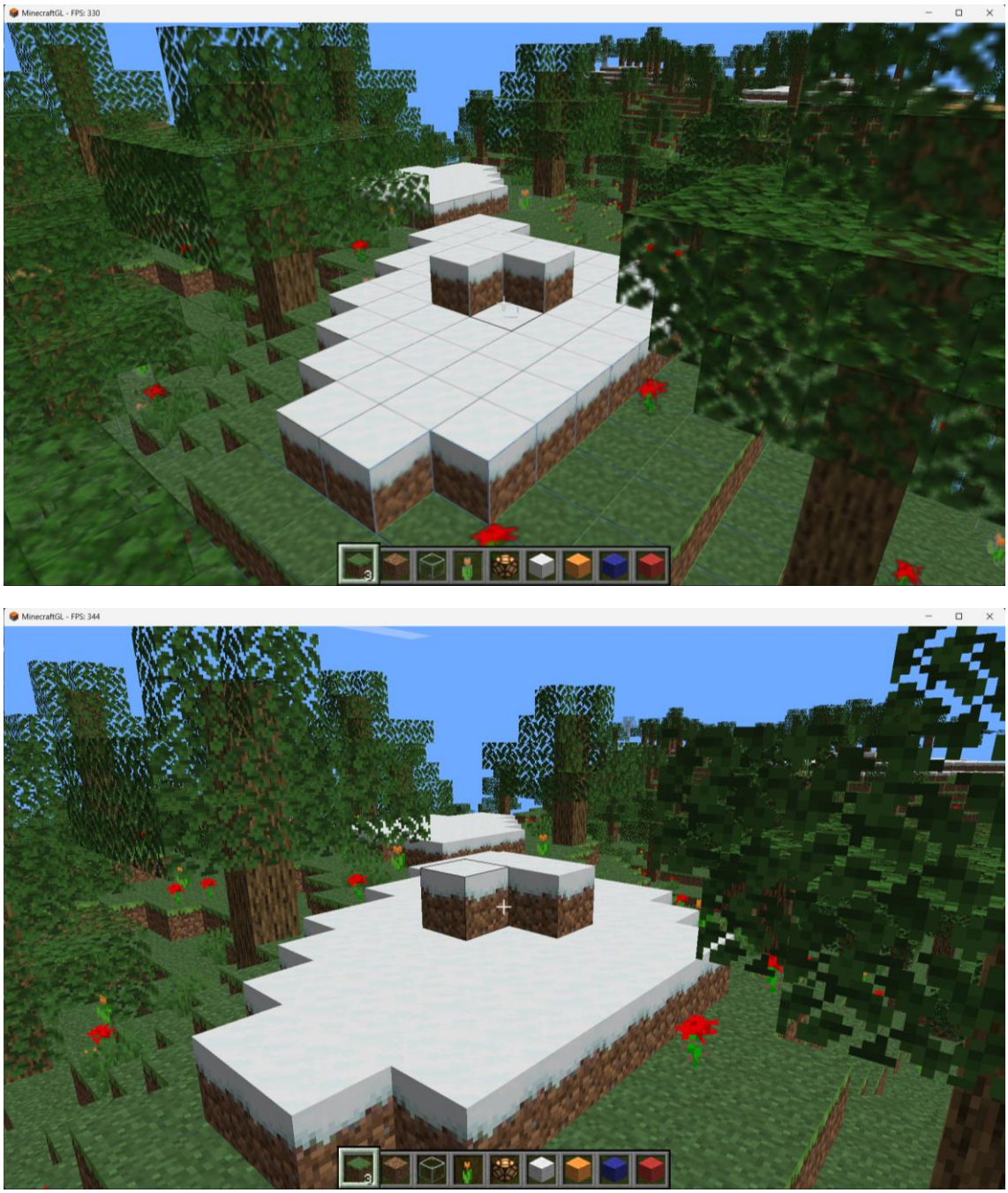


Figura 5.12: a dalt, textures fent ús de GL_LINEAR, a baix, fent ús de GL_NEAREST

Un cop configurada la textura, hem de carregar la imatge. Farem servir la funció `stbi_load` de la llibreria stb_image. Li hem de passar on es troba la textura, on pot guardar la informació, i quants components té. Hi ha un 4 perquè volem que hi hagi transparència (el quart component és l'alfa). El resultat es guarda a `data`.

Si s'ha pogut carregar, amb la funció `glTexImage2D` podem vincular data amb la textura amb la que estem treballant. Per això posem `GL_TEXTURE_2D`. No ens preocupem pel segon paràmetre, però el tercer especifica com volem que OpenGL guardi la textura (`GL_RGBA` farà que sigui transparent). Després li hem de dir la mida i, per qüestions internes de la llibreria, hem de ficar un 0 a continuació sí o sí. Els tres següents són el format de la imatge, que hauria de ser similar al tercer paràmetre, com hem guardat la

informació de la imatge i la informació com a tal. Amb tot això, OpenGL ja ha creat la textura i la ha guardat a l'identificador corresponent, llesta per poder utilitzar-la a l'hora de renderitzar.

La línia 26 neteja la imatge en memòria. Si ja la tenim a OpenGL, no ens fa falta tenir-la dos cops. La següent línia li diu a OpenGL que ja no fa falta continuar treballant amb la textura (0 significa cap textura connectada).

El destructor només crida a la funció `glDeleteTextures`, a la que li hem de passar l'id per destruir-la.

5.2.3. `use`

Aquesta funció li comunicarà a OpenGL que ha de fer servir aquesta textura. De fet ja hem vist com es fa: utilitzant `glBindTexture`. Només li hem de passar l'identificador i OpenGL s'encarrega de carregar-la per treballar amb ella. És literalment la línia 7 de la Figura 5.11.

5.2.4. `obtTamany`

Retorna un `vec2` amb el `width` i el `height` de la Textura.

5.3. Blocs i Items

Per poder visualitzar el món, hem de trobar alguna manera de guardar els tipus de vòxels que podem fer servir, així com informació sobre aquests: si es tracta de vegetació, si és un vòxel transparent, si el podem travessar, etc. La classe Blocs es tractarà d'una base de dades que es guardarà tota mena d'informació dels vòxels. Tot i que encara no la farem servir, és important notar que la classe Items es gairebé idèntica, però la farem servir a l'Apartat 5.18, quan parlem de la classe Inventari.

5.3.1. Atributs

- `dades` (`vector<Bloc*>`): es guarda tots els blocs carregats. Més concretament, punters a aquests `structs` per poder fer-los servir com a referència al motor.

5.3.2. Bloc

Es tracta d'un `struct` que es guarda tota la informació d'un sol vòxel:

- `nom`: el nom del tipus de vòxel. Per exemple: "Terra", "Pedra", etc.
- `id`: l'identificador del vòxel.
- `transparent`: el vòxel deixarà passar la llum.
- `semitransparent`: reservat especialment per aigua, ja que és transparent però també té color.
- `sòlid`: el vòxel es pot travessar pel jugador.
- `vegetació`: per saber si s'ha de renderitzar d'una manera o una altra.
- `costats`: id de la textura que s'ha de fer servir pels costats del vòxel.
- `sota`: id de la textura per la cara de sota del vòxel.
- `adalt`: el mateix, però per la de dalt.

5.3.3. Constructor

El constructor emplenarà el vector dades amb tota la informació necessària. Per això, primer crearem tots els punters que puguem fer servir. Alguns estaran buits, ja que pot ser que un tipus no s'hagi especificat, però altres tindran informació. L'ordre ens ho donarà el mapa de textures de la Figura 4.20. Si comencem amb 0, el vòxel 0 és aire, l'1 és gespa, etc. No podem simplement posar un tipus on vulguem, ha de seguir l'ordre dibuixat al mapa. A la Figura 5.13 tenim el codi del constructor.

```
1  Blocs::Blocs() {
2
3      dades.assign(MAX_BLOCS,NULL);
4      for (int i = 0; i < MAX_BLOCS; i++) {
5          dades[i] = new Bloc();
6      }
7
8      ifstream arxiuBlocs("./Tipus/blocs.json"); // arxiu json amb tots els tipus de blocs
9      json jsonBlocs = json::parse(arxiuBlocs);
10
11     auto it = jsonBlocs.begin();
12     cout << "Carregant blocs..." << endl;
13
14     // Per tots els blocs que hi hagi a l'arxiu
15     while (it != jsonBlocs.end()) {
16         json data = *it;
17         int id = data.value("id", 0);
18         string nom = data.value("nom", "");
19         dades[id] = new Bloc{ nom, id, data.value("transparent", false), ...
20         it++;
21     }
22
23     cout << "Blocs carregats!" << endl << endl;
24 }
```

Figura 5.13: codi del constructor de Blocs

Les línies 2 a 5 són les que hem explicat. A la línia 7 trobem la manera de examinar la "base de dades": fent servir un arxiu JSON. Per això fem servir la llibreria nlohmann/json. A la línia següent construïm un objecte `json` amb l'arxiu que hem especificat. L'arxiu es troba a `/Tipus/blocs.json`. A la Figura 5.14 trobem dos tipus de vòxels especificats: aire i gespa. Podem veure que, com l'aire es transparent i no és sòlid, ho especifiquem. A gespa, per contra, no s'ha de posar perquè per defecte és al contrari: normalment un vòxel no és transparent però sí és sòlid. La particularitat de la gespa és que els costats tenen una textura, per sobre una i per sota una altra.

```
1  "aire":{
2      "nom": "Aire",
3      "id": 0,
4      "transparent": true,
5      "solid": false
6  },
7  "gespa": {
8      "nom": "Gespa",
9      "id": 1,
10     "costats": 4,
11     "sota": 3
12 },
```

Figura 5.14: exemple de dos tipus al fitxer blocs.json

És l'usuari el que ha de confeccionar aquesta llista. A l'arxiu `blocs.json` hi ha una llista amb 45 tipus predefinits, que són els vòxels que el jugador pot fer servir a l'inventari. L'usuari pot canviar qualsevol d'aquests detalls per fer un nou tipus.

Fent servir un iterador, carregarem tots els tipus del JSON al vector, que és el que fa el `while` de la línia 14. Com veiem a la línia 17, obtenim un valor del JSON amb la funció

`value`. Li especifiquem quin atribut volem, com l'id, i ens ho retornarà. Si no troba l'atribut, retornarà el que nosaltres especifiquem com a segon paràmetre. Per tant, si no troba un id, retornarà 0. La línia 19 és la que crea el Bloc amb la informació. Ha estat tallada per fer-la més breu, però s'intueix que va emplenant el nou Bloc amb tot el que hi ha al JSON. Amb tot això, finalment tindrem un vector ple dels blocs que es troben al fitxer.

El destructor simplement itera sobre el vector eliminant els punters.

5.3.4. `getBloc`

Retorna un punter a on es troba la informació del Bloc amb l'id que passem per paràmetre.

5.3.5. Tipus.h

Hi ha un arxiu anomenat Tipus.h que no té a veure amb aquesta classe, però és important esmentar-lo. Conté un `enum` amb noms de vòxels i un enter associat. Aquest fitxer no interactua amb el motor com a tal, sinó que és per l'usuari. Per exemple, `AIGUA` té l'id 205. Si l'usuari llegís 205 en algun lloc, potser no sap al que ens referim. És molt més senzill escriure `AIGUA`, ja que ens dona una millor idea del que volem expressar. No fa falta que hi hagin tots els vòxels, només els que no volem escriure l'id directament. A la Figura 5.15 trobem un tros de l'`enum`.

```
1  enum TIPUS_BLOC {
2      AIRE = 0,
3      GESPA = 1,
4      PEDRA = 2,
5      TERRA = 3,
6      FUSTA_PLANXA = 5,
7      MAONS = 8,
8      ROSA = 13,
9      ...
```

Figura 5.15: contingut de Tipus.h

5.4. Recursos

Ara que sabem com estan organitzades les classes Textura, Blocs i Items, podem veure de què tracta aquesta classe Recursos que ha anat apareixent al llarg del codi. Aquesta classe està destinada a diversos objectius, però el principal és proporcionar una classe global a la que es pugui accedir des de qualsevol lloc. No és una bona idea tenir variables globals, però amb aquesta llista d'objectius ens quedarà clar per què ens fa falta:

- **Precarregar textures**: una bona idea per gestionar textures és precarregar totes les imatges que necessitem i crear des del principi les instàncies de Textura que s'hagin de fer servir al motor. A més, veurem que els Sprites (Apartat 5.14) tenen una textura associada, i si molts agafen la seva textura del mateix mapa, no haurien de carregar la imatge del mapa cada cop que fem un sprite. Per tant, la idea és tenir alguna manera de gestionar les textures i accedir a aquestes fàcilment.
- **Informació de la finestra**: si bé és veritat que hi ha una classe Renderer (Apartat 5.7) que s'encarrega de gestionar la finestra, és molt útil saber, per exemple, l'altura i l'amplada d'aquesta. Passar la finestra o el Renderer cada cop que volem obtenir aquesta informació és molt pesat pel programador.
- **Gestionar les bases de dades**: tenim les classes Blocs i Items, però si volem accedir a aquesta informació, han d'estar presents a algun lloc. Igual que la finestra i les textures, només hauríem de tenir una base de dades, i anar passant-la per les classes no és una bona idea ja que hi ha diferents classes que han de tenir accés.

Amb aquestes justificacions, hem trobat que aquesta classe serà molt útil a l'hora de gestionar aquestes dades.

5.4.1. Atributs

- `Blocs` i `Items`: la base de dades de Blocs i Items.
- `_textures` (`map<string,Textura*>`): un `map` amb totes les textures que hem precarregat. El nom de la Textura és la clau.
- `texturesPrecarrega` (`vector<string>`): un vector amb tots els noms de les textures que s'han de carregar.
- `width` i `height` (`int`): l'amplada i l'altura de la finestra. Abans hem vist que el Joc les actualitza.
- `COLORS` (`vector<vec3>`): un vector de colors per tal de tenir-los ja carregats. Ens referirem a aquests gràcies a un `enum` amb el nom de cadascun.
- `jocAcabat` (`bool`): informa si ha acabat el joc.

5.4.2. *Getters*

Aquesta classe es composa principalment de *getters*: funcions que retornen un valor. Podem obtenir un punter al Bloc, Item o Textura que vulguem amb l'id o el nom d'aquests gràcies a `getBloc`, `getItem` i `obtTextura`. Aquest últim, a més, si veu que el vector de textures és buit, l'emplena amb el vector de noms creant les textures adients. D'aquesta manera tindrem tots els blocs, items i textures compactats en un sol lloc al que podem accedir des de qualsevol classe sense haver d'estar passant referències o creant-los constantment. El mateix fem amb els colors.

5.5. ShaderProgram

Com hem vist a l'Apartat 4.2.6, un *shader* és un petit programa que s'executa a la GPU. Al tractar-se d'un programa, aquest també s'ha de compilar, a més que pot rebre diferents paràmetres des del programa principal un cop executat. Per això hem creat la classe ShaderProgram, que gestionarà una parella de *vertex shader* i *fragment shader*.

5.5.1. Atributs

- `vertexShaderSource` (`string`): i l'equivalent al *fragment*. Són els noms dels *shaders* que hem d'anar a buscar. Es poden passar per paràmetre al constructor per tal de carregar un *shader* propi. Es troben a la carpeta `/Shaders`.
- `shaderProgram` (`int`): l'id per OpenGL dels *shaders*. Té un *getter*.

5.5.2. `carregaShaders`

Per carregar els *shaders* hem de llegir l'arxiu on es troben i deixar que OpenGL els compili. La funció retornarà 1 o -1 depenent si s'ha pogut completar o no. El codi és llarg però molt senzill d'entendre. A la Figura 5.16 podem veure com es compila un *shader*. El procés és el mateix tant pel *vertex* com el *fragment*.

```
1  // Vertex Shader
2  unsigned int vertexShader = glCreateShader(GL_VERTEX_SHADER);
3
4  ifstream vertexShaderStream(vertexShaderSource); // Obrim l'arxiu
5
6  if (!vertexShaderStream.is_open()) return -1; // Si no es pot obrir, sortim de la funció
7
8  // Llegim el codi en un sol string
9  string vertexShaderCodi;
10 stringstream sstr;
11 sstr << vertexShaderStream.rdbuf();
12 vertexShaderCodi = sstr.str();
13
14 vertexShaderStream.close(); // Tanquem l'arxiu
15
16 // Compilem el Vertex Shader
17 cout << "Compilant Vertex Shader... ";
18 char const* vertexShaderPunter = vertexShaderCodi.c_str(); // Per compilar no es permeten strings
19 glShaderSource(vertexShader, 1, &vertexShaderPunter, NULL);
20 glCompileShader(vertexShader);
21
22 // Comprovem si hi ha errors
23 int success;
24 char infoLog[512];
25 glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
26 if (!success)
27 {
28     glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);
29     cout << "ERROR: No s'ha pogut compilar el Vertex Shader\n" << infoLog << endl;
30     return -1;
31 }
32
33 cout << "Vertex Shader compilat" << endl;
```

Figura 5.16: codi per compilar el vertex shader

Li demanem a OpenGL que generi un id pel *shader*, especificant que és un *vertex*. Després hem d'obrir l'arxiu (si no s'ha pogut obrir, retornarem un -1) i llegirem tot el que hi ha dins, guardant-ho en un `string` molt llarg (línies 9 a 14). Per tal que OpenGL ho pugui compilar, hem de convertir l'`string` en un punter a `chars`. Li passem aquest

codi a la funció `glShaderSource` juntament amb l'id i el compilem amb `glCompileShader`. El procés és molt senzill i mecànic. Després per comprovar si hi ha hagut algun error al compilar, com que el programador hagi comés algun error escrivint el *shader*, obtindrem l'estatus de la compilació (línia 25) i, depenent si ha anat bé o no, mostrarem l'error i retornarem -1 (línies 26 a 31). Farem el mateix amb el *fragment*.

Un cop compilats, hem de crear un identificador pel programa que conté els dos *shaders* i connectar-lo a OpenGL, especificant quins dos *shaders* són (veure Figura 5.17).

```
1  // Connectem els shaders amb el programa
2  cout << "Fent link dels shaders..." << endl;
3  shaderProgram = glCreateProgram();
4  glAttachShader(shaderProgram, vertexShader);
5  glAttachShader(shaderProgram, fragmentShader);
6  glLinkProgram(shaderProgram);
7
8  glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
9  if (!success) {
10     glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);
11     cout << "ERROR: No s'ha pogut crear el link entre el programa y els shaders\n" << infoLog << endl;
12     return -1;
13 }
14
15 glDeleteShader(vertexShader);
16 glDeleteShader(fragmentShader);
17
18 cout << "Shaders compilats: {" << vertexShaderSource << ", " << fragmentShaderSource << "}" << endl << endl << endl;
19
20 return 1;
```

Figura 5.17: codi per connectar els shaders a OpenGL

Amb `glAttachShader` connectem els *shaders* al programa que hem creat a la línia 3, i amb `glLinkProgram` connectem el programa a OpenGL. Comprovarem si hi ha hagut algun error com abans i, si està tot bé, no necessitem els *shaders* tenint el programa connectat. Retornem 1 si ha anat tot bé.

5.5.3. `usar`

Li diu a OpenGL que ha de fer servir aquest programa de *shaders* a l'hora de renderitzar. En una sola línia: `glUseProgram(shaderProgram)`. Es tracta d'una funció donada per OpenGL i només li hem de passar l'id.

5.5.4. `obtenirUniform`

Un `uniform` és la manera que té l'usuari de comunicar-se des de fora amb un *shader*. Té un nom propi, que li hem de passar per paràmetre a la funció. Aquesta retorna l'id del uniform que té guardat OpenGL. Com la funció anterior i la que veurem al següent apartat, es pot fer en una sola línia bastant autoexplicativa: `glGetUniformLocation(shaderProgram, uniform)`.

5.5.5. `colocar...`

Un `uniform` pot ser de molts tipus diferents, com un vector o una matriu. Al cap i a la fi, no deixa de ser una variable que el programador col·loca des de fora. Per poder donar-li un valor, s'han creat una sèrie de funcions que, segons el tipus de variable que volem posar, criden una funció d'OpenGL determinada. La llista de funcions (la del motor i la que crida d'OpenGL) és la següent:

- `colocarMat4: glUniformMatrix4fv`

- `colocarVec4`: `glUniform4fv`
- `colocarVec3`: `glUniform3fv`
- `colocarVec2`: `glUniform2fv`
- `colocarInt`: `glUniform1i`
- `colocarFloat`: `glUniform1f`

Totes fan el mateix: col·locar el tipus que expressa el nom. Només li hem de passar el nom del `uniform` que volem modificar i el valor.

5.6. Framebuffer

A l'apartat 5.1.15, a la funció `ObtenirCostat`, hem parlat de com necessitàvem un *buffer* diferent al principal per poder dibuixar en algun lloc que no sigui a la finestra, ja que el jugador veuria el vòxel si aquest fos el cas. Un *framebuffer* és simplement una col·lecció de *buffers* amb l'objectiu de poder renderitzar. A l'hora de renderitzar hem vist que en fem servir dos *buffers*: color i profunditat. En aquest *framebuffer* personalitzat només volem el de color, tot i que l'usuari es lliure de modificar la implementació per afegir-ne més *buffers*.

5.6.1. Atributs

- `fbo` (`int`): id pel *framebuffer*. Té un *getter* (`ObtID`).
- `textura` (`int`): veurem que per tal que es pugui guardar informació de color, necessitarà generar una textura molt senzilla (per això no fem servir la classe Textura). Té un *getter* (`ObtTextura`).

5.6.2. Constructor

El constructor per defecte només inicialitza els ids a 0, però tenim un altre al que li hem de passar la mida de la finestra per tal que pugui renderitzar exactament de la mateixa manera que el *framebuffer* per defecte. El que farem en aquest constructor és generar el *framebuffer* i connectar-lo a OpenGL (veure Figura 5.18).

```
1  Framebuffer::Framebuffer(int width, int height) {
2      // Generar framebuffer
3      glGenFramebuffers(1, &fbo);
4      glBindFramebuffer(GL_FRAMEBUFFER, fbo);
5
6      // Creem la textura pero encara no posem res!
7      glGenTextures(1, &textura);
8      glBindTexture(GL_TEXTURE_2D, textura);
9      glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, 0);
10
11     // Li posem la textura
12     glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, textura, 0);
13
14     // Comprovem errors
15     if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE) {
16         cout << "Hi ha hagut un problema amb el framebuffer." << endl;
17         return;
18     }
19
20     // Fem que torni a la finestra
21     glBindFramebuffer(GL_FRAMEBUFFER, 0);
22
23 }
```

Figura 5.18: codi del constructor per crear un framebuffer

El codi és molt senzill: primer hem de generar l'id del *framebuffer* amb `glGenFramebuffers`. Com hem pogut veure al llarg del codi, OpenGL sempre proporciona funcions que comencen amb `glGen` per generar ids d'objectes que s'hauran de connectar més endavant, així que l'usuari es començarà a acostumar a veure funcions d'aquest tipus. A la següent línia especifiquem que treballarem amb aquest nou *framebuffer* passant-li l'id a `glBindFramebuffer`.

Les línies 6 a 9 són familiars: les hem vist a l'Apartat 5.2.2 quan creàvem una Textura. En aquest cas no fa falta configurar-la tant ja que no es veurà en cap moment. Un cop creada la textura, amb `glFramebufferTexture2D` podem connectar-la al *framebuffer* que hem generat. `GL_COLOR_ATTACHMENT0` indica que volem posar la textura al *buffer* de color. Tot i que podem tenir fins a 8 *attachments* de color, amb un fem prou. Després comprovem que no hi hagi errors determinant si l'estatus del *framebuffer* és igual a `GL_FRAMEBUFFER_COMPLETE` i retornem el control al *framebuffer* donat per OpenGL passant un 0 a `glBindFramebuffer`.

El destructor crida a la funció `glDeleteTextures`, igual que una Textura normal.

5.6.3. `Unir` i `Desunir`

Aquestes funcions les podem cridar cada cop que vulguem dibuixar al *framebuffer* i que el jugador no vegi el que es renderitza o retornar el control al que hi ha predeterminat per OpenGL i dibuixar a la finestra. Són, respectivament, les línies 4 i 21 de la Figura 5.18. És a dir, criden `glBindFramebuffer` i col·loquen l'id o un 0.

5.7. Renderer

Necessitem una classe per poder gestionar els aspectes gràfics de la renderització de món: iniciar la finestra, gestionar els dos *shaders* principals i gestionar el *framebuffer* secundari. És possible estalviar-nos aquesta classe i posar tot el contingut dins de la classe Joc, però és molt millor destinar aquestes funcions a una sola classe Renderer per poder distribuir millor la feina.

5.7.1. Atributs

- `texturaMon` (`Textura*`): un punter a la Textura que s'ha d'utilitzar per renderitzar els vòxels.
- `shaderActual` (`int`): determina el *shader* que s'està fent servir. Amb la funció `obtShader` podem obtenir un punter al *shader* actual.
- `shaders` (`ShaderProgram[]`): un vector amb els *shaders* principals per renderitzar el món. Es fan servir 2, però l'usuari pot afegir-ne més si fa falta.
- `framebuffer` (`Framebuffer`): el *framebuffer* que farem servir per renderitzar un cub sense que el jugador ho vegi.
- `window` (`GLFWwindow*`): una referència a la finestra creada per GLFW. Té un *getter* (`finestra()`).
- `WIDTH i HEIGHT` (`int`): les dimensions inicials de la finestra. Comencen en 1920*1080, però es pot canviar.

5.7.2. `crearFinestra`

Si recordem al principi del capítol, vam parlar que la classe Joc cridava aquesta funció per crear una finestra i avaluar si s'ha pogut fer o no. Per tant, aquesta funció és la base del motor. Retornarà 1 o -1 si ha succeït en crear la finestra o no. A la Figura 5.19 trobarem el codi de la funció.

```
int Renderer::crearFinestra()
{
    // Iniciem GLFW
    glfwInit();

    // Expressem la versió de GLFW que volem, 3.3
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

    // Creem una nova finestra amb unes dimensions i un nom i que comenci en Windowed mode
    window = glfwCreateWindow(WIDTH, HEIGHT, "MinecraftGL", NULL, NULL);
    Recursos::width = WIDTH;
    Recursos::height = HEIGHT;

    // Si no aconseguim crear-la, terminem el programa
    if (window == NULL)
    {
        cout << "No s'ha pogut crear la finestra" << endl;
        glfwTerminate(); // Termina el context GLFW
        return -1;
    }
    // Fem que la finestra creada es torni el context actual
    glfwMakeContextCurrent(window);

    // Iniciem GLAD
    if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
    {
        cout << "No s'ha pogut iniciar GLAD" << endl;
        glfwTerminate(); // Termina el context GLFW
        return -1;
    }

    // Li diem la mida al viewport, i des d'on comença
    glViewport(0, 0, WIDTH, HEIGHT);

    framebuffer = Framebuffer(WIDTH, HEIGHT);

    centrarFinestra();

    // Podem posar una icona a la finestra
    int iconwidth, iconheight, nrChannels;
    string path = "Textures/icon.png";
    unsigned char* img = stbi_load(path.c_str(), &iconwidth, &iconheight, &nrChannels, 4);
    GLFWimage icon;
    icon.width = iconwidth;
    icon.height = iconheight;
    icon.pixels = img;
    glfwSetWindowIcon(window, 1, &icon);

    return 1;
}
```

Figura 5.19: codi per crear la finestra

Comencem iniciant la llibreria GLFW i especificant la versió. Farem servir la versió 3.3, ja que és la que ens hem descarregat, i li diem que faci servir OpenGL. Després, amb la funció `glfwCreateWindow` podrem crear la finestra i retornar una referència a l'atribut. En aquest moment li podem canviar el nom i fins i tot dir-li si volem pantalla completa o no. En el nostre cas, farem que no. Avaluem si hem pogut crear la finestra mirant si el punter apunta a algun lloc i, si no és així, terminarem el programa retornant un -1. Altrament, utilitzarem la funció `glfwMakeContextCurrent` i així li donarem el control a la finestra creada.

Les línies 27 a 32 serveixen per iniciar OpenGL i GLAD. Fem servir `gladLoadGLLoader` perquè tenim GLFW present. Un altre cop, si veiem que no s'ha pogut fer, terminem i retornem -1.

Després es tracta d'iniciar el *viewport* i el *framebuffer* amb la mida correcta. També cridarem una funció de Renderer anomenada `centrarFinestra`. Per últim, donem la oportunitat a l'usuari de posar una icona personalitzada a la finestra: només s'ha de canviar la que li oferim a `/Textures/icon.png`. Si veiem el que fa el codi, fem servir un altre cop la funció `stbi_load` per carregar la imatge, però en comptes de posar-la en una textura, la col·loquem en una `GLFWimage` i cridem `glfwSetWindowIcon`.

5.7.3. `centrarFinestra`

Aquesta funció centra la finestra al monitor principal. Per això, obtindrem la mida del monitor amb `glfwGetVideoMode(glfwGetPrimaryMonitor())`, la dividirem entre 2 per obtenir el centre i cridarem `glfwSetWindowPos`, que mou la finestra. Si no utilitzéssim aquesta funció, la finestra sortiria en una posició aleatòria.

5.7.4. `obtenirTamany`, `obtenirCentre` i `aspectRatio`

Aquestes funcions serveixen per obtenir informació de la finestra. `obtenirTamany` és el mateix que anar a Recursos i obtenir la mida d'allà, però era la funció que s'utilitzava abans de que existís Recursos. `obtenirCentre` és simplement dividir la mida entre 2 i `aspectRatio` divideix l'amplada entre l'altura. Les funcions no agafen la mida de Recursos, sinó que criden a `glfwGetFramebufferSize`, que també retorna aquesta informació.

5.7.5. `carregaShaders` i `usarShader`

`carregaShaders` carrega els dos shaders principals que es faran servir: `VertexShader`, `FragmentShader`, `VertexPla` i `FragmentPla` (veurem què fan a l'apartat 5.8). La funció només crea els ShaderProgram corresponents i crida el `carregaShaders` de cadascun. També retorna si s'ha pogut fer, ja que és essencial a l'inici del motor, i li dona valor al punter `texturaMon` amb la textura del mapa de textures.

5.7.6. Funcions de *shaders*

Per comunicar-nos amb els *shaders* principals podríem fer servir el ShaderProgram que retorna obtShader o les funcions proporcionades per aquesta classe: `obtenirUniform`, `colocarMat4` i `colocarInt` criden les seves homònimes als *shaders*, mentre que `activaAigua`, `activaBounding` i `activaNit` alternen els uniforms que veurem a `FragmentShader`.

5.7.7. `usarTexturaMon`

Simplement crida la funció `use` de `texturaMon` per poder fer-la servir a l'hora de renderitzar.

5.7.8. `DibuixarDarrera` i `DibuixarFront`

Criden `Unir` i `Desunir` respectivament del *framebuffer*. Tenen aquests noms per tal que sigui fàcil entendre el que fan.

5.8. Chunk

Finalment, després d'aprendre sobre les classes més bàsiques, entrem en matèria de gestionar i visualitzar el món. Al Capítol 4 vam explicar què era un chunk i com facilita la generació d'un món dinàmic. En aquesta classe veurem també en detall com podem renderitzar un vòxel, com gestionar la llum d'un vòxel, la tècnica de *frustum culling*, com s'emplena el chunk segons la informació d'una classe Generador i els *shaders* involucrats en tot el procés.

5.8.1. Atributs

- `X`, `Y`, `Z` (`int`): definits al principi de Chunk.h, especifiquen les dimensions d'un chunk. Y és l'altura, no la profunditat del chunk. Comencen en 16*128*16.
- `_vertices`, `_vertices_transp` (`vector<GLubyte>`): vectors per guardar els vèrtices de vòxels.
- `elements`, `elements_transp` (`int`): quants elements hi ha a cada vector.
- `canviat` (`bool`): informa si el chunk ha estat canviat.
- `descarregant`, `preparat`, `carregat`, `generat` (`bool`): diferents `bools` que especifiquen l'estat del chunk.
- `VBO`, `VBO_TRANSP` (`int`): els ids dels VBO que farem servir
- `posX`, `posY` (`int`): la posició al món del chunk. Té un *getter* (`obtPos`).
- `chunk` (`Cub[]`): una *array* que conté la informació dels vòxels del chunk.
- `veiEsq`, `veiDre`, `veiUp`, `veiBaix` (`Chunk*`): punters als veïns del chunk.

5.8.2. Cub

Un `struct` que defineix com es un vòxel. chunk és un vector que conté `X*Y*Z Cubs`. Conté dos `uint8_t` pel tipus i per la llum del vòxel i un `int` pel color. `uint8_t` sortirà molt sovint: és un tipus d'`int`, sense signe i que només ocupa 8 bits, útil per tenir vèrtexs compactes i estalviar memòria.

5.8.3. Constructor i destructor

Al constructor li hem de passar la posició del chunk, que es guardarà als atributs adients. També fem servir `memset` per inicialitzar tota l'array de Cubs. El destructor s'encarrega de cridar `glDeleteBuffers` i eliminar els buffers creats pels VBO. A més també ha de posar a `NULL` els punters que apunten a ell dels respectius veïns (a `veiUp` ha de posar `veiBaix` a `NULL`, per exemple).

5.8.4. `afegirVeins`

Modifica els atributs veïns amb els que passem per paràmetre i modifica aquests alhora per tal que ells també tinguin una referència a ell mateix.

5.8.5. `canviarCub`

Cada cop que vulguem afegir o eliminar un vòxel, ja sigui per part del jugador o per part del motor, hem de passar per aquesta funció. Com hem vist, els cubs es guarden en una *array*. Eliminar i afegir en una *array* estàtica és una mica tediós, i és molt més senzill i ràpid simplement canviar el tipus d'un cub que anar eliminant-los, ja que `AIRE` representa un vòxel buit de totes maneres.

Aquesta funció rep la posició del cub a canviar i el tipus que es vol posar. A més, es poden passar dos paràmetres opcionals: `reemplacar`, que per defecte és `false`, i `color`, que per defecte és blanc. Si `reemplacar` és a `true`, independentment del que hi hagués abans, es substituirà el bloc pel tipus definit. Altrament, només es substituirà si el que hi havia abans és `AIRE`.

L'atribut `canviat` es posarà a `true`, així com també els `canviat` dels veïns en cas que estiguem a la vora del chunk. Per exemple: si som al bloc (0,0,0), els chunks que hi ha a l'esquerra i a baix (perquè som a la cantonada esquerra inferior del chunk) de l'actual també s'han de posar com canviats. Canviar un cub amb vegetació a sobre eliminarà la vegetació.

5.8.6. `canviarLlumNaturalCub` i `canviarLlumArtificialCub`

Aquestes dues funcions actualitzen la llum d'un vòxel. Per calcular la llum d'un vòxel, veurem que distingim dos tipus de llums: natural i artificial. En l'estat actual del motor, la llum natural no es farà servir, però l'artificial sí. La llum artificial és aquella que el jugador modifica col·locant vòxels que emeten llum (només n'hi ha un: `LLUM`). Per combinar aquests dos tipus de llums, tot i que no fem servir la natural al final, combinarem les dos en un sol `uint8_t`. Per tant, els dos tipus tindran un rang de valors entre 0 i 15, sent 0 foscor total (veure Figura 5.20) i 15 totalment il·luminat.

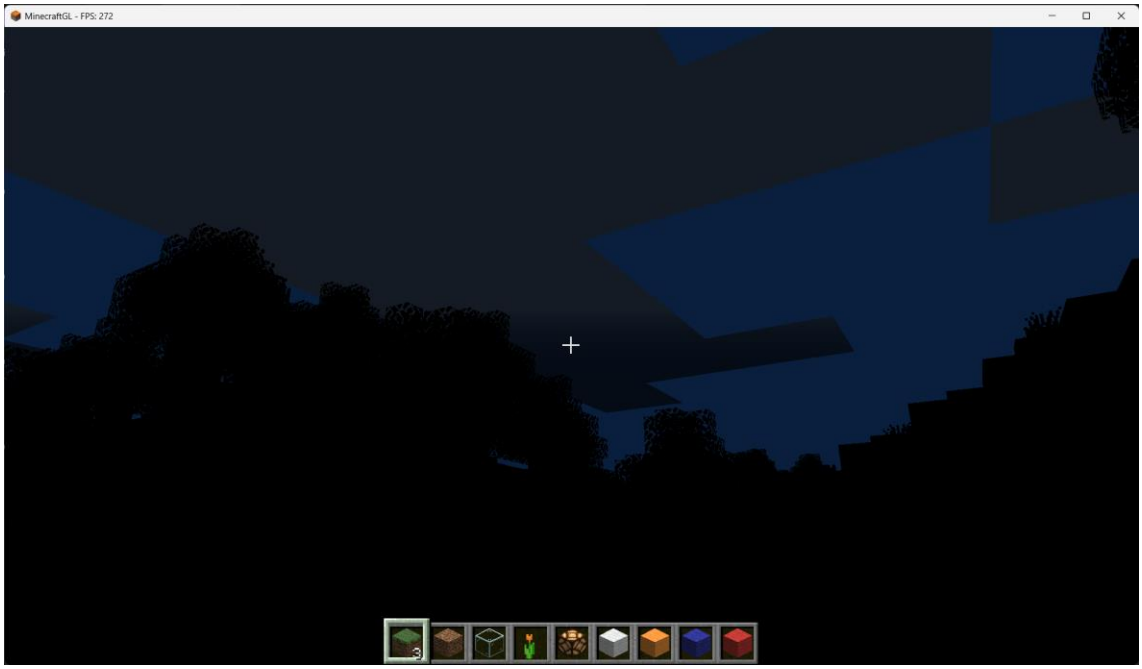


Figura 5.20: el món de nit, on la il·luminació es 0

Combinarem els dos valors canviants els bits del `uint8_t`. Els 4 primers bits (`XXXX0000`) seran llum natural, i els 4 últims (`0000XXXX`), artificial. Per tant, les dues funcions el que fan es canviar aquests bits:

- `(chunk[x][y][z].llum & 0xF0) | llum` – Amb aquesta línia modifiquem els 4 últims bits. Utilitzem una màscara per quedar-nos amb els 4 primers bits (fer una operació `&` amb `0xF0` fa que els 4 primers bits es quedin iguals i els últims desapareguin). Amb la operació `|` actualitzarem els últims bits (llum artificial).

- `(chunk[x][y][z].llum & 0xF) | (llum << 4)` – El mateix que abans, però aquest cop ens quedem amb els 4 últims i actualitzem amb la llum que ens arriba, però li afegim 4 zeros al final. Així només modifiquem els 4 primers bits de la llum que hi havia abans.

Amb aquest mètode, obtenim una manera més compacte de guardar la llum que no amb dos valors diferents. La idea darrere dels vèrtexs és que siguin tan compactes com sigui possible. Veurem que, de fet, fem servir `GLubyte` pels vèrtexs precisament perquè de cada informació només guardarem 8 bits. Si utilitzéssim dos valors diferents, estaríem emprant 16 bits en comptes de 8. Ja que podem fer aquesta optimització, la tindrem en compte.

5.8.7. `obtenirCub`, `obtenirLlumNaturalCub` i `obtenirLlumArtificialCub`

Si les anteriors funcions eren *setters*, aquests són els respectius *getters*. Per la il·luminació només hem de fer les màscares necessàries:

- `chunk[x][y][z].llum & 0xF` – Com hem vist abans, retorna els últims 4 bits.
- `(chunk[x][y][z].llum >> 4) & 0xF` – Amb aquesta màscara retornem els 4 últims bits un cop empesos 4 cops endavant els primer 4 bits.

5.8.8. `esValid`

A les funcions anteriors primer hem de comprovar si la posició que ens envien és vàlida. Per tant, aquesta funció retorna `true` si cada element de la posició que li passem per paràmetre està entre `0` i `X`, `0` i `Y` i `0` i `Z`.

5.8.9. `emplenarChunk`

El primer pas abans de renderitzar el chunk és emplenar-lo. Amb emplenar ens referim a obtenir la informació necessària de terreny per modelar el chunk, és a dir, aquesta funció determina com es veurà el chunk. La funció rep per paràmetre una instància de la classe Generador (Apartat 5.10), que definirà el terreny, i retornarà un vector amb les estructures que s'han col·locat, com els arbres o les flors. A la figura 5.21 tenim el codi de la funció.

```
1  vector<pair<int,glm::vec3>> Chunk::emplenarChunk(const Generador&generador)
2  {
3      vector<pair<int, glm::vec3>> res;
4      const int W = 0, H = 0;
5      for (int i = 0; i < X; i++) {
6          for (int k = 0; k < Z; k++) {
7
8              int height = Y/2;
9              float x = (W + i + X * posX);
10             float y = (H + k + Z * posY);
11
12             if (generador.tipusMon == NORMAL) {
13                 height = generador.obtAltura(x,y);
14             }
15
16             height += 5;
17
18             for (int j = 0; j <= height; j++) {
19                 int tipus = generador.obtTipus(j, height);
20                 if (tipus == GESPA || tipus == NEU) {
21                     float probArbre = (float)(rand()) / (float)(RAND_MAX);
22                     float probFlor = (float)(rand()) / (float)(RAND_MAX);
23
24                     if (probArbre < generador.probabilitatArbre) {
25                         // Marquem l'arbre
26                         res.push_back({ 0, glm::vec3(i + X * posX,j + 1,k + Z * posY) });
27                     }
28                     else if (probFlor < generador.probabilitatFlor) {
29                         // Marquem la flor
30                         res.push_back({ 1, glm::vec3(i + X * posX,j + 1,k + Z * posY) });
31                     }
32
33                 }
34                 canviarCub(i, j, k, tipus, true, Recursos::BLANC);
35
36             }
37             for (int j = 0; j < generador.nivellMar; j++) {
38                 canviarCub(i, j, k, generador.aigua, false, Recursos::AIGUA);
39
40             }
41         }
42     }
43
44     return res;
45
46 }
```

Figura 5.21: codi de la funció emplenarChunk

Per emplenar el chunk hem de recorre tot el requadre format per `X` i `Z` (per defecte 16*16) i donar-li una altura. És a dir, per cada columna al chunk, hem de determinar l'altura. Aquesta vindrà donada per la funció **`obtAltura`**, a la que li hem de passar la posició global de la columna. Per calcular-la només hem de sumar `i` i `k` amb `X` i `Z` multiplicats per la posició del chunk (línies 9 i 10), obtenint així la localització al món (si no, estaríem passant sempre de 0 a `X`). Per exemple, la columna (0,1) del chunk (1,1) és la columna (15,16) al món si no hem modificat `X` o `Z`.

La altura només la obtindrem del generador en cas que el món sigui de tipus `NORMAL`. L'altre tipus és `PLA`, que sempre té la altura determinada per defecte a la línia 8 (l'usuari pot canviar aquest valor per canvia l'altura d'un món `PLA`). Si la altura sempre és la mateixa i no varia, totes les columnes tenen la mateixa, així que el món quedaria pla. La línia 6 és un afegit que es pot treure perfectament, però mostra que encara es pot modificar la altura tant com es vulgui.

El `for` que ve després itera sobre cada bloc de la columna fins a l'altura determinada. El tipus del vòxel que va a cada lloc ens ho donarà un altre cop una funció del generador: `obtTipus`. Li hem de passar la altura abans calculada i la altura del bloc que estem mirant actualment. Amb el tipus obtingut només queda cridar a la funció `canviarCub` que hem explicat abans amb aquest tipus i la posició actual.

Si el tipus és `GESPA` o `NEU`, calculem una probabilitat entre 0 i 1 per veure si podem plantar un arbre o una flor a sobre. En cas que la probabilitat sigui menor que la probabilitat donada pel generador, marcarem l'espai just a sobre del bloc amb l'estructura adient.

Amb un altre for emplenarem d'`AIGUA` fins al nivell del mar determinat, un altre cop, pel generador. Fixem-nos que en aquest cas no reemplacem el vòxel que ja hi hagi col·locat allà, per tant només s'emplenarà d'aigua aquells vòxels que siguin `AIRE`.

5.8.10. `crearVertexs`

Un cop tenim el chunk emplenat, podem començar a definir cadascun dels vèrtexs d'aquest. Els vèrtexs es guardaran en un `vector` de `GLubyte` i, depenent de si es tracta d'un vòxel semitransparent o no, es guardaran en un vector o en un altre. Aquesta funció no farà res si el chunk no té marcat el `bool` de `canviat` o si té marcat `generat` o `descarregant`.

Iterarem per `chunk`, obtenint el tipus de cada vòxel. Si no és `AIRE`, afegirem els vèrtexs del cub a l'`array` corresponent utilitzant la funció `afegirCub` (veure Apartat 5.8.12). Un cop haguem acabat, actualitzem els atributs `elements`, `_vertices`, `elements_transp` i `_vertices_transp` i posem `generat` a `true`.

Si el chunk no tenia marcat `carregat`, avisarà a cadascun dels seus veïns posant els `bools canviat` a `true` i després posarà `carregat` a `true`.

5.8.11. `afegirVertex`

Abans de veure la funció `afegirCub`, definirem com és un vèrtex i en quin ordre té les propietats. En realitat ja ho vam definir a l'Apartat 4.2.5, però en aquesta funció quedarà clar com es fa al codi si s'observa la Figura 5.22.

```
1  // Posicio
2  vertices.push_back(x);
3  vertices.push_back(y);
4  vertices.push_back(z);
5
6  // Quantitat de llum
7  vertices.push_back(llum);
8
9  // Coordenades de textura
10 vertices.push_back(u);
11 vertices.push_back(v);
12
13 // Tipus (posició del mapa de textures)
14 vertices.push_back(tipus%16);
15 vertices.push_back(tipus/16);
16
17 // Costat del cub que representa
18 vertices.push_back(costat);
19
20 // Color
21 glm::vec3* _color = Recursos::obtColor(color);
22 vertices.push_back(_color->r);
23 vertices.push_back(_color->g);
24 vertices.push_back(_color->b);
```

Figura 5.22: codi de la funció afegirVertex

A la funció se li ha de passar per paràmetre el vector que es vulgui emplenar amb el vèrtex. Podríem haver fet una classe o un `struct` que guardés la informació del vèrtex i posar això només al vector, però hem preferit posar la informació directament. L'ordre és molt important pel que veurem a la funció `render` (Apartat 5.8.14). Tot el que afegim al vector amb `push_back` s'ha de passar a la funció. L'ordre queda aclarit pels comentaris al codi (en verd a la figura).

A les línies 14 i 15 podem veure que el tipus no s'envia tal qual, sinó que serveix per obtenir la posició al mapa de textures de la imatge adient pel vèrtex. A la línia 21 podem veure que obtenim el color de la classe Recursos i cadascun dels valors s'afegeix com un `GLubyte` al vector.

Quan acaba la funció, `vertices` té concatenada la informació d'aquest vèrtex.

5.8.14. `afegirCub`

Aquesta funció és la que va cridant a `afegirVertex` per construir un vòxel amb cadascun dels seus vèrtexs. Necessita que arribi el vector `vertices` on posarem cada vèrtex, la posició del vòxel que es vol construir, el tipus i el color.

Primer de tot obtindrem el bloc de Recursos amb el tipus corresponent per saber si és vegetació o no, ja que no es construirà de la mateixa manera un vòxel de vegetació que un normal. A la Figura 5.23 podem veure com un arbust o una flor no es fan de la mateixa manera que, per exemple, el tronc que tenen al costat.

Figura 5.23: arbustos i flors en un terreny

La vegetació es construeix en forma de X. Podria semblar que només es necessiten dos plans, i és veritat, però construirem 4 plans. La idea és que, es miri des d'on es miri, la vegetació tingui la mateixa textura. Per això necessitem un pla per cada lloc, un apuntant a una direcció diferent però tots amb les mateixes UVs (coordenades de textures).

Per fer un pla, hem de fer dos triangles. Si el pla que volem col·locar es troba a `(x,y,z)`, aleshores el primer vèrtex serà el de la cantonada esquerra inferior. Per tant, el primer triangle serà el de l'esquerra. El segon vèrtex a afegir és el de la cantonada dreta inferior, és a dir, `(x+1,y,z)`. Si recordem la Figura 4.7, els vèrtexs han d'anar en ordre contrari a les agulles del rellotge per tal que puguin ser visualitzats on volem. A la Figura 5.24 observem un pla en dos dimensions i els vèrtexs corresponents. Això dona una idea de com hem de sumar cada component per tal que ens quedi bé.

A més, les UVs són els mateixos vèrtexs que es veuen a la figura però intercanviats els de dalt amb els de baix (el `0,0` quedaria a la cantonada esquerra superior i el `1,1` a la dreta inferior).

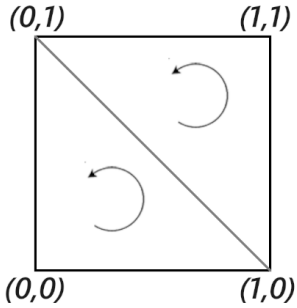


Figura 5.24: construcció d'un pla 2D amb la posició dels vèrtexs definida

No ens capficarem gaire en com construir la vegetació, simplement que es tracten de dos plans (repetits dos cops però girats 180 graus). Si no és vegetació, el vòxel es

construeix de manera normal en aquest ordre: cara esquerra, dreta, frontal, darrera, amunt i a baix. Abans de construir un pla, hem de determinar si aquest és visible. Si es tracta d'un vòxel de `FULLES`, farem que sempre sigui visible. A la Figura 5.25 podem veure la diferència entre fer que siguin sempre visibles o no. Podem veure que les fulles dels arbres són més frondoses si fem aquesta distinció.

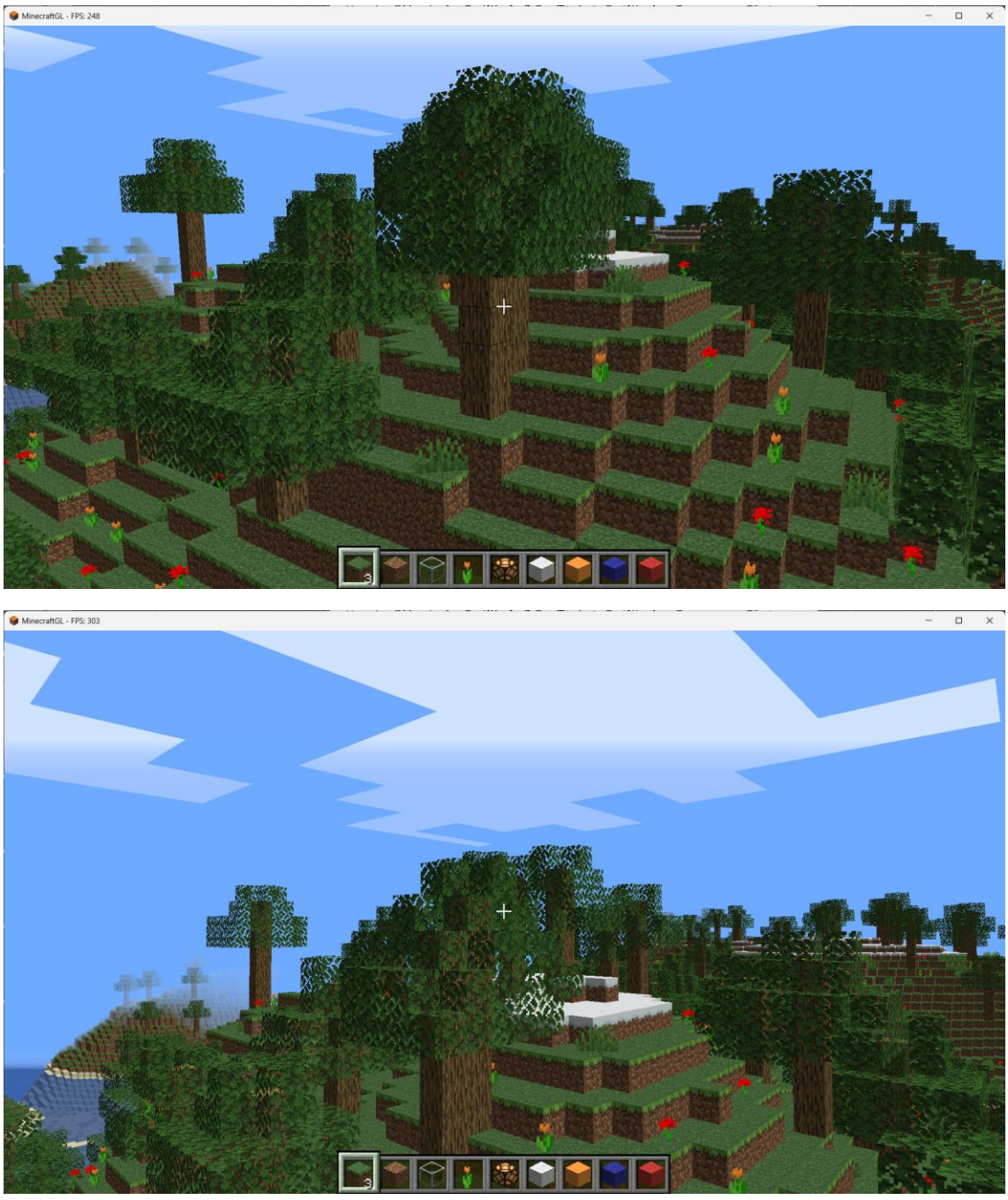


Figura 5.25: a dalt, fent que els vòxels de fulles tinguin totes les cares visibles, a baix, fent que s'ocultin

També hem de detectar si hi ha un cub sòlid just en front de la cara que volem construir. Si no hi ha res o és transparent, podem construir el pla. A més, farem que agafi la llum que té aquell vòxel, ja que **només els vòxels transparents** tindran la llum canviada. Això farà que cada cara tingui una il·luminació diferent. Aquestes condicions s'han de reunir

per cada cara, així que s'han d'avaluar els sis vòxels que envolten el que volem renderitzar.

Si es reuneixen les condicions (és transparent i no és AIRE o AIGUA o no hi ha un vòxel sòlid davant o el que hi ha davant és transparent), es construeix la cara. A la Figura 5.26 podem veure el codi per generar la cara esquerra.

```
1 afegirVertex(vertices, x, y, z, tipus, 0, 1, llum, 1, color);
2 afegirVertex(vertices, x, y, z + 1, tipus, 1, 1, llum, 1, color);
3 afegirVertex(vertices, x, y + 1, z, tipus, 0, 0, llum, 1, color);
4 afegirVertex(vertices, x, y + 1, z, tipus, 0, 0, llum, 1, color);
5 afegirVertex(vertices, x, y, z + 1, tipus, 1, 1, llum, 1, color);
6 afegirVertex(vertices, x, y + 1, z + 1, tipus, 1, 0, llum, 1, color);
```

Figura 5.26: sis vèrtexs diferents composen dos triangles formant un pla

En aquest cas, `y` i `z` segueixen el patró format per la Figura 5.24. La `x` es manté igual perquè l'eix `x` no canvia a l'hora de fer un pla format als eixos `y` i `z`. Si recordem la Figura 5.22, estem enviant la mateixa informació, tot i que no en el mateix ordre: primer posició `(x,y,z)`, després el tipus, que ve donat per paràmetre, les UVs (que, si ens hi fixem, realment són les coordenades de la Figura 5.24 capgirades verticalment), la llum (que ja hem mencionat que ve donada pel cub que hi hagi davant), el costat (cadascun té un enter assignat) i el color, que també ve donat per paràmetre.

Com podem veure, amb 3 vèrtexs formem un triangle i, amb 6, un pla sencer. Si repetim el procés 6 cops per cada cara amb una distància d'1, avaluant les condicions necessàries cada vegada, acabem amb un cub de mida 1. Si un cub té diferents textures per cada cara, abans de la cara amb la textura diferent s'agafarà la variable corresponent del `struct` Bloc (Apartat 5.3.2).

La raó per la qual passem un color és perquè, si ens fixem en la Figura 4.20, la gespa és de color gris. Està amb aquest color per tal que l'usuari pugui canviar el seu color quan vulgui sense que hi hagi interferències de color. Les fulles i els arbustos fan el mateix.

Quan acaba la funció, `vertices` té concatenats tots els vèrtexs que s'hagin pogut posar del vòxel.

5.8.13. `update`

Aquesta funció es cridarà cada cop que s'hagi canviat d'alguna manera el chunk. Si el VBO encara no s'ha iniciat, es generarà ara amb `glGenBuffers`, tant el normal com el `VBO_TRANSP`. Aquest segon és el VBO dedicat a vòxels semitransparents. A la Figura 5.27 trobem el codi de la funció.

```
1  void Chunk::update()
2  {
3      if (VBO == 0) { glGenBuffers(1, &VBO); glGenBuffers(1, &VBO_TRANSP); }
4      if (!generat || elements == 0) return;
5
6      glBindBuffer(GL_ARRAY_BUFFER, VBO_TRANSP);
7      glBufferData(GL_ARRAY_BUFFER, elements_transp, _vertices_transp.data(), GL_STATIC_DRAW);
8
9      glBindBuffer(GL_ARRAY_BUFFER, VBO);
10     glBufferData(GL_ARRAY_BUFFER, elements, _vertices.data(), GL_STATIC_DRAW);
11
12     canviat = false;
13     generat = false;
14 }
```

Figura 5.27: codi de la funció update

A la línia 4 mirem si no hi ha cap element (vèrtex) creat o si no està `generat` (el `bool` que vam posar a `true` a l'Apartat 5.8.10) i, si es així, sortim de la funció. Si no, significa que hem de posar aquests vèrtexs als respectius VBO per tal de ser interpretats per OpenGL a la funció `render` (Apartat 5.8.14). Per això fem servir `glBindBuffer` i `glBufferData`. La primera indica a quin VBO aniran associats els vèrtexs i la segona, els vèrtexs i quants n'hi ha. Això ho hem de fer tant pel VBO de vòxels semitransparents com pel normal.

Al final posarem a `false` els `bools canviat` i `generat`. Com hem pogut comprovar al llarg de tot el codi de la classe, es tracten d'una mena de semàfors que fa que les funcions vagin en ordre d'aparició d'aquest document.

5.8.14. `render`

Finalment, després de les funcions per construir triangles i guardar-los, trobem la funció que dibuixa els vèrtexs que hem generat. A la Figura 5.28 tenim el codi.

```
1  void Chunk::render(bool semi)
2  {
3      if (!preparat || !carregat) return;
4      if (canviat) update();
5
6      unsigned int vbo_bind = VBO, ele = elements;
7      if (semi) { vbo_bind = VBO_TRANSP; ele = elements_transp; }
8
9      glBindBuffer(GL_ARRAY_BUFFER, vbo_bind);
10
11     glVertexAttribPointer(0, 3, GL_UNSIGNED_BYTE, GL_FALSE, 12 * sizeof(GLubyte), (void*)0);
12     glEnableVertexAttribArray(0);
13     glVertexAttribPointer(1, 1, GL_UNSIGNED_BYTE, GL_FALSE, 12 * sizeof(GLubyte), (void*)(3 * sizeof(GLubyte)));
14     glEnableVertexAttribArray(1);
15     glVertexAttribPointer(2, 2, GL_UNSIGNED_BYTE, GL_FALSE, 12 * sizeof(GLubyte), (void*)(4 * sizeof(GLubyte)));
16     glEnableVertexAttribArray(2);
17     glVertexAttribPointer(3, 2, GL_UNSIGNED_BYTE, GL_FALSE, 12 * sizeof(GLubyte), (void*)(6 * sizeof(GLubyte)));
18     glEnableVertexAttribArray(3);
19     glVertexAttribPointer(4, 1, GL_UNSIGNED_BYTE, GL_FALSE, 12 * sizeof(GLubyte), (void*)(8 * sizeof(GLubyte)));
20     glEnableVertexAttribArray(4);
21     glVertexAttribPointer(5, 3, GL_UNSIGNED_BYTE, GL_FALSE, 12 * sizeof(GLubyte), (void*)(9 * sizeof(GLubyte)));
22     glEnableVertexAttribArray(5);
23
24
25     glDrawArrays(GL_TRIANGLES, 0, ele);
26
27     glDisableVertexAttribArray(0);
28     glDisableVertexAttribArray(1);
29     glDisableVertexAttribArray(2);
30     glDisableVertexAttribArray(3);
31     glDisableVertexAttribArray(4);
32     glDisableVertexAttribArray(5);
33 }
```

Figura 5.28: codi de la funció render

Si el chunk no està `preparat` o `carregat`, no podrem renderitzar res. D'altra banda, si ha estat `canviat` per la funció `canviarCub`, hem de cridar `update`. Tenim dues opcions per renderitzar: dibuixar el VBO normal o el dels vòxels semitransparents. El paràmetre semi ens ho dirà. Sigui quin sigui, s'ha de cridar la funció `glBindBuffer` per avisar a OpenGL de quin és l'escollit. Recordem que la funció `update` ha escrit al *buffer* els vèrtexs que hem generat. Ara només hem d'explicar com està organitzat un vèrtex.

Amb `glVertexAttribPointer` especifiquem com estan construïts els vèrtexs. El primer paràmetre és l'identificador de la informació per tal que el *shader* sàpiga on es troba. Si recordem la Figura 4.10, els *shaders* fan ús de `layout (location = n)` per saber aquesta informació, on `n` és aquest primer paràmetre. En el nostre cas, 0 és la posició i es tracta d'un vector de 3 elements, 1 serà la llum, etc. El segon paràmetre és quants valors li estem passant. En el cas de la posició, son 3 valors. Després li hem de dir de quin tipus és la informació, i son bytes sense signe. No ens hem de preocupar per `GL_FALSE`, però sí dels dos següents: el total de la mida del vèrtex i la mida del que ja hem mirat. El total es calcula multiplicant quants valors passem, 12, per la mida d'un `GLubyte`. Després l'últim valor (la mida del que ja hem mirat) sempre és l'últim de l'anterior + quants valors hem afegit (fixem-nos en la línia 13 és `3*sizeof(GLubyte)` i la línia 15 és `4*sizeof(GLubyte)` perquè la llum és només un valor i la posició són 3, 3+1=4).

Un cop hem especificat tota la informació del vèrtex, només queda dibuixar. És tan senzill com cridar `glDrawArrays`, dir-li que uneixi els vèrtexs com triangles (GL_TRIANGLES) i passar-li quants vèrtexs volem. Els `glDisableVertexAttribArray` són un mirall dels `glEnableVertexAttribArray` per netejar-los. Aquests últims sempre tenen l'identificador de la línia anterior.

I així hem renderitzat un chunk sencer, només enviant el VBO amb tots els vèrtexs. Aquesta funció es repetirà moltíssims cops, però com la GPU ja té el VBO, no hem de passar-li la informació dels vèrtexs tota la estona, només quan s'hagi `canviat` el chunk.

5.8.15. *Shaders*

Els chunks no tenen un ShaderProgram cadascun, sinó que el té la classe Renderer. Tot i així, hem d'explicar com són els *shaders* per poder entendre com acabem amb una imatge com la de la Figura 5.29, que utilitzarem com referència per la explicació.

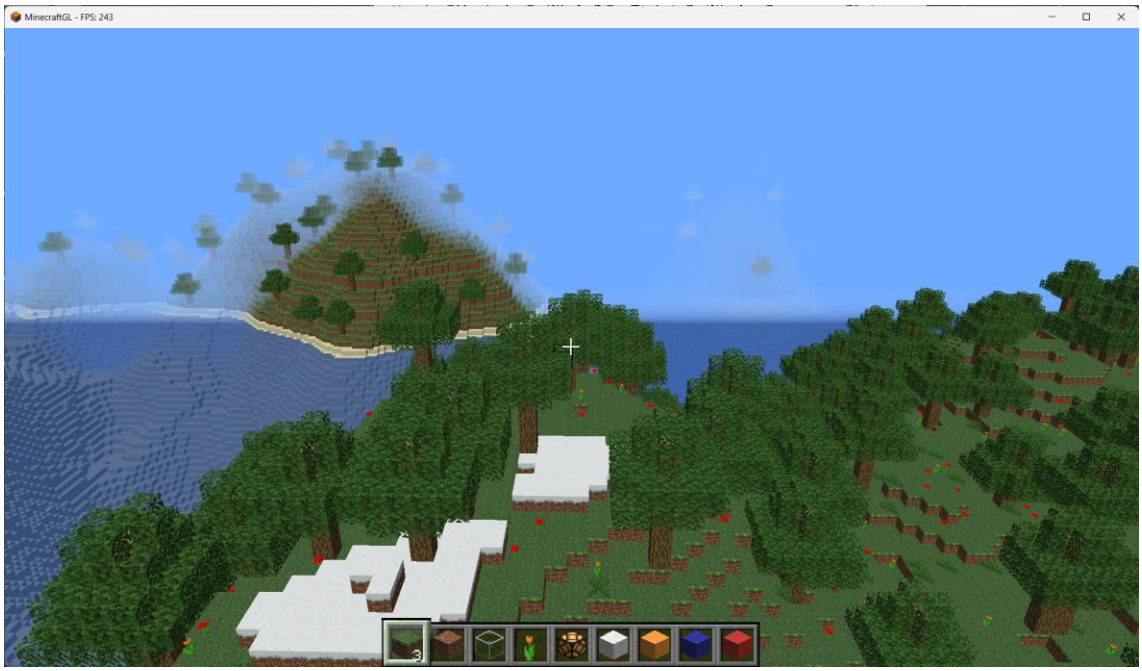


Figura 5.29: renderitzat final d'un conjunt de chunks

El *vertex shader* dels chunks és l'encarregat de col·locar cada vèrtex on toca, per això fan falta les matrius model, *view* i projecció. Encara no hem parlat d'elles, així que és el moment idoni per fer-ho. Per obtenir la posició resultant d'un vèrtex, hem de fer la multiplicació `projecció*view*model*posició`, en aquest ordre. Cadascuna d'aquestes matrius afecten a la posició final d'una manera especial:

- **Model**: és la matriu de transformacions d'una posició. Li podem aplicar una translació (és a dir, moure la posició), una rotació i una escala. A la funció `loop` del Joc (Apartat 5.1.4) trobem una matriu model per defecte inicialitzada com una matriu identitat. Això significa que no volem cap transformació de cap mena. A l'Apartat 5.9 veurem que aplicarem una translació per tal que tots els chunks no acabin al mateix lloc. Multiplicar aquesta matriu fa que les **coordenades del model** es converteixin en **coordenades de món**.
- ***View***: serveix per canviar les **coordenades de món** a **coordenades de càmera**. Si movem la càmera cap a la dreta, essencialment és equivalent a moure el món sencer cap a l'esquerra. En resum: fem que els vèrtexs es defineixin relativament a la càmera, no al món. Aquesta matriu ens la donarà la Càmera (Apartat 5.12).
- **Projecció**: també vindrà donada per la Càmera. Ja vam parlar de que aquesta matriu es modifica cada cop que es canvia la mida de la pantalla, i és perquè converteix les **coordenades de càmera** en **coordenades homogènies** (o el que és

el mateix, **coordenades de pantalla**) tenint en compte la distància de la càmera a cada vèrtex.

No fa falta entendre a la primera aquestes matrius, ja que GLM proporciona dues funcions que les calcularà per nosaltres i les veurem quan visitem la classe Càmera. Tot i així, és important saber què fan: convertir els vèrtexs dels chunks a coordenades de la pantalla. La resta del *vertex shader* es basa en enviar la informació que hem rebut del vèrtex al *fragment shader*.

Aquest sí que és interessant examinar-lo en detall, ja que conté els detalls de com acabem amb una imatge en pantalla. Anirem pas a pas, començant per la Figura 5.30.

```
#version 330 core

out vec4 color;
// Boira
vec4 fogcolor = vec4(0.7, 0.8, 1.0, 1.0);
const float densitat = .000375;
float distanciaFog = 7.5;

// Iluminació
in float llumArtificial;
in float llumNatural;

// Textura
in vec2 TexCoord;
flat in int offsetX;
flat in int offsetY;
float tamanyMapaX = 16.0;
float tamanyMapaY = 16.0;

// Costat
flat in int costat;

// Color del vèrtex
in vec3 colorTint;

//Uniforms
uniform vec3 lightColor;
uniform sampler2D textura;
uniform bool bounding;
uniform bool sotaAigua;
uniform bool nit;
```

Figura 5.30: atributs del fragment shader dels chunks

Són molts d'atributs, però és normal tenint en compte que és el *shader* principal. Veurem en futurs *shaders* que seran molt més curts perquè no estan destinats a una tasca tan important.

En qualsevol cas, organitzem els atributs de la següent manera:

- **Boira**: informació de la boira, com el seu color, la densitat i la distància.
- **Informació del vèrtex:** tot el que li ha arribat al *vertex shader*, però sense la posició. A més, el *vertex* ha sintetitzat la informació: ha separat `llum` en `llumArtificial` i `llumNatural` i enviat tot lo necessari com les UVs (`TexCoord`), el `tipus` separat en `offsetX` i en `offsetY`, el `costat` i el `color`. Els que son enters s'han de especificar amb un `flat` a l'inici de la línia.

- **Uniforms:** recordem que venen des de fora. Hem de rebre la `textura`, que això es fa sol gràcies a OpenGL, si estem dibuixant el contorn d'un vòxel (`bounding`), si som sota l'aigua (`sotaAigua`) i si és de nit (`nit`).

Com el `main` es massa gran, l'anirem explicant per parts. A la Figura 5.31 veurem la primera part de la funció.

```
if(bounding){
    color = vec4(0,0,0,1);
    return;
}

if(sotaAigua){
    distanciaFog = 0.35;
    fogcolor = vec4(0.25,0.3,1.0,1);
}

// Textura
vec2 posTex = vec2(TexCoord.x+(offsetX/tamanyMapaX),TexCoord.y+(offsetY/tamanyMapaY));
vec4 colorText = texture(textura, posTex);

if(colorText.a == 0) discard;
```

Figura 5.31: inici del main del fragment shader

Si `bounding` és `true`, significa que el que estem fent és dibuixar el contorn d'un cub. Volem que aquest color sigui negre, així que igualem color, que és el que retorna el *shader*, a un color negre i retornem.

Si som sota l'aigua, canviarem la distància de la boira per fer-la molt més propera i canviarem el seu color per un to més blau, com es pot veure a la Figura 5.32.

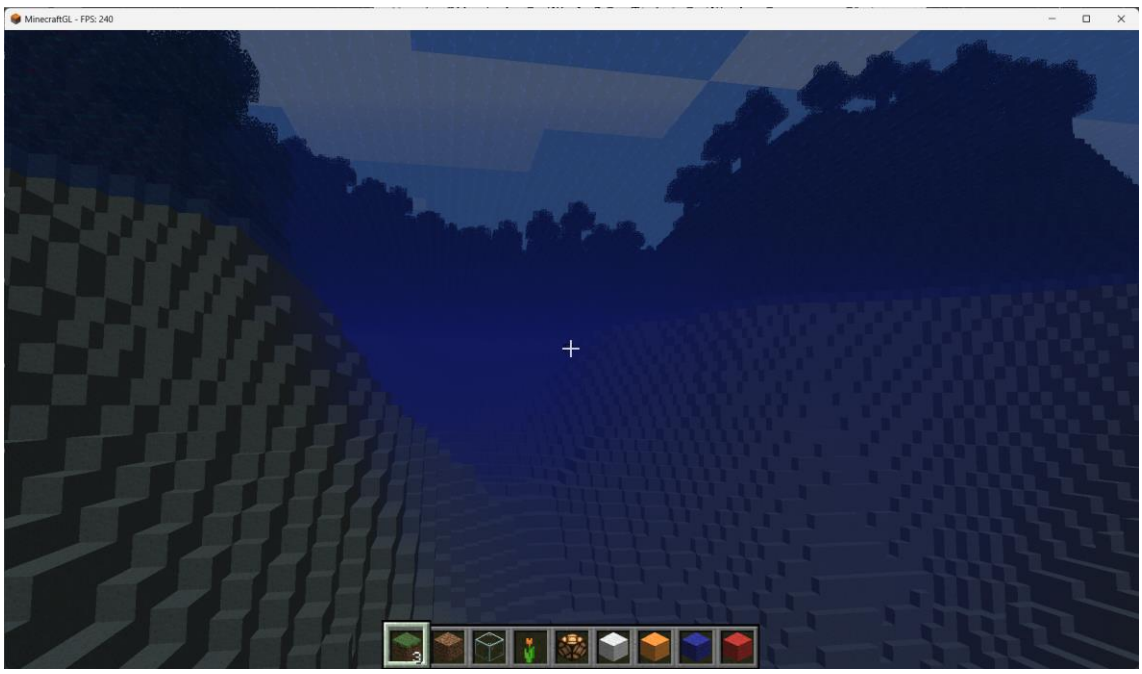


Figura 5.32: la boira sota l'aigua és més propera i blava

Per saber quin color hem de posar a un fragment, hem de saber la posició de la textura. Per això necessitem també com de gran es cada textura dins del mapa de textures. En

el nostre cas són de 16*16 píxels. Simplement agafem les UVs i li sumem els *offsets* dividits pel tamany, és a dir, 16. Després amb la funció `texture` podrem agafar la posició d'aquesta textura i obtenir el color. Si veiem que l'alfa és 0, significa que és transparent i no la necessitem, així que podem **descartar** el fragment.

Després calcularem la boira. És aquest efecte que es veu al fons de la Figura 5.29 i de color blau a la Figura 5.32. Per calcular-la utilitzarem la distància de la pantalla al fragment, la distància de la boira i la densitat utilitzant la funció `exp` (exponencial natural). Això és el que fa la funció `calculaBoira`, juntament amb altres càlculs, a la Figura 5.33.

```
1  // Ens diu el factor que s'ha d'utilitzar per la boira (0: tot boira, 1: sense boira)
2  float calculaBoira(float distancia){
3      float fogFactor = exp(distanciaFog-densitat * distancia * distancia);
4      return clamp(fogFactor, 0.0f, 1.0f);
5  }
6
7  ...
8
9  // Calculem la distancia del fragment a la pantalla
10 float z = gl_FragCoord.z / gl_FragCoord.w;
11 float fog = calculaBoira(z);
12
13 // Si la quantitat és molt petita, no fa falta renderitzar res
14 if(!sotaAigua && fog <= 0.075) discard;
15
16 // Quanta més boira hi hagi, menys es veurà el fragment.
17 float alfa = fog * colorText.a;
18 if(sotaAigua) alfa = colorText.a;
```

Figura 5.33: codi per calcular boira

La quantitat de boira es guarda a `fog`. Si `fog` s'apropa a 1, significa que és la part més propera al jugador. Tot allò que estigui molt allunyat no serveix, així que es descarta. A més, per tal que la boira no es talli en mig del no res, multiplicarem l'alfa que havíem obtingut amb la boira, formant un gradient amb el fons molt més suau (comparar Figura 5.29 i Figura 5.34).

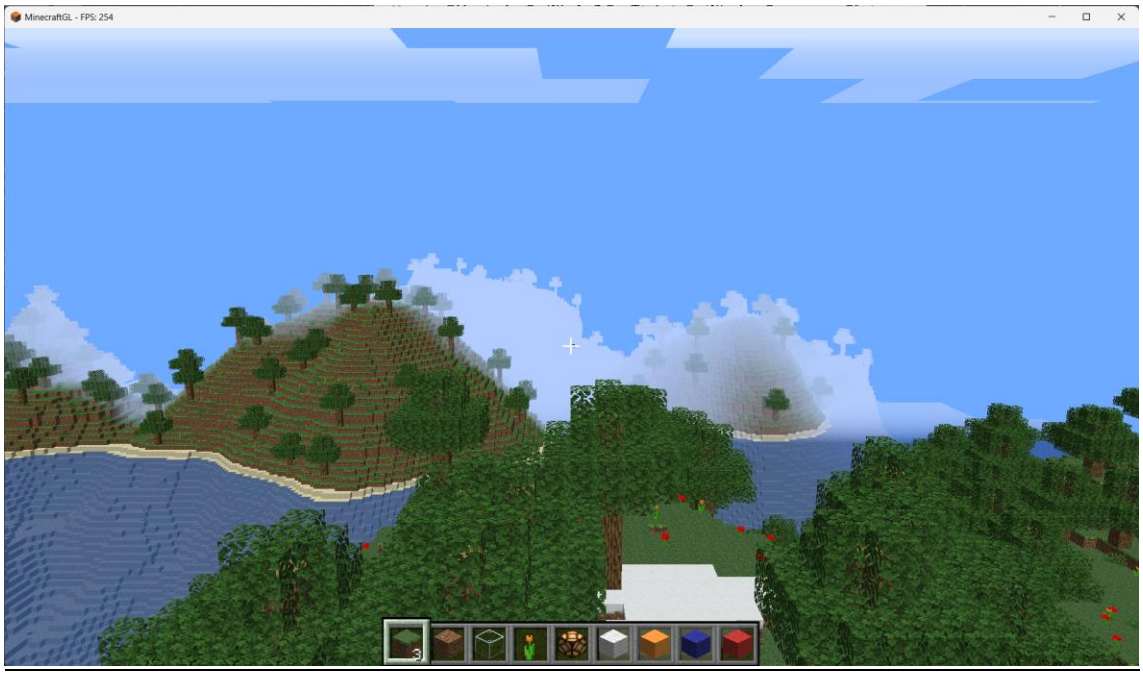


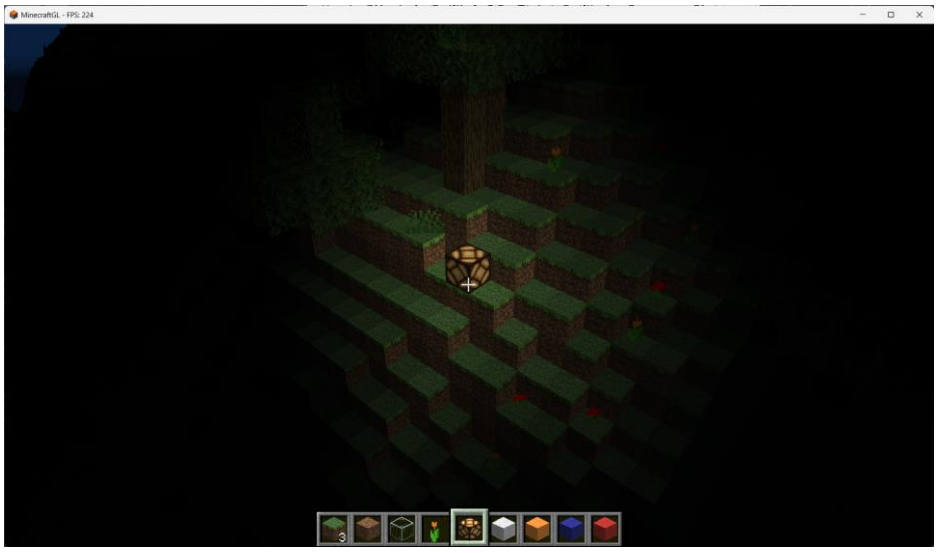
Figura 5.34: efecte de boira sense afectar a la transparència

Tots aquests detalls es poden modificar si a l'usuari li agrada més una manera que altra. No hi ha manera millor, només una predeterminada. A continuació el que es fa es calcular la il·luminació. Aquí juga un paper important la llum artificial, que tindrà un color diferent a la natural (que és gairebé blanca). A la Figura 5.35 trobarem el codi final del *shader*.

```
// Ambient, per tal que la foscor no sigui tan fosca
float ambientStrength = 0.1;
vec3 ambient = ambientStrength * lightColor;

// Iluminació
float gamma = 0.95;

float resArtificial = pow(llumArtificial / 15f, gamma);

float intensitatNatural = 0.9;
float resNatural = 15;

if(nit){
    resNatural = 0;
    fogcolor *= 0.01;
}

vec3 colorLlum = vec3(0.98,0.98,0.98);
if(resArtificial > resNatural) colorLlum = vec3(1.0,1.0,0.87);

float suma = resArtificial+resNatural;
if(suma > 1) suma = 1;
vec3 LlumFinal = colorLlum*suma*0.9;

vec4 brillantor = vec4(0.99,0.99,0.99,1);
vec4 ombres = vec4(1.0,0.8,0.45,0.2);

color = vec4( (ambient + LlumFinal) * colorText.xyz * ombres[costat] * colorTint, colorText.w)*brillantor;

color = mix(fogcolor, color, fog);
color.a = alfa;

// Si som a l'aigua, fem que els blocs tinguin un color blau
if(sotaAigua) color *= vec4(0.25,0.3,0.375,1);
```

Figura 5.35: part final, on es calcula la il·luminació

L'*ambient* és un valor que fa que la foscor no sigui tan fosca. Quan tinguem il·luminació 0 podrem distingir un vòxel d'un altre gràcies a això. Després tenim la `gamma` i la `llumArtificial`. Dividim aquesta última entre 15 ja que és el valor màxim que pot obtenir, i l'elevarem a `gamma`. Depenent d'aquest valor, obtindrem resultats molt diferents en quant a la suavitat de la il·luminació. A la Figura 5.36 podem veure el resultat, sobretot on es difumina la llum, d'un valor gamma d'1.5, 0.9 i 0.3.



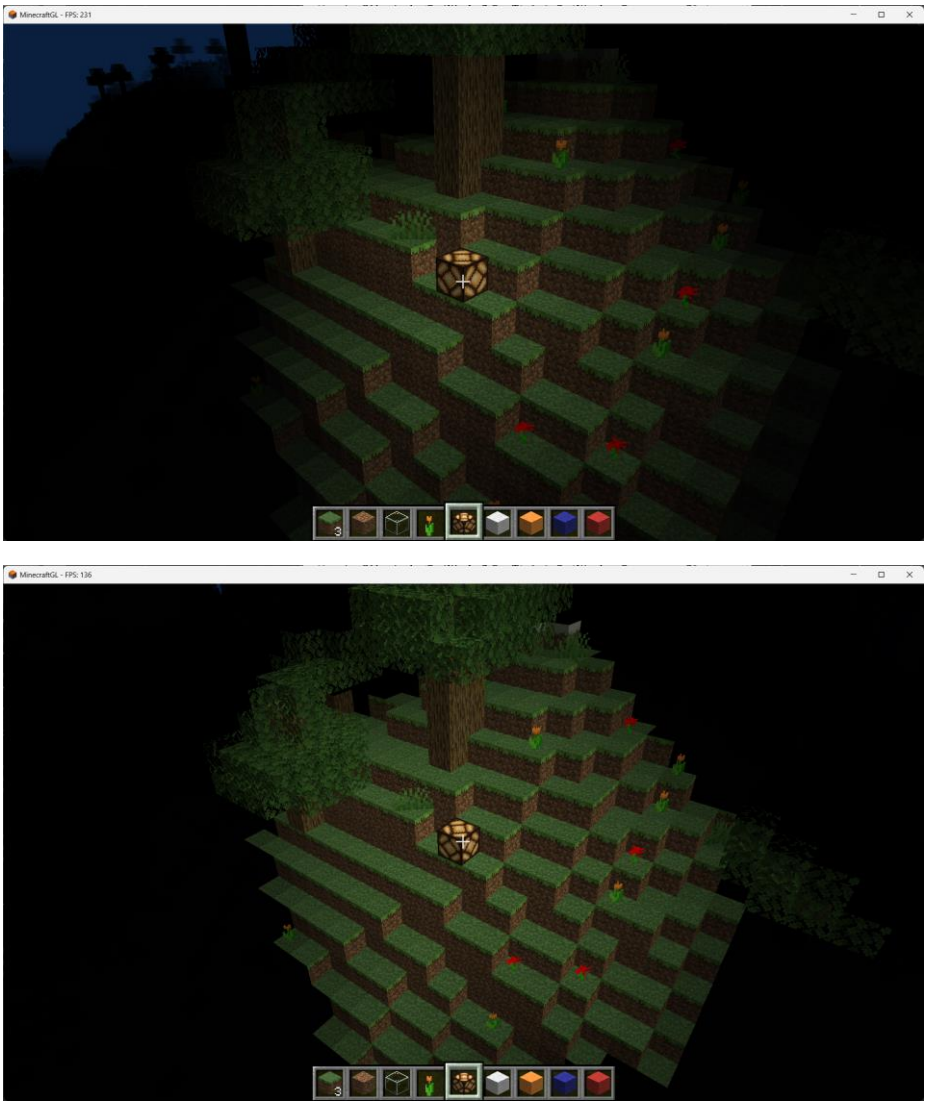


Figura 5.36: diferents captures de diferents valors de gamma: 1.5, 0.9 i 0.3 (de dalt a baix)

Si és de nit, disminuirem molt la il·luminació natural i el color de la boira. També detectarem quina llum li arriba més a un vòxel. Si és llum artificial, obtindrà un color amb un to més groc. La llum final és la suma d'aquestes, però no sobrepassant 1, atenuada un 90%. També tenim un vector brillantor per pujar o baixar la intensitat de la il·luminació final.

El vector ombres farà que un mateix vòxel, depenent del seu costat, estigui il·luminat de manera diferent per cada cara. La cara de dalt obté un 100% de llum, les de esquerra i dreta un 80%, la de davant i darrera un 45% i la de sota un 20%. Així un vòxel donarà la sensació d'ombres. Sense això, el món seria molt pla (comparar Figura 5.29 i 5.37).

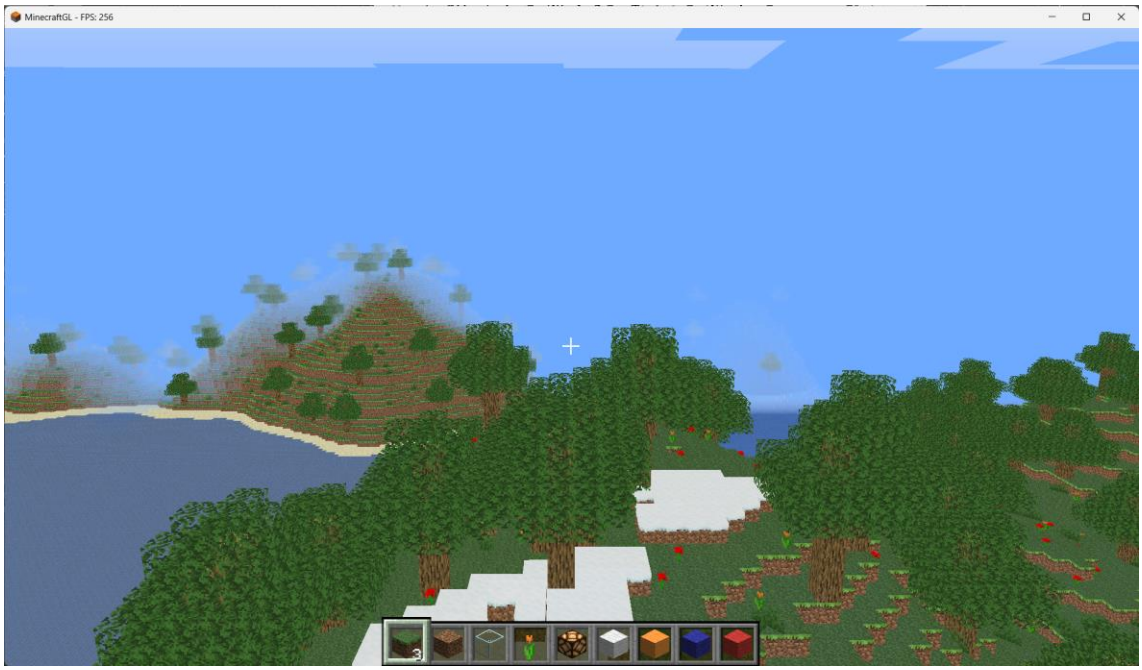


Figura 5.37: el món sense ombres. Gairebé no es distingeix la profunditat del terreny.

El resultat final de color és la suma de l'ambient i la llum final, multiplicat pel color obtingut de la textura, per la ombra i pel color del vòxel. Tot això multiplicat per la brillantor. Finalment deixem que `fog` decideixi si agafar aquest color o el color de la boira o un entre mig fent servir la funció `mix`. Si som sota l'aigua, multiplicarem el color final per un color més blavós.

Així, finalment, obtenim la imatge de la Figura 5.29 després de passar pels dos *shaders*. Aquest és el *shader* més complicat en comparació.

5.8.16. `renderCub`, `afegirCubFlat` i `afegirVertexFlat`

Quan volem obtenir el costat d'un vòxel (veure Apartat 5.1.15), hem de renderitzar un sol cub de manera que els costats tinguin un color diferent cadascun. En comptes de fer servir el render normal, utilitzarem un amb un VBO especial que només té dos atributs: posició i color.

Els *shaders* són molt més senzills, ja que només hi ha posició i color. El *vertex shader* de fet és molt semblant al normal, però sense passar tanta informació, i el *fragment shader* és molt més curt, com es pot veure a la Figura 5.38. No ens fixarem en aquestes funcions, ja que són versions molt reduïdes de les seves germanes grans.

```
#version 330 core

out vec4 color;

in vec3 vertexColor;

void main()
{
    color = vec4(vertexColor, 1.0);
}
```

Figura 5.38: fragment shader pla

5.8.17. `esVisible`

La funció final de chunk determinarà si aquest és visible pel jugador o no. La tècnica que fem servir per aconseguir-ho es diu *frustum culling*, i veurem que té un augment de rendiment enorme.

Quan renderitzem els chunks, no tenim en compte si estan darrera del jugador o en alguna zona que no pot veure. Simplement els dibuixem i anem al següent. Tot i que el jugador no els pugui veure, aquests chunks s'estan renderitzant, gastant molts recursos en uns chunks bastant grans que ni es poden veure. Per tant, el que farem serà calcular quins estan a dins del rang de la càmera. Aquest rang s'anomena *frustum*, i està composat pels plans que formen la càmera: el *near* i el *far*. A la Figura 5.39 podem veure que les estrelles amb color, que són les que cauen justament dins del *frustum*, es renderitzen i les estrelles en negre, no.

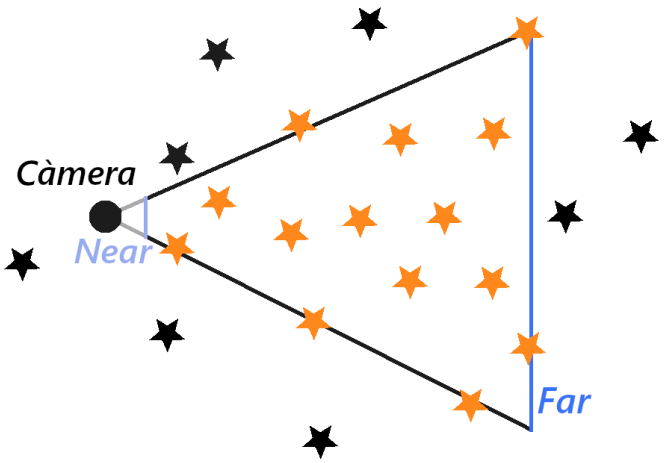


Figura 5.39: representació gràfica de frustum culling. Les estrelles en negre no es renderitzen.

Podem veure que el *frustum* és un prisma format per 6 plans. De calcular-lo s'encarregarà la càmera, mentre que el chunk mirarà si ell mateix està dins d'aquests. Calculem el centre del chunk i la distància respecte a cada pla. Si aquesta distància és menor que un umbral, direm que el chunk no és visible. Això s'ha de fer per cada pla i, si determinem que en al menys un no és visible, aleshores descartarem el chunk i no el renderitzarem.

Una possible implementació del que s'ha explicat és la que trobem a la Figura 5.40.

```
bool Chunk::esVisible(Frustum* frustum) const
{
    glm::vec3 min = { posX * X, -Y/1.5, posY * Z }, max = {posX*X + X, Y*1.75, posY * Z + Z};
    glm::vec3 centre = (min + max) * 0.5f;
    glm::vec3 extents = (max - min) * 0.5f;

    for (const Pla& pla : frustum->obtPlans()) {

        float distancia = glm::dot(centre - pla.pos, pla.normal);

        if (distancia < -glm::length(extents) / 4) return false;
    }

    return true;
}
```

Figura 5.40: codi de frustum culling

Per demostrar que la funció realment aplica aquesta tècnica, desactivem temporalment l'actualització del *frustum* de la càmera, pausant el *frustum culling*. El resultat és el que es veu a la Figura 5.41. El jugador es troba al punt taronja, i els chunks que pot veure son els que forma el prisma de la càmera. Els que té darrera no es renderitzen, augmentant el rendiment.

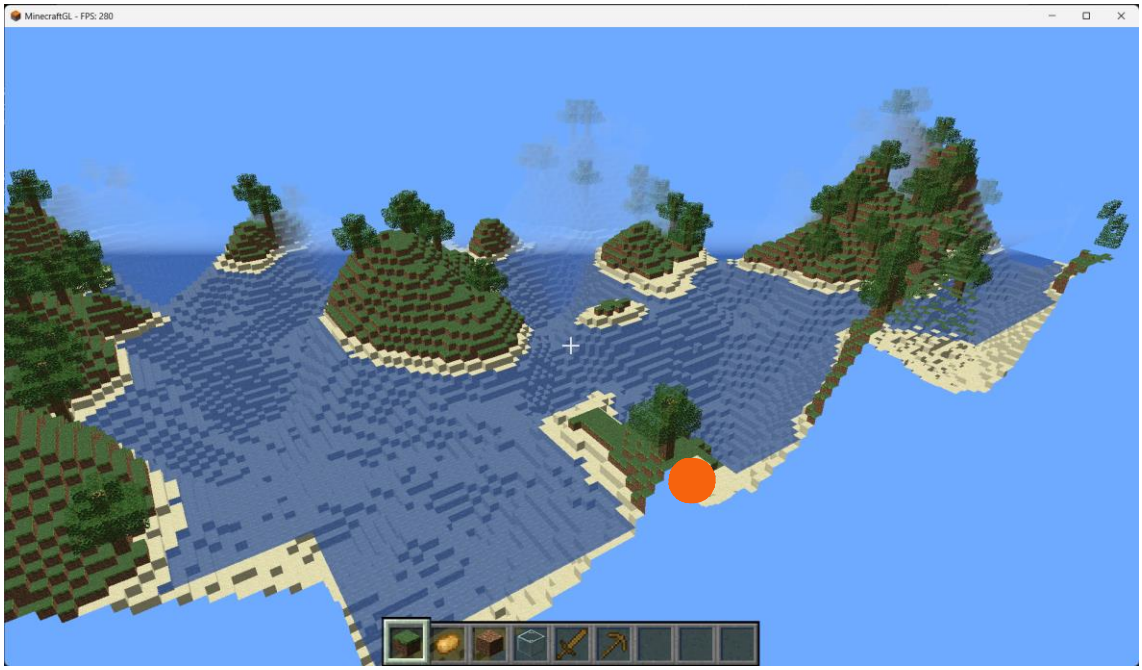


Figura 5.41: demostració del frustum culling

Es pot verificar mirant al cel (ja que no hi ha cap chunk renderitzat): els FPS pugen moltíssim. S'ha incorporat una tecla per poder activar i desactivar el *frustum culling* i, d'aquesta manera, demostrar la eficàcia de la tècnica. A la Figura 5.42 podem veure aquesta millora, que és d'un 105%.

Figura 5.42: d'esquerra a dreta, FPS mirant al cel amb frustum culling, sense fc, i mirant al front amb fc

5.9. SuperChunk

Ara que sabem com funciona i com es renderitza un chunk, passem a veure la classe que els gestiona per tal de tenir un món dinàmic: la classe SuperChunk. El nom prové de l'inici del projecte, quan el món era estàtic i era simplement una col·lecció de chunks, semblant un únic chunk gegant.

5.9.1. Atributs

- `DISTANCIA`: el radi de chunks que poden haver-hi renderitzats al mateix cop. Quant més pugi aquest valor, menor serà el rendiment però major el camp visual.
- `NCHUNKS`: quants chunks es poden processar per una funció en un sol frame.
- `activaFrustum` (`bool`): si `true`, es farà servir *frustum culling*.
- `Chunks` (`map<vec2,Chunk*>`): `map` on es guarden els chunks carregats.
- `cuaLlums` (`queue<vec3>`): una cua auxiliar per quan calculem la il·luminació.
- `posicions` (`vec3[]`): una *array* que es guarda on poden estar els vòxels colindants, és a dir, `(-1,0,0)`, `(0,-1,0)`, etc.
- `descarregarMutex` i `cuaMutex` (`recursive_mutex`): uns semàfors per poder gestionar bé els chunks en un altre fil.
- `chunksCarregar` (`deque<vec2>`): una cua que es guarda els chunks que s'han de carregar (crear).
- `chunksDescarregar` (`deque<Chunk*>`): el mateix, però els que s'han de descarregar (eliminar).
- `blocsNoPosats` i `blocColocats` (`map<vec2,vector<vec3,uint8_t>>`): es guarden, respectivament, els blocs que el motor no ha pogut col·locar per algun motiu i tots els blocs modificats pel jugador.
- `generador` (`Generador`): una instància de la classe Generador.
- `VAO` (`int`): l'id del VAO.

5.9.2. Constructor

El constructor ha de rebre el Renderer per tal que la classe pugui tenir una referència. A més, generarà el *buffer* del VAO. El destructor eliminarà tots els punters que hi hagi a `Chunks` i eliminarà el VAO.

5.9.3. `canviarCub`

Aquesta funció és l'equivalent al `canviarCub` de Chunk, amb la diferència de que aquesta es crida primer i després crida a la de Chunk. Ha de rebre per paràmetre la posició del vòxel que volem canviar, el tipus, si volem reemplaçar el contingut, si és un bloc col·locat pel jugador i el color.

Primer mirarem si es tracta d'una posició vàlida amb la funció `esValid`. Si no és vàlida, s'afegirà la posició i el tipus al `map` `blocsNoPosats`, on la clau serà la posició del chunk on es troba. Si és vàlida, trobarem el chunk que hem de modificar gràcies a la funció `BlocChunk` i la coordenada exacta del chunk amb `Mon2Chunk`. Veurem que hem de transformar les coordenades globals `(16,0,16)` a coordenades de chunk `(1,0,1)`. Després només hem de cridar el `canviarCub` del chunk.

Si es tracta d'un cub col·locat pel jugador, l'afegirem al `map` `blocsColocats`. Això servirà per donar un nivell de permanència, tot i que la idea seria fer servir un arxiu per guardar aquesta informació. De moment guardarem els blocs al `map` i els obtindrem més tard quan creem el chunk.

A més, si el vòxel posat és `LLUM`, hem de calcular la llum al voltant d'aquest (Apartat 5.9.14).

5.9.4. `esValid` i `esCarregat`

Una posició és vàlida si el chunk en el que es troba està carregat. Per això farem servir la funció `BlocChunk` i `esCarregat` per trobar la posició del chunk i saber si està carregat respectivament.

La funció `esCarregat` simplement avalua si el chunk es troba al `map` `Chunks`. Si ho troba, retorna `true` sempre que el punter no sigui `NULL`, que no s'estigui `descarregant` i que s'hagi `preparat`.

5.9.5. `BlocChunk`

Rep una posició, però només la `x` i la `z`. La `y` no interessa que es proporcioni, ja que el món està composat per chunks disposats en una quadrícula i no hi ha chunks per sobre d'altres. Retorna la posició del chunk on es troba el vòxel. Per exemple, si volem saber on es troba el vòxel `(32,0,16)` i el tamany d'un chunk és de 16*16, el vòxel es troba al chunk (32/16,16/16) = `(2,1)`.

5.9.6. `Mon2Chunk`

Rep la coordenada d'una posició i la mida del chunk en aquella coordenada i retorna la posició dintre del chunk del vòxel en aquella coordenada. Tenint el mateix vòxel d'abans en coordenades de món `(32,0,16)`, hauríem de cridar `Mon2Chunk(32,16)` i `Mon2Chunk(16,16)`. El que fa la funció és, si el primer valor es `n` i el segon `m`, `n%m`. Així obtindríem que el vòxel es troba al chunk `(2,1)`, a la coordenada local de chunk `(0,0)`.

5.9.7. `canviarLlumNaturalCub` i `canviarLlumArtificialCub`

Aquestes funcions criden a les seves homònimes dins del chunk i li canvien la llum. S'han de passar la posició global del bloc i la llum, i la funció comprovarà si és una posició vàlida, el chunk on es troba i la seva localització dins d'aquest.

5.9.8. `obtenirCub`, `obtenirLlumNaturalCub` i `obtenirLlumArtificialCub`

Com les funcions anteriors, comprovarà la validesa de la posició i obtindrà el chunk i les coordenades locals. Després només es crida la funció equivalent al chunk i es retornarà el resultat.

5.9.9. `emplenar` i `emplenarArea`

Serveixen per poder manipular quantitats més grans de vòxels. Són útils per no haver d'estar cridant `canviarCub` constantment. Cadascuna fa una forma diferent: `emplenar` emplena un rectangle centrat en una posició i estenent-se en una amplitud i una llargada i `emplenaArea` ho fa amb un prisma, donades dues coordenades: origen i destí. A

ambdues es pot especificar el tipus a emplenar, si es vol reemplaçar el que hi havia abans o no i un color. A més, a `emplenar` es pot passar un paràmetre probabilitat. La funció generarà un nombre aleatori per cada cantonada i, si és menor que aquesta probabilitat, la eliminarà. Veurem que serà útil per donar varietat als arbres. A la Figura 5.43 podem veure dos exemples de figures creades amb les funcions.

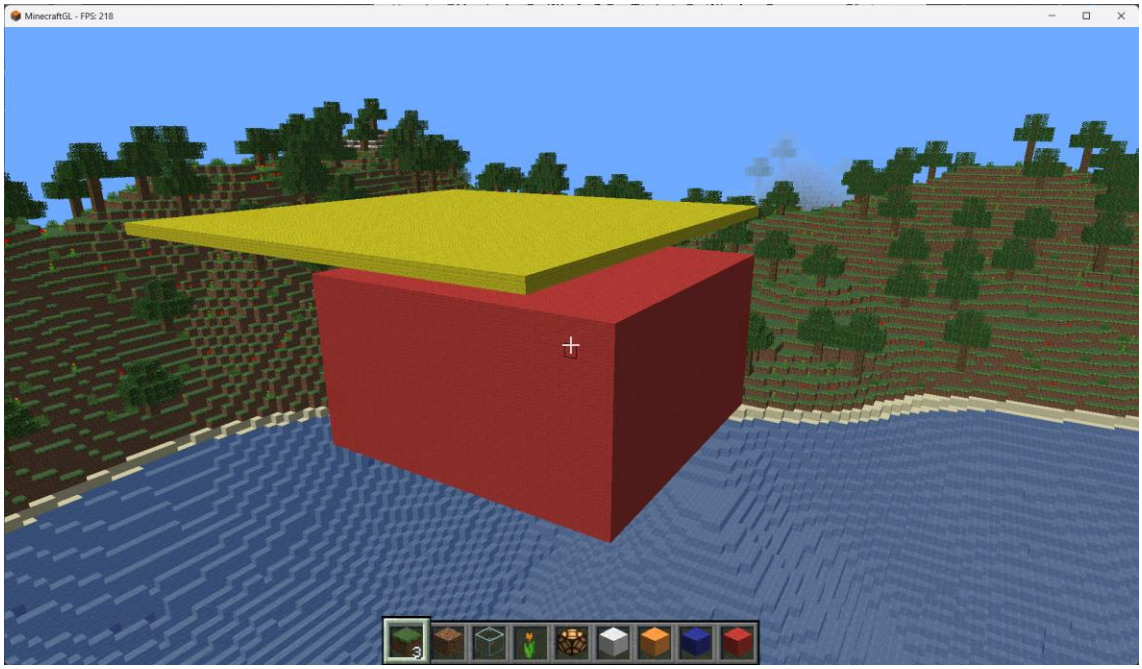


Figura 5.43: rectangle i prisma creats per emplenar i emplenarArea

5.9.10. `arbre`

La funció genera un arbre en la posició indicada. El tronc és d'una altura aleatòria (tot i que sempre hi ha un mínim de 4 blocs de fusta), i les fulles es generen sempre seguint un patró: dos rectangles de 5*5 per fer una base i dos de 3*3 per la copa de l'arbre. Utilitzant la funció emplenar explicada a l'apartat anterior podem fer que els arbres tinguin més varietat: alguns tindran cantonades on faltaran fulles, i aquestes cantonades no sempre seran les mateixes en tots els arbres.

A la Figura 5.44 podem veure un grup d'arbres. El que està al davant és petit, mentre que els dels costats són més alts, donant molta més varietat al món. Fixem-nos que al de davant li falta alguna cantonada i al de l'esquerra li falta però només a la copa.

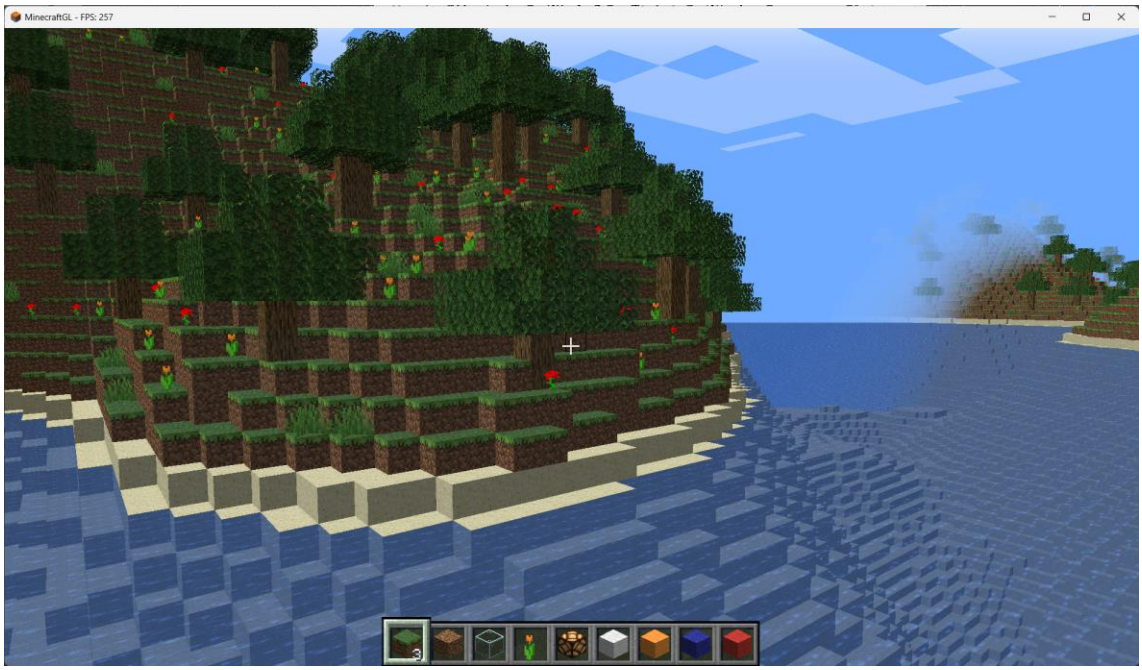


Figura 5.44: diferents arbres amb altures variants

5.9.11. `existeixCub`

Avalua al voltant d'un vòxel si existeix un tipus sense tenir en compte el propi vòxel.

5.9.12. `obtenirColindants`

Retorna un vector amb la posició i els tipus dels cubs al voltant d'una posició. Podem decidir si volem tots els vòxels colindants, si només els sòlids o només els transparents. També podem incloure el vòxel que hi ha a la posició o no.

5.9.13. `obtenirAABB`

AABB és una mena de *bounding box*, és a dir, la caixa més petita que podem construir que pot emmagatzemar un vòxel. Utilitzarem les AABB per detectar les col·lisions del jugador. En aquesta funció retornem un vector amb totes les AABB al voltant d'una posició, aprofitant la funció anterior.

5.9.14. `afegirLlum` i `posarLlum`

Cada vegada que volem afegir un cub que emet llum hem de cridar a la funció `afegirLlum`, com vam veure a l'Apartat 5.9.3. Aquesta s'encarrega de mirar al voltant de la llum i actualitzar el valor de la resta de vòxels que l'envolten. Veurem que aquesta llum és artificial, i per tant cridarem a la funció `canviarLlumArtificialCub`.

En col·locar una llum, es posa a la cua `cuaLlum`. L'algorisme és senzill:

1. Obtenim el primer vòxel de la cua i el traiem d'aquesta.
2. Iterem al voltant d'aquest vòxel i obtenim els que té als costats, a sobre i a sota.
3. Si el vòxel iterat és transparent i la llum d'aquest és menor en almenys dos unitats que la del vòxel de la cua:
    a. L'afegim al final de la cua.

b. Llum del vòxel mirat = llum de la cua – 1. Així evitem que es propagui infinitament.

El pas 3 és el que es troba a la funció `posarLlum`.

Amb aquest algorisme aconseguim que la llum es calculi només un cop: cada vegada que es col·loca un vòxel `LLUM`. Després no ens hem de preocupar per la llum, ja que és estàtica i sempre serà la mateixa. Això s'anomena fer un *baking* de la llum, i l'avantatge principal és que no hem d'estar constantment calculant la seva incidència a la resta de vòxels.

A la Figura 5.45 podem observar l'algorisme en funcionament. Podem notar també que, tot i que estigui envoltat de blocs transparents, la llum segueix propagant-se. A la Figura 5.46 trobem que la llum no passa per una paret que hem col·locat just davant.

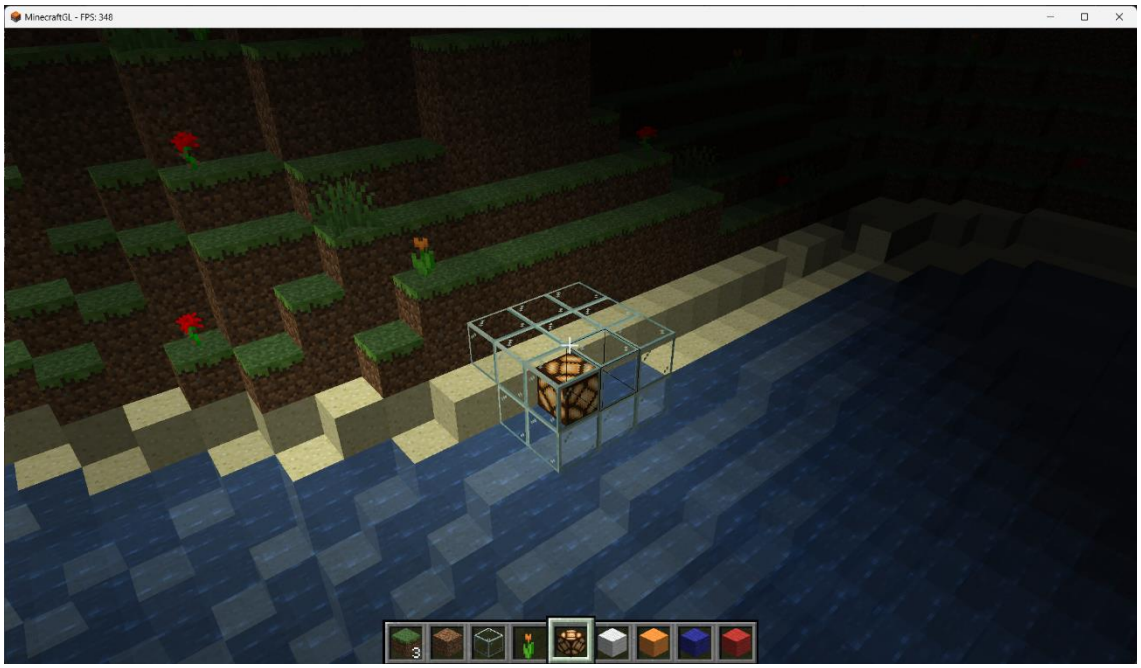


Figura 5.45: un bloc de llum il·luminant de nit una platja. Al voltant té cristall.

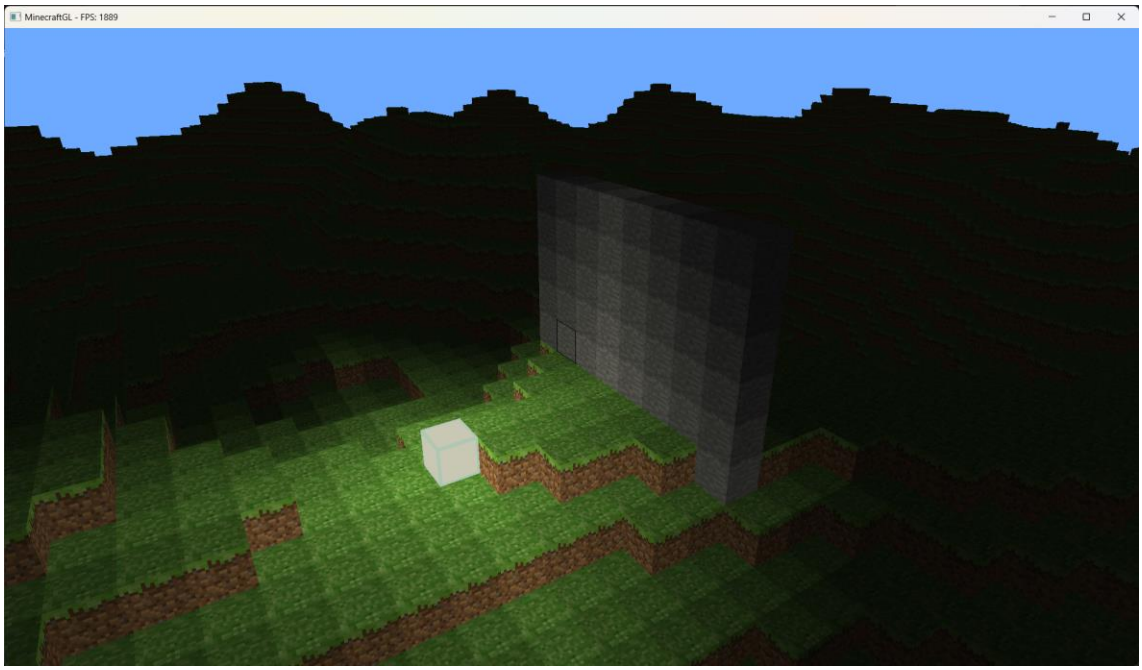


Figura 5.46: una paret bloquejant el pas de la llum

Tot i que hi ha un equivalent per calcular la llum natural, només funciona en un món estàtic i no es parlarà d'aquesta al document. La implementació es troba al projecte, però, i es pot habilitar canviant la *flag* `DEBUG`. També hi ha una implementació per treure una llum, entre d'altres, però es va descartar la seva continuació a favor de continuar altres aspectes molt més importants, com la gestió del món dinàmic.

5.9.15. `generarChunk`

Abans de parlar de com el SuperChunk gestiona aquest món que va canviant al voltant del jugador, hem de veure com es genera un chunk. Si bé és veritat que vam veure a la classe Chunk com es construïa amb la funció `emplenarChunk`, hem de tenir en compte que s'ha de cridar en algun lloc. Aquest lloc és la funció `generarChunk`.

La primera cosa que hem de fer és crear un punter a un chunk nou: `Chunk* nou = new Chunk(pos.x, pos.y)`. Si recordem el constructor d'un Chunk, necessita que especifiquem una posició. Aquesta vindrà donada per paràmetre. Després hem de veure quins són els seus veïns. Per això mirarem els 4 vòxels que hi ha al voltant i, **si estan carregats**, cridarem a `afegirVeins` i passarem els punters corresponents.

Després hem de cridar la funció `emplenarChunk` i guardar-nos les estructures que ha retornat. Amb això fem que es construeixi el chunk. Per cada estructura hem de mirar si es tracta d'una flor o un arbre. Si és una flor, la podem col·locar cridant canviarCub i, si és un arbre, amb la funció `arbre`.

Una possible optimització, si s'implementés el sistema de permanència de chunks en fitxers, és la de guardar-nos el chunk un cop s'hagi generat. Així no s'hauria de generar un altre cop mirant els mapes de soroll, sinó que s'agafaria directament del fitxer i es muntaria a partir de la informació que hi hagi. De moment pel projecte actual ja va bé fer-ho de la manera implementada ja que els chunks tampoc estan tan decorats.

Ara podem posar el `bool preparat` del chunk a `true` i l'afegim al `map Chunks`. Ara és el torn dels `maps blocsNoPosats` i `blocsColocats`. Si trobem que la posició del chunk és una clau als `maps`, aleshores hem d'iterar i cridar la funció `canviarCub` per col·locar els blocs que no es van poder posar en aquest chunk (principalment i molt probablement perquè no estaria carregat) i blocs que va col·locar el jugador en algun moment. Això últim és molt útil, ja que, en cas que el chunk es descarregui en un futur, els blocs tornaran a aparèixer i donarà la sensació al jugador de que mai es van eliminar.

Pot semblar que aquest pas no és important, però a la Figura 5.47 hem diferenciat per color les fulles dels arbres, que són els més afectats per blocs invàlids. Les fulles verdes són les que es van poder construir quan la funció arbre es va cridar. Les blanques són fulles que per algun motiu, com que el chunk encara no s'havia carregat o potser ni existia, no es van poder construir en aquell moment i s'han construït al final d'aquesta funció. Si no ho haguéssim fet, els arbres quedarien podats i els hi faltaria les fulles blanques.

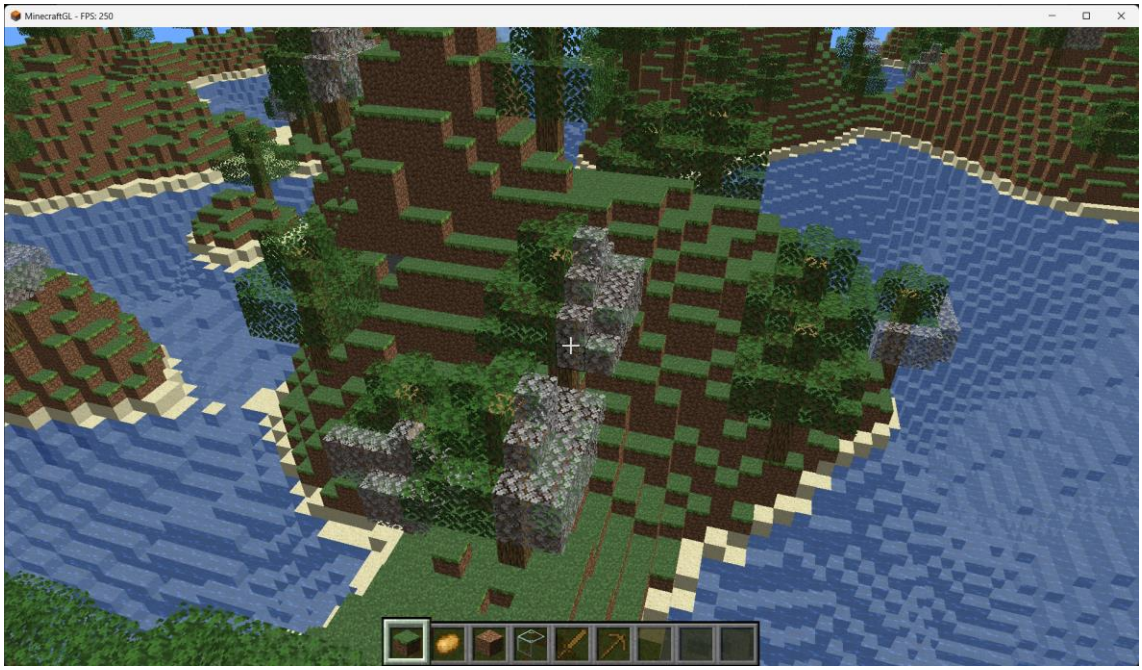


Figura 5.47: arbres amb fulles blanques que no es van poder col·locar en algun moment

5.9.16. `comprovarChunks`

En aquest apartat i en el següent veurem, en ordre d'execució del fil secundari, com el SuperChunk és capaç d'eliminar i construir chunks sencers i donar al jugador un món infinit. La primera funció que es crida és `comprovarChunks`, i ha de rebre el chunk on es troba el jugador.

La funció primer itera sobre els chunks que ja es troben a `Chunks` (és a dir, els que ja hi ha carregats). Si el punter no és `NULL` i no està marcat per descarregar, calculem la distància respecte el chunk del jugador. Si aquesta distància supera la especificada per la *flag* `DISTANCIA`, determinarem que el chunk està massa lluny com per seguir renderitzant-lo així que el marcarem per descarregar posant a `true` l'atribut `descarregant` i afegirem la posició a la cua de chunks que s'han de descarregar.

D'aquesta manera, tots els chunks que no es trobin a prop del jugador no es guardaran al map, i només els que es trobin a dins seran visibles.

Ara iterarem al voltant del jugador per trobar chunks que no estan carregats. Hem d'anar amb compte de no passar-nos de la `DISTANCIA` delimitada. Si trobem que un chunk no està carregat i tampoc es troba a la cua de chunks que s'han de carregar, aleshores l'afegirem.

Quan s'acabi aquesta funció, la cua `chunksCarregar` i `chunksDescarregar` tindran chunks que s'han de tractar, i aquí entren en joc les següents funcions.

5.9.17. `descarregarChunks` i `carregarChunks`

Un cop tenim elements a tractar a les cues, cada funció s'encarregarà d'obtenir el primer chunk de cadascuna i tractar `NCHUNKS` (aquesta *flag* determina quants poden tractar en una crida). `descarregarChunks` eliminarà el chunk del `map Chunks` i farà `delete` del punter, eliminant-lo finalment. El destructor de la classe Chunk es cridarà, esborrant el chunk. `carregarChunks`, per altra banda, obtindrà la posició del chunk que es vol crear i cridarà la funció `generarChunk` explicada a l'Apartat 5.9.15.

Amb aquestes dues funcions, estarem constantment creant els chunks necessaris i eliminant els que no facin falta. A la Figura 5.48 podem veure el resultat de tenir buides les cues: un "cercle" de radi `DISTANCIA`. Si ens movem, veurem que el cercle s'actualitza amb nous chunks, eliminant els que estiguin allunyats.

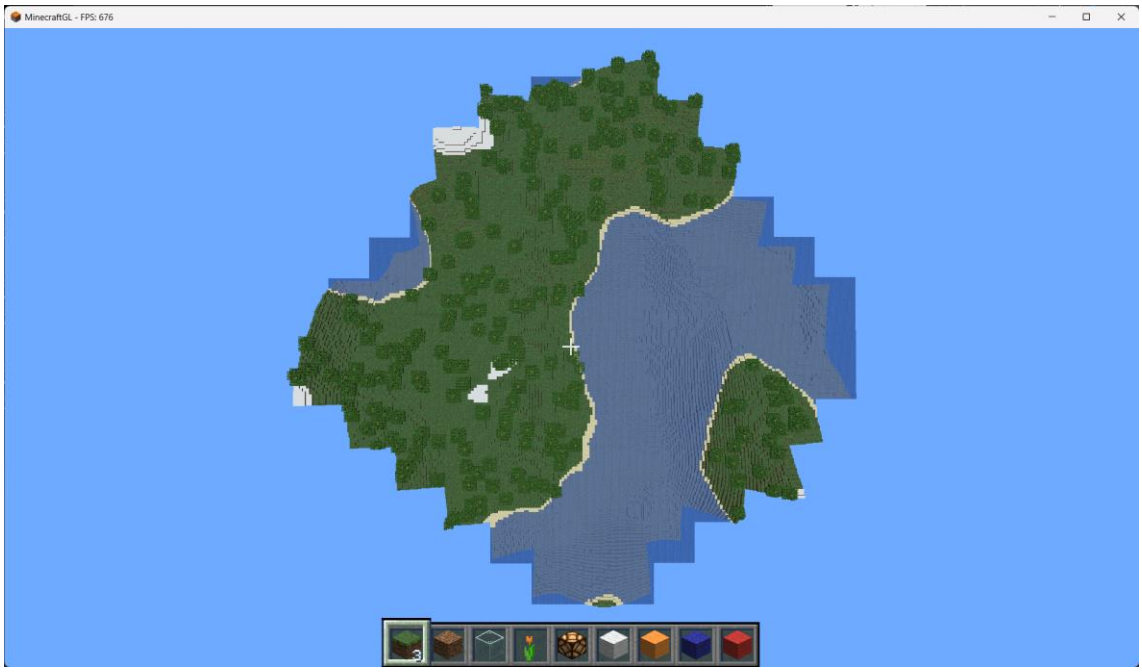


Figura 5.48: l'àrea de chunks que es forma al voltant del jugador amb una distància de 7 chunks

I així, el SuperChunk pot gestionar aquest món i mostrar al jugador un món infinit sense que se n'adoni. La il·lusió és encara major si pensem que la boira obstaculitza la visió de la creació de nous chunks, fent que el jugador no pugui veure l'aparició o desaparició espontània d'un chunk, donant la sensació de que els chunks sempre estan carregats.

5.9.18. `update`

La funció `update` iterarà també al voltant del jugador en busca de chunks carregats. Si en troba un, cridarà la funció `crearVertexs` i, si aquell chunk estava canviat, la funció construirà els vèrtexs com ja es va explicar. Sempre se li dona prioritat al chunk con comença el jugador.

5.9.19. `render` i `renderCub`

Per renderitzar el món, hem de cridar la funció `render` de cada chunk carregat visible. Per tant, iterarem sobre `Chunks` i els renderitzarem un a un, avaluant també la funció `esVisible`. Abans, però, hem de modificar la matriu model per moure els chunks. Si no, estarien tots al mateix lloc. Per això només hem de calcular una translació (molt ràpid gràcies a la funció `translate` de GLM) i multiplicar la posició del chunk pel tamany d'aquest.

Podríem acabar aquí, però si fem memòria, teníem dos maneres de renderitzar un chunk: els vòxels normals i els vòxels semitransparents. S'han de renderitzar en aquest ordre (normals i després semitransparents) per tal que no hi hagi cap problema de superposició.

Encara no hem aclarit per què estem fent aquesta separació de VBO i de renderitzat. Podríem simplement renderitzar-ho tot alhora i seria molt més senzill, però hi ha un problema. Com els chunks es construeixen d'esquerra a dreta, si mirem des de la dreta als chunks amb aigua, tot va bé, per què ja hi havia un chunk que veure, però si no, apareixen uns artefactes visuals. Això passa perquè renderitzem el chunk sencer en comptes de només la part "sòlida". Per tant, si mirem a través d'un vòxel semitransparent cap a un chunk que s'ha fet després, el que veurem és el fons, que és el que hi havia just abans de renderitzar el chunk al que estem mirant a través. A les Figures 5.49 i 5.50 podem veure aquests artefactes. A la Figura 5.49 es pot veure la gran quantitat d'aigua que "tapa" el món, mentre que a la 5.50 podem veure com el gel només deixa veure a través el chunk on es troba.

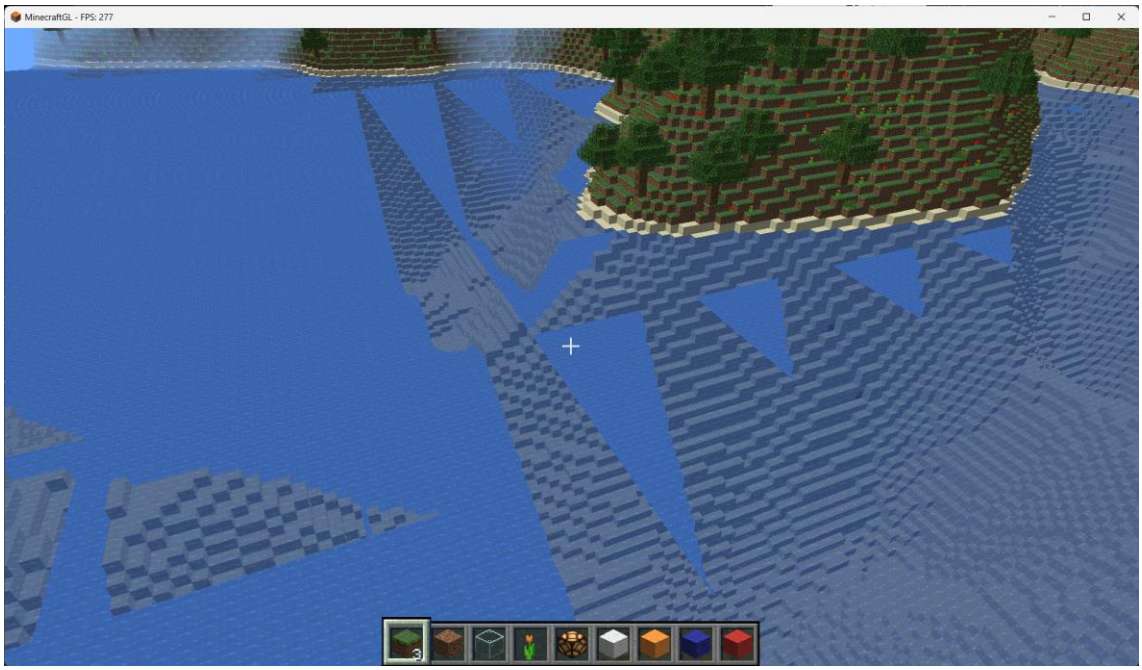


Figura 5.49: aigua no deixant veure el que hi ha darrera

Figura 5.50: un bloc de gel amb un altre bloc dins. Podem notar que només podem veure el seu propi chunk a través.

La solució és renderitzar primer els vòxels sòlids. D'aquesta manera, els semitransparents tindran alguna cosa que veure a través. Podem veure la diferència a les Figures 5.51 i 5.52.

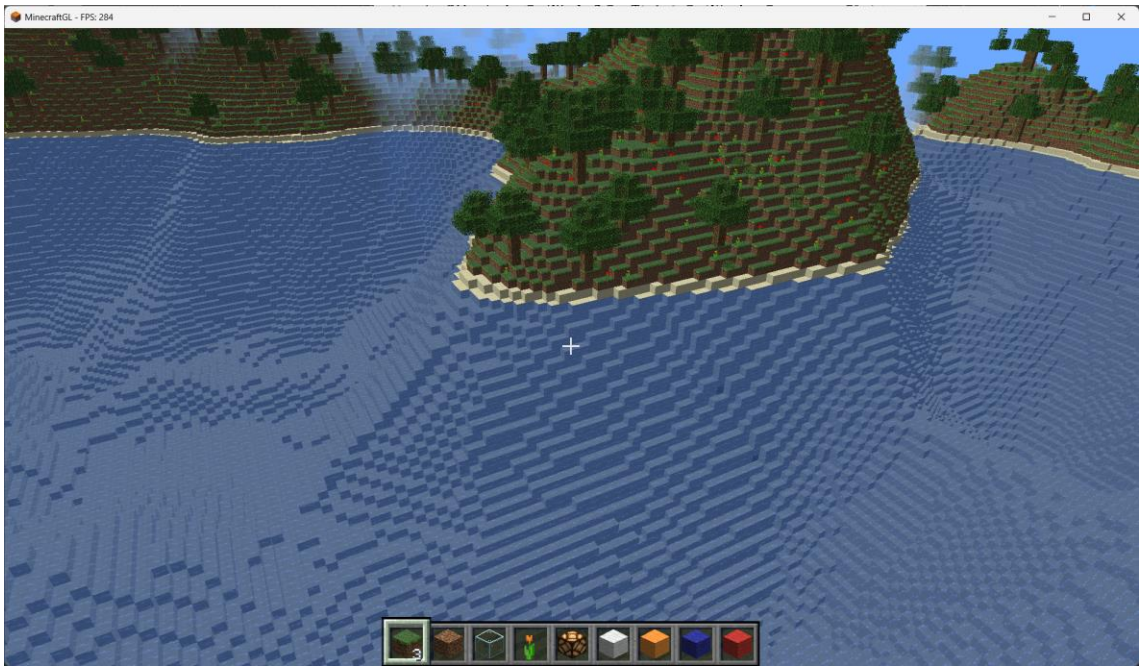


Figura 5.51: l'aigua és semitransparent i es pot veure el fons

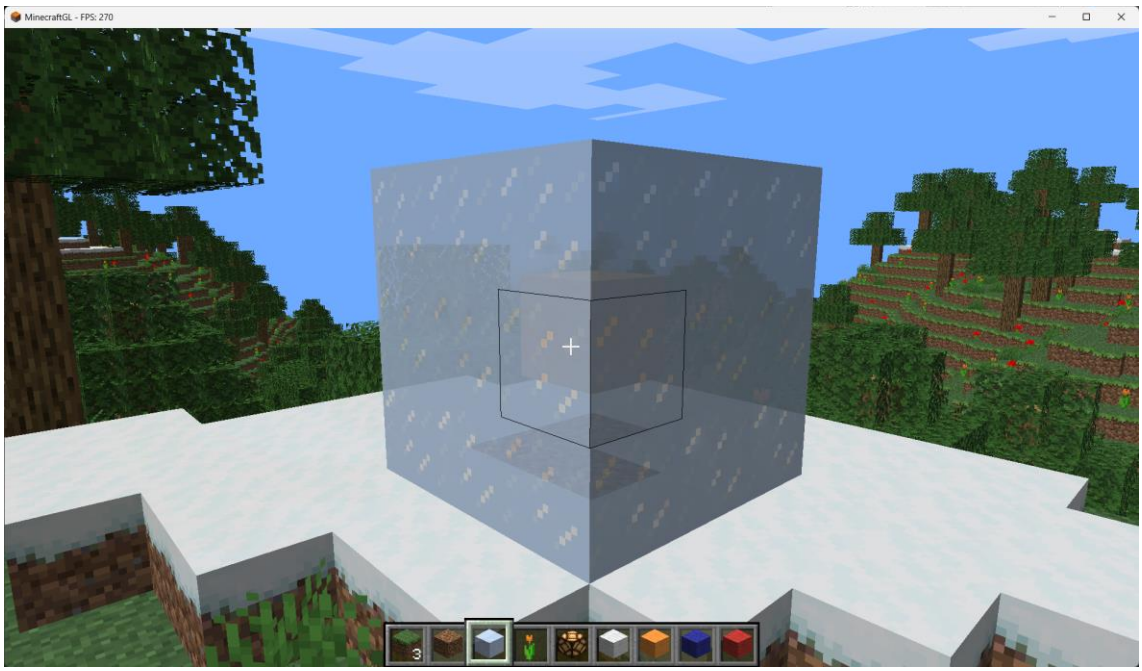


Figura 5.52: el bloc de gel deixa veure el que hi ha darrera

També hem de tenir un petit detall en compte, el GL_CULL_FACE, que es aquella tècnica que fa servir OpenGL per ocultar cares que miren en la mateixa direcció que el jugador. Això fa que, si el jugador està sota l'aigua i mira cap a dalt, no pugui veure l'aigua. Queda molt més realista si, just abans de renderitzar els vòxels semitransparents, desactivem la tècnica. Després la tornarem a activar. Veiem la diferència a la Figura 5.53.

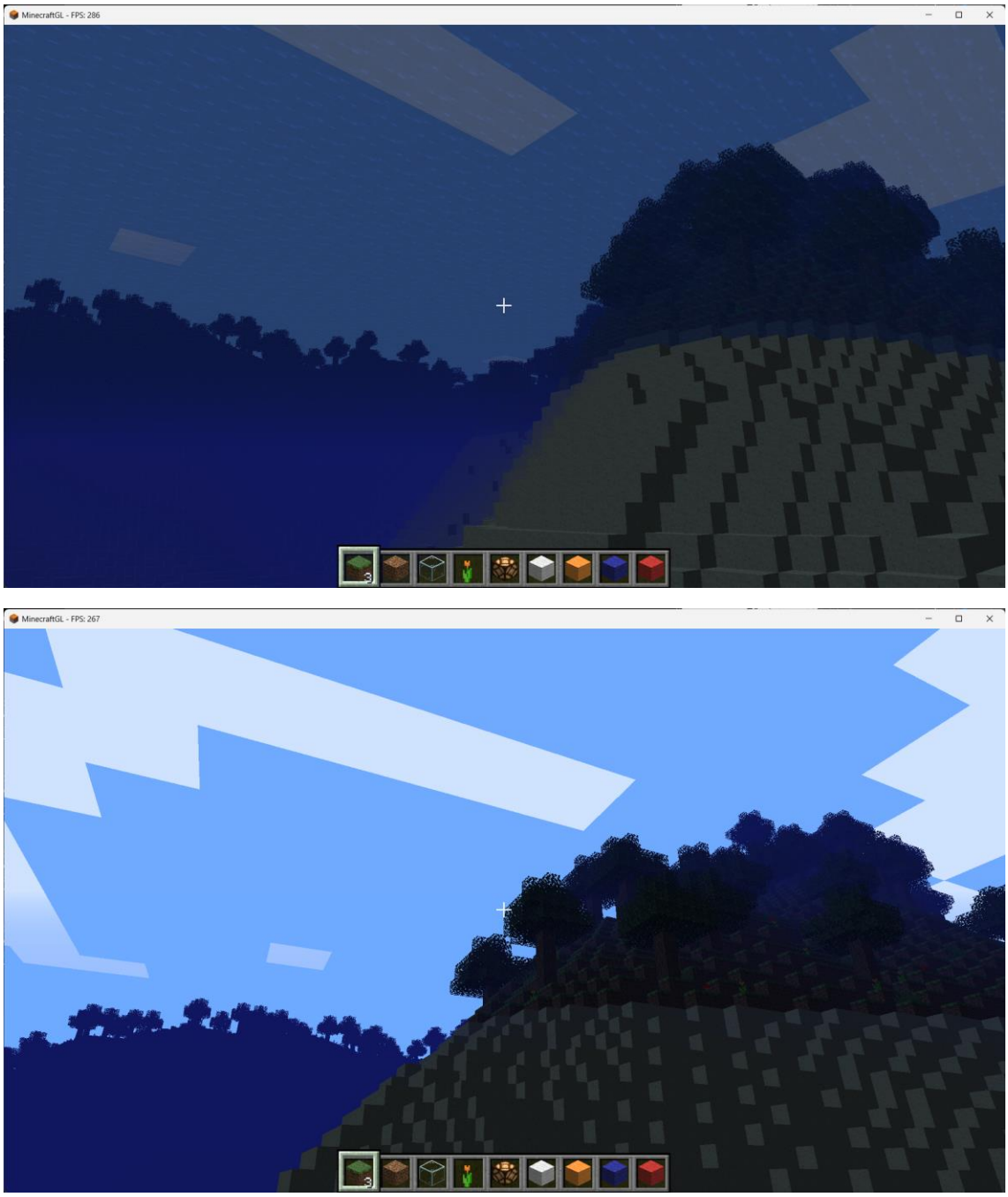


Figura 5.53: a dalt, desactivant temporalment la tècnica de culling, a baix, amb la tècnica sempre activa

La funció `renderCub` és molt més senzilla, ja que només ha de cridar la funció amb el mateix nom del chunk al que pertany la posició passada per paràmetre.

5.9.20. `BoundingBox`

Crea i renderitza el contorn del vòxel de la posició donada. Com sempre que volem renderitzar, necessitem un VBO i una sèrie de vèrtexs. En comptes de fer servir `glDrawArrays`, utilitzarem la funció `glDrawElements` per poder dibuixar línies (`GL_LINE_LOOP`). Per això fem servir una *array* de vèrtexs i a més una *array* que especifica l'ordre d'aquests. Si pensem en com renderitzàvem un vòxel, necessitàvem bastants vèrtexs perquè no sabíem si una cara es renderitzaria. En aquest cas sabem que sempre

hi hauran els mateixos vèrtexs (moguts a la posició que toca, és clar), així que podem fer servir només 8 i construir el cub sense emplenar indicant-li a OpenGL l'ordre d'aquests.

A la Figura 5.54 podem observar un cub de llana i la seva *bounding box* al voltant d'aquesta, marcant el vòxel per tal que al jugador li quedi clar que està sent seleccionat.

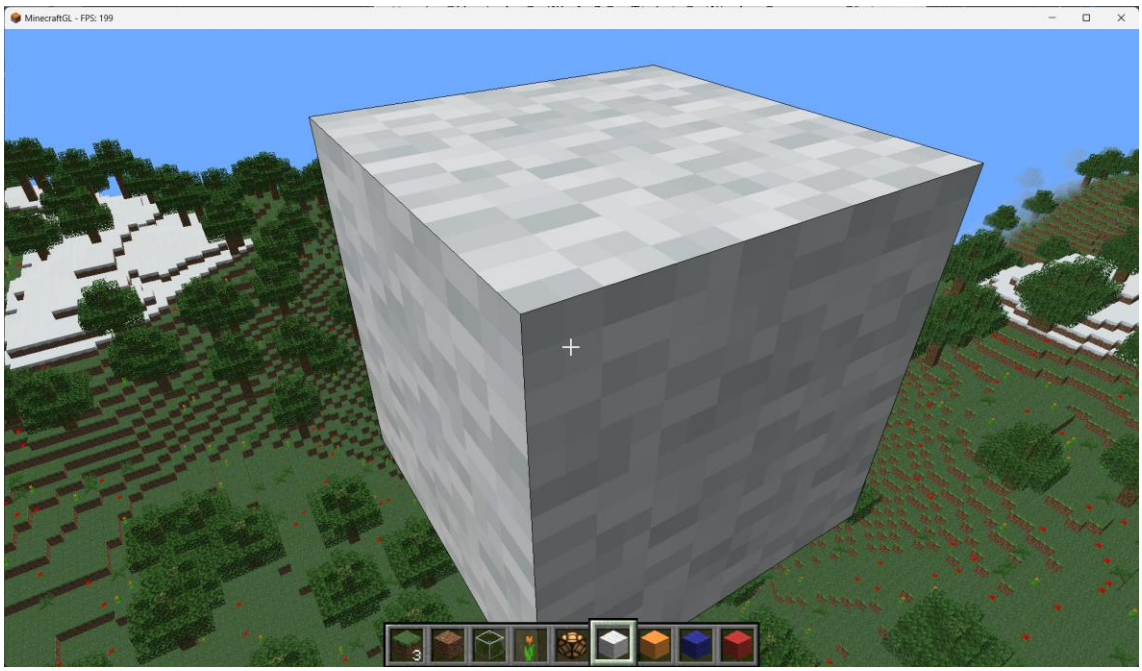


Figura 5.54: cub de llana amb la bounding box ressaltada en negre

5.10. Generador

La classe Generador s'encarrega de tot el que està relacionat amb la generació de món. Hem vist les parts més importants quan parlàvem del chunk, però l'usuari pot modificar gairebé qualsevol part d'aquesta classe per fer un món personalitzat, diferent i únic.

5.10.1. Atributs

- `llavor` (`int`): la llavor d'on surten els nombres aleatoris. Una mateixa llavor sempre retornarà la mateixa sèrie de nombres aleatoris. Totes les figures del document s'han capturat en un món amb la mateixa llavor: 874.
- `tipusMon` (`int`): pot ser `NORMAL` o `PLA`.
- `nivellMar` (`int`): altura fins on arriba el mar.
- `nivellNeu` (`int`): altura a partir de la qual la `GESPA` es torna en `NEU`.
- aigua (int): el tipus de vòxel per emplenar el mar. Per defecte és `AIGUA`.
- `probabilitatArbre` i `probabilitatFlor` (`float`): les probabilitats de plantar un arbre o una flor en gespa.
- `flors` (`vector<int>`): si es pot posar una flor, s'agafarà d'aquest vector un tipus aleatori de flor. Per defecte hi ha `ROSA`, `TULIPA_TARONJA` i `ARBUST`.
- `nivells` (`vector<int>`): la llista de capes. Hi ha 6 capes, de més profunda a més alta.

5.10.2. Soroll

Ens guardarem un vector amb `structs` anomenats Soroll. Cada Soroll tindrà un mapa de soroll associat, una importància i una sèrie de punts.

5.10.3. Constructor

El constructor cridarà a la funció `srand` i col·locarà la `llavor`. Si la *flag* `LLAVOR_RANDOM` és a `true`, s'agafarà una llavor aleatòria. Al constructor l'usuari pot definir diferents mapes de soroll que seran utilitzats per saber l'altura del terreny fent ús de la llibreria `FastNoiseLite`. Un cop tingui un mapa de soroll configurat, li ha de donar una importància i uns punts. Això serà utilitzat per saber l'altura final d'una columna a la funció `obtAltura`. Cada Soroll s'ha d'afegir al vector `noises` per tal que surti efecte.

5.10.4. `obtAltura`

La funció rep una `x` i una `y` del món i retornarà l'altura que hauria de tenir la columna. S'iterarà pel vector `noises` i obtindrem el valor de soroll de cadascun. Aquest valor s'utilitzarà als punts que tenen guardats per interpolar un valor. Els punts de cada `struct` representen una funció matemàtica per trossos. La x és el valor de soroll que hem obtingut i el que retornarà la funció és el valor determinat pels punts. Per això, sempre hauria d'haver mínim 2 punts que formin una recta. El valor final es multiplica per la importància del Soroll i es suma a l'altura final que es retornarà.

L'usuari pot modificar aquesta funció per retornar una altura arbitrària. Per exemple, podríem retornar un nombre arbitrari, com 64, o el sinus a la coordenada `x` i obtindríem el resultat mostrat a les Figures 5.55 i 5.56.

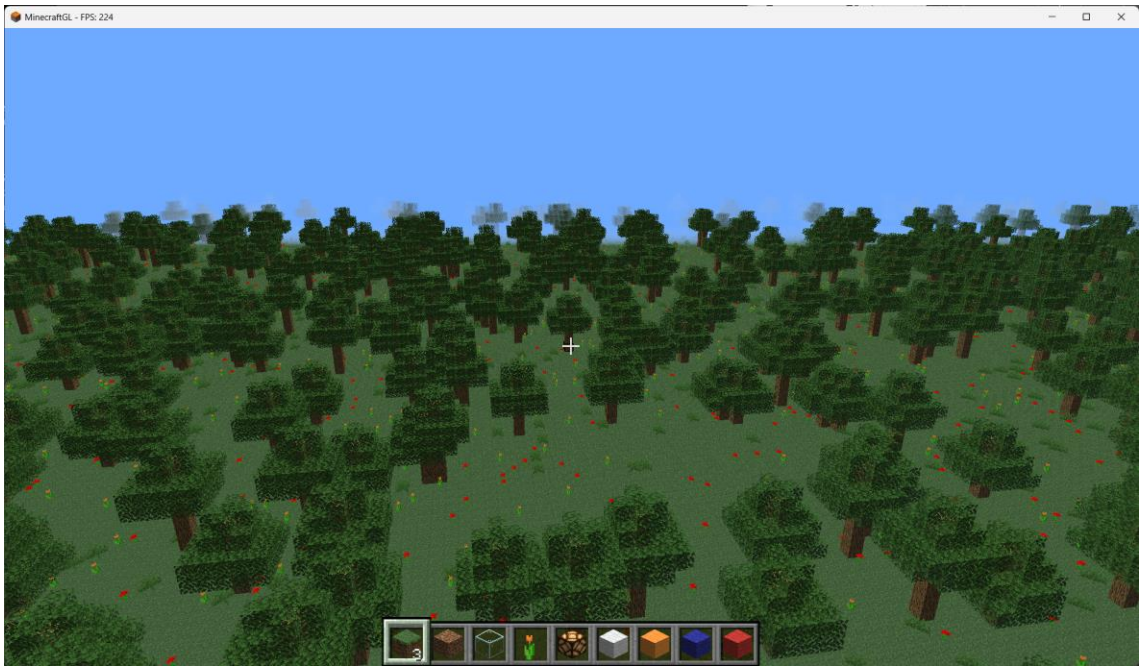


Figura 5.55: un món pla degut a que totes les altures han retornat el mateix nombre

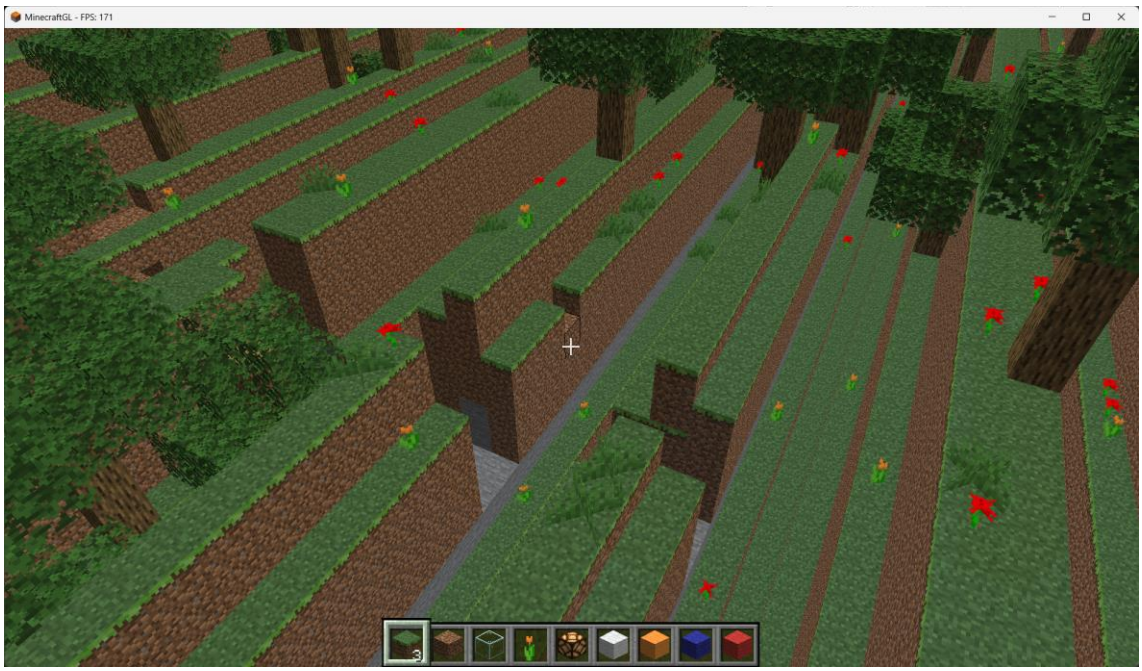
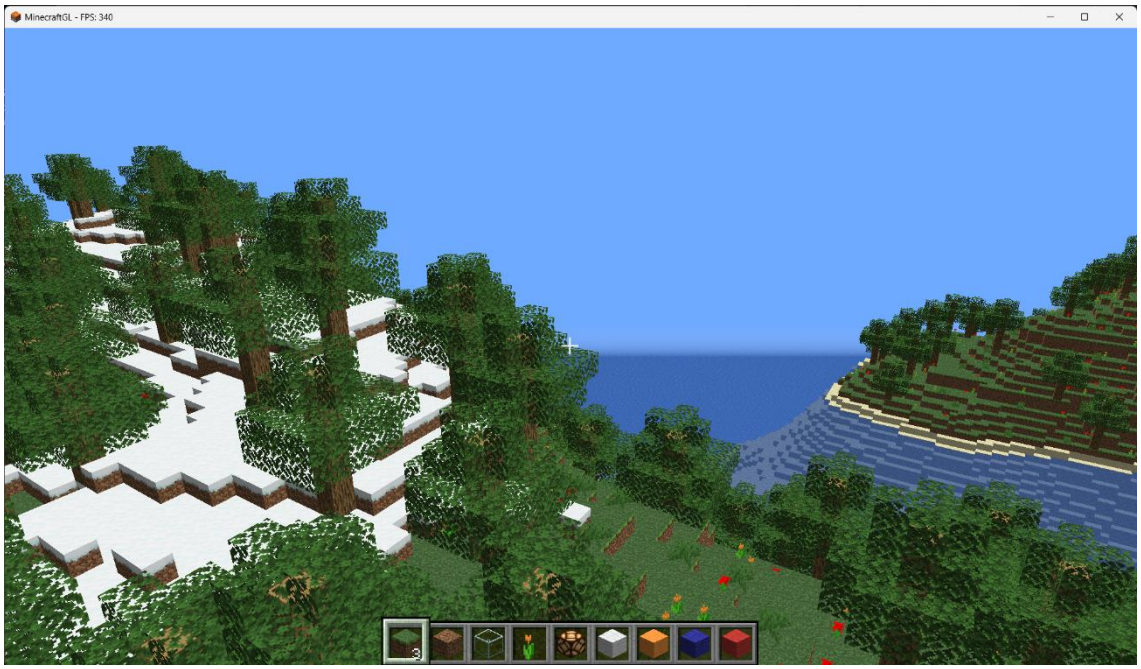


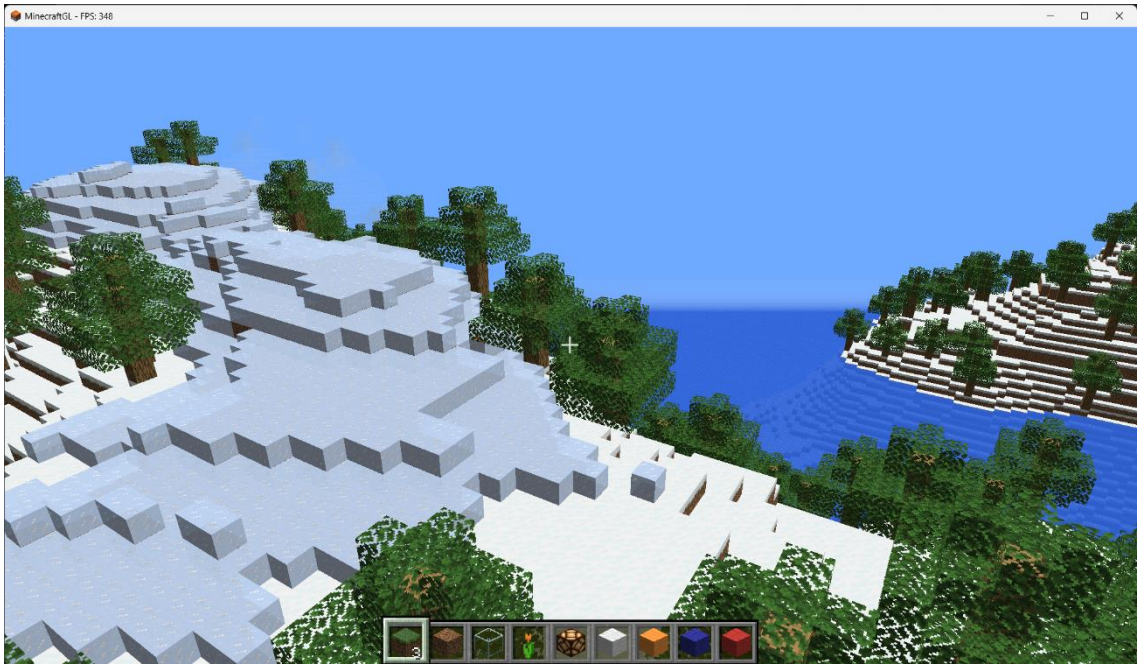
Figura 5.56: el món si la funció retornés el sinus d'x

5.10.5. `obtTipus`

La funció rep una altura dintre del rang (0,max) i retorna el tipus que hauria de tenir el vòxel. Un conjunt d'`ifs` avalua a quina capa es troba l'altura i retorna el tipus de vòxel que hi ha en aquella capa al vector `nivells`. Les diferents capes són: baixa, mitja, mitja-alta, platges, alta i alta a altura de neu, i els blocs per defecte assignats: `BEDROCK`, `PEDRA`, `TERRA`, `SORRA`, `GESPA` i `NEU`.

Si canviem el vector `nivells` amb diferents tipus podem obtenir resultats molt variats, com els mostrats a la Figura 5.57, formant un món nevat o un món fet només de llana. Les possibilitats estan a la mà de l'usuari.





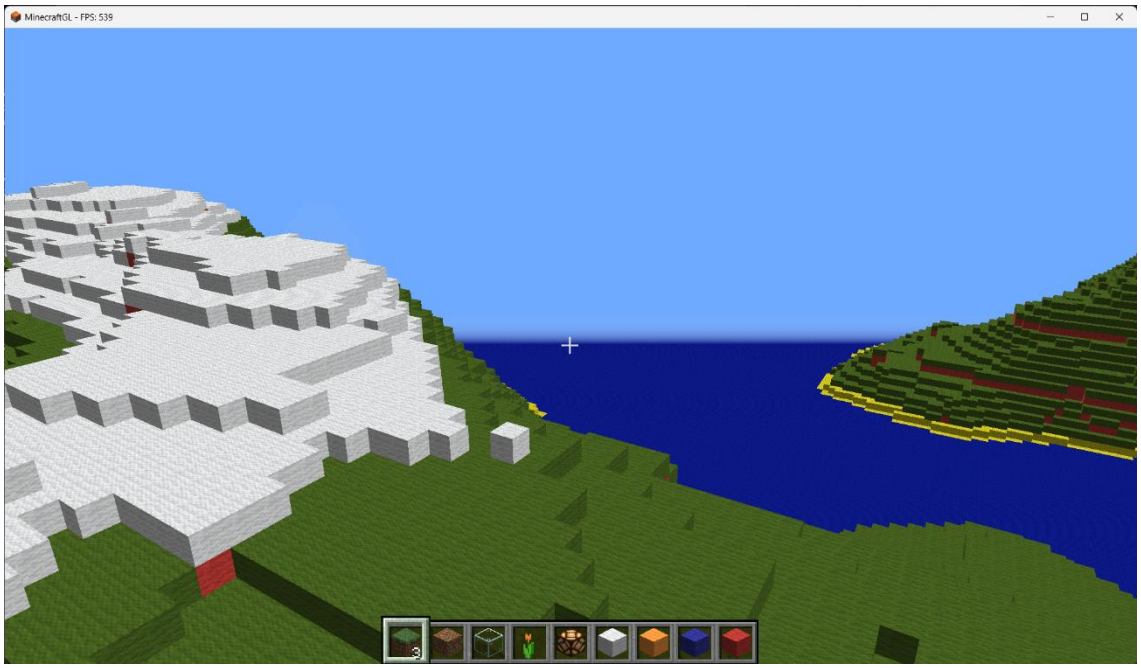
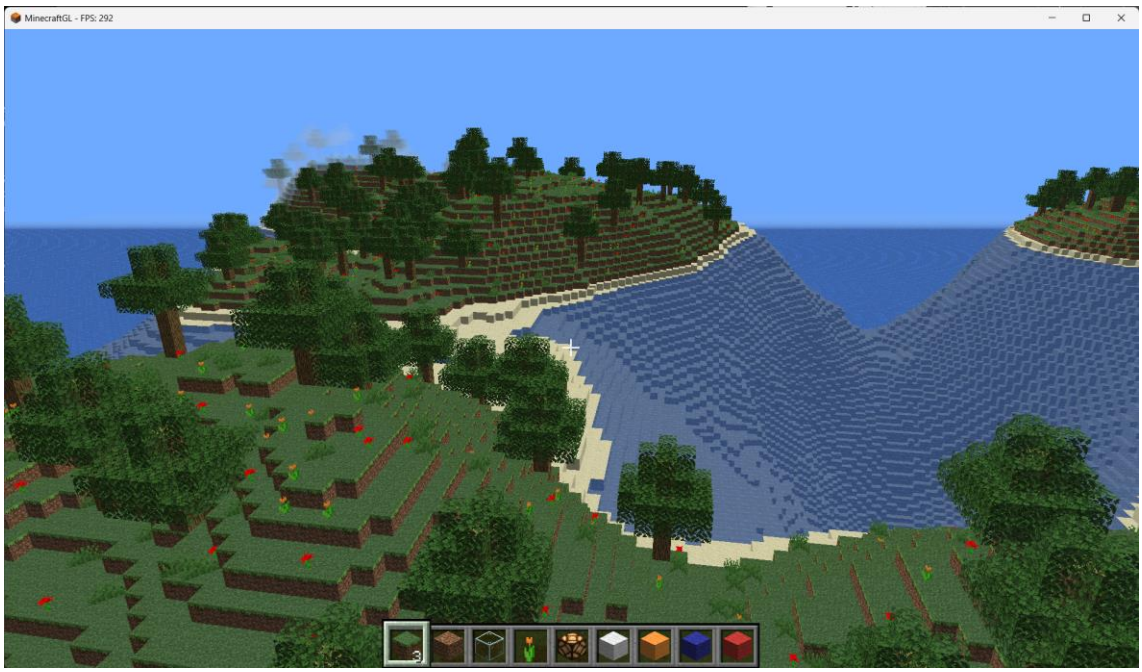


Figura 5.57: de dalt a baix: un món per defecte, un món de gel i un món fet amb llana

La flexibilitat d'aquesta classe permet a l'usuari crear qualsevol tipus de món, des d'un inundat a un ple de muntanyes, com els que mostra la Figura 5.58.



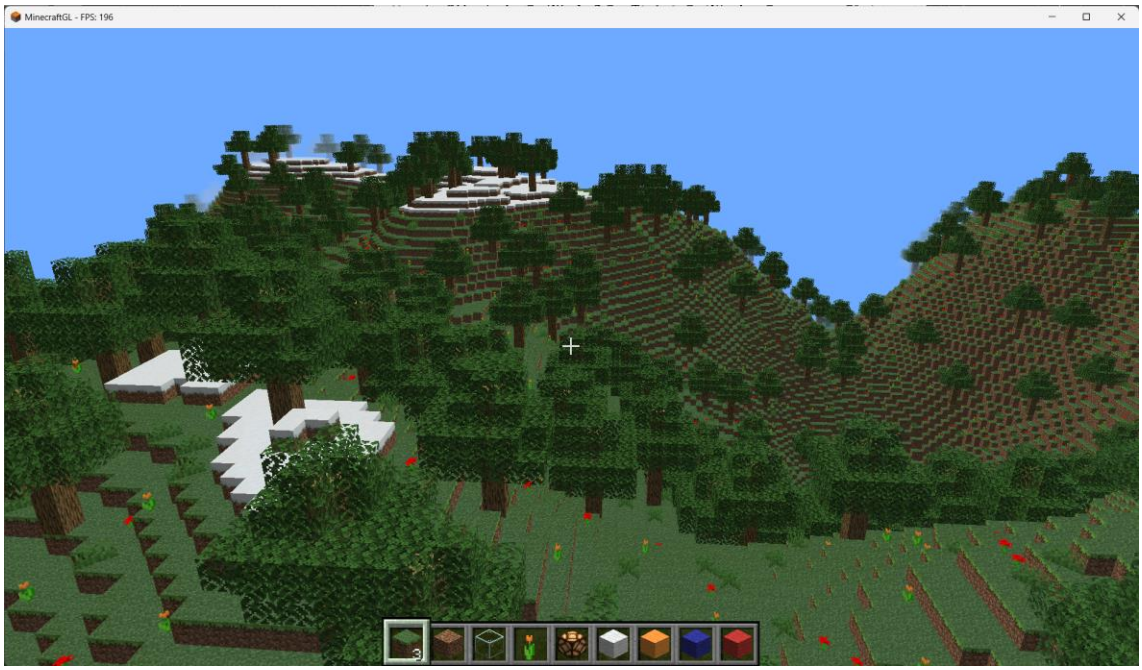


Figura 5.58: el mateix món gairebé inundat (a dalt) i sense aigua (a baix)

5.10.6. `obtFlor`

Retorna una flor aleatòria del vector de flors. Es pot posar qualsevol tipus, no només vegetació.

5.11. Nuvols

Es tracta d'una classe per gestionar els núvols. Són plenament decoratius i no afecten en res al jugador ni a les seves mecàniques, però la seva addició apropa molt més el motor al joc original.

5.11.1. Atributs

- `velocitat` (`float`): la velocitat a la que viatgen els núvols. Per defecte, 0.075.
- `altura` (`float`): l'altura dels núvols. Per defecte, 200.
- `tamany` (`float`): el tamany dels núvols. Per defecte, 15.
- `nit` (`bool`): determina si és de nit o no. Amb `canviaHora` podem canviar el valor.
- `textura` (`Textura*`): la textura associada als núvols.
- `shader` (`ShaderProgram*`): el *shader* associat.
- `projection` (`mat4`): la matriu de projecció de la càmera, pel *shader*.
- `VAOnuvols, VBO` (`int`): l'id del VAO i VBO associats.

5.11.2. Constructor

El constructor agafarà un nombre aleatori per posar els núvols. També agafarà la textura de Recursos anomenada `nuvols.png`. Els *shaders* emprats són `VertexNuvols` i `FragmentNuvols`. Després cridarà a la funció `initRenderData`, que inicialitzarà el VAO i el VBO. No la veurem en detall, ja que es tracta del mateix concepte que la funció render del chunk (Apartat 5.8.14): utilitzar les funcions `glBindBuffer`, `glBufferData`, `glVertexAttribPointer`, etc. Els núvols com a tal són en realitat un pla gegant amb unes dimensions de 1000*1000, en comptes de fer molts rectangles, un per cada núvol. L'`altura` ve donada per l'atribut.

El destructor elimina les referències dels punters i els *buffers* creats pel VAO i el VBO.

5.11.3. `update`

Per actualitzar els núvols, farem servir la posició del jugador i posarem sempre el pla a sobre del cap del jugador. Per moure els núvols, el que fem és recórrer la textura associada. No es mostra la textura sencera a sobre del pla, sinó que s'agafa una part, com si la retalléssim. Aquest retall és el que es mou segons la velocitat. Ens guardarem aquest *offset* per fer-lo servir a `render`.

5.11.4. `render`

Per renderitzar els núvols cridem la funció `usar` del *shader* per poder-lo utilitzar. Hem de col·locar l'*offset* per poder agafar la part correcta de la textura, el tamany que volem d'aquest retall i les matrius model, *view* i projecció com abans, ja que estem visualitzant un element en 3D. Les matrius *view* i projecció sempre són les mateixes, però la matriu model la modificarem amb una translació per posar el pla a sobre del jugador. Cridarem `glDrawArrays` com abans.

Si canviem tamany, podem veure que el retall canvia (veure Figura 5.59). A més, el *fragment shader* també té un factor de boira per poder donar-li un gradient als núvols i

fer-los més interesants. El *vertex* només determina la posició i li envia al *fragment* les UVs.

Figura 5.59: a dalt, núvols amb un tamany 5, a baix, amb un tamany de 200

5.12. Camera

Ara que ja sabem com renderitzar el món, parlarem de com el jugador pot visualitzar-lo. La classe Jugador tindrà una instància de Càmera, que serà l'encarregada de permetre al jugador girar el ratolí i veure el món que l'envolta.

5.12.1. Atributs

- `sensibilitat` (`float`): controla la sensibilitat del ratolí. Per defecte 0.1, però l'usuari el pot modificar.
- `yaw` (`float`): el gir en l'eix vertical de la càmera. Comença en -90 graus, però es pot canviar.
- `pitch` (`float`): el gir en l'eix horitzontal. Comença en 0.
- `ALTURA_JUG`: l'altura del jugador. Comença en 1.
- `frustum` (`Frustum`): una instància de l'`struct Frustum`.
- `fov` (`float`): l'angle que formen el pla esquerre i el pla dret del *frustum*, l'amplada del camp de visió.
- `near` i `far` (`float`): distància de la càmera del pla `near` i del `far`.
- `pos`, `cameraUp`, `right` i `front` (`vec3`): diferents vectors que determinen l'estat de la càmera. Tenen uns *getters*.
- `model`, `view` i `projection` (`mat4`): les matrius que hem estat tractant al llarg del projecte. Tenen uns *setters* i uns *getters*.

5.12.2. Definició

Abans de capficar-nos en la implementació de la càmera, hem de tenir clar què és cada element d'aquesta. La Figura 5.60 il·lustra els vectors `pos`, `front`, `up` i `right`.

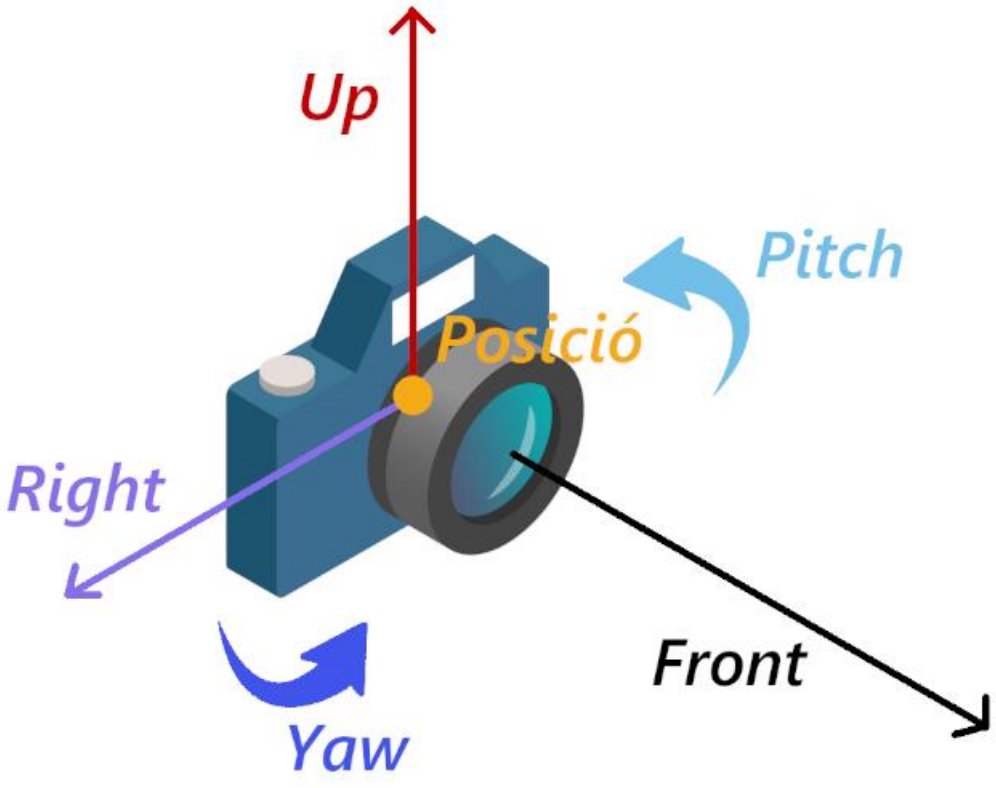


Figura 5.60: vectors d'una càmera

- `pos`: la posició de la càmera al món. El jugador mourà aquest punt amb les seves entrades.
- `front`: cap a on mira la càmera. El jugador sempre tindrà aquest vector davant seu.
- `up`: el vector que mira cap a dalt. Al principi hauria de ser `(0,1,0)`, però no sempre serà així.
- `right`: el vector que mira cap a la dreta. L'obtindrem de fer el producte vectorial de `front` amb un vector que sempre miri cap a dalt. No el podem fer amb up perquè no sempre serà igual.

Aquests vectors aniran canviant segons el jugador es mogui por el mapa i giri la càmera, però el sistema sempre és el mateix.

5.12.3. Constructor

El constructor només inicialitza la posició de la càmera al `(0,0)` del món, però elevada 90 blocs (l'usuari pot canviar això per començar al món on vulgui). A més, cridarà a la funció `mirar` per primera vegada.

5.12.4. mirar

La funció actualitza els tres vectors principals segons el `yaw` i el `pitch` actuals. Primer es calcula el `front`, i després el `right` i el `up`. A la Figura 5.61 trobem els càlculs necessaris per poder obtenir aquests vectors. Fixem-nos que el `right` s'obté fent el producte vectorial del `front` i un vector que **sempre apunta cap a dalt**, i el up fent el producte vectorial del `right` i el `front`. No hem de confondre l'`up` amb el vector que sempre apunta amunt.

```
1 void Camera::mirar() {
2     glm::vec3 nouFront;
3     nouFront.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
4     nouFront.y = sin(glm::radians(pitch));
5     nouFront.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
6     front = glm::normalize(nouFront);
7     right = glm::normalize(glm::cross(front,glm::vec3(0,1,0)));
8     cameraUp = glm::normalize(glm::cross(right, front));
9 }
```

Figura 5.61: codi de la funció mirar

5.12.5. girar

La funció actualitza `yaw` i `pitch` amb la posició del ratolí. Calcula quina és la diferència entre la posició anterior i la actual per saber com ha d'actualitzar els components. Els nous components es multipliquen per la sensibilitat del ratolí abans de ser sumats als atributs. A més, restringirem el pitch entre 90 i -90, ja que el jugador podria donar una volta sencera verticalment.

5.12.6. lookAt

Fa servir una funció de GLM que té el mateix nom. `lookAt` retorna una matriu *view* que podem fer servir a l'hora de renderitzar. Per això, necessita saber la posició, l'objectiu de la càmera (és a dir, `pos+front`) i l'`up`. La nostra funció, a més, actualitzarà els plans del `frustum` fent servir la funció `actualitzaPlans`, tot i que també retornarem la nova matriu *view* (guardant-la prèviament a la classe).

5.12.7. setProjection

La funció rep un *aspect ratio* per paràmetre i calcularà una nova matriu de projecció fent servir la funció `perspective` de GLM, que rep el `fov`, *l'aspect ratio*, el `near` i el `far`. El primer i els dos últims són configurables ja que es troben com atributs de la classe.

5.12.8. `teletransporta`

El que fa aquesta funció és bàsicament teletransportar la posició de la càmera a una de nova. Veurem que la fa utilitzar el jugador per moure's pel món.

5.12.9. `actualitzaPlans`

Per poder tenir un *frustum* actualitzat, necessitem que els plans que el formen reflectin els canvis que hi ha hagut a la càmera. La funció actualitza aquests plans fent els càlculs necessaris per obtenir-los. Un pla el definirem amb un punt i una normal. Recordem que el *frustum* és un prisma i la base és el pla `near`. No explicarem en detall els càlculs que s'han de fer per obtenir els plans, però la idea principal és obtenir la posició i cap a on apunten cadascun. Per això farà falta el `near`, el `far`, els vectors principals, el `fov` i *l'aspect ratio.* A la Figura 5.62 trobem els càlculs necessaris per calcular els plans.

```
1  void Camera::actualitzaPlans()
2  {
3      const float halfVSide = far * tanf(fov * .5f);
4      const float halfHSide = halfVSide * aspect;
5      const glm::vec3 frontMultFar = far * front;
6
7      frustum.nearFace = Pla(pos + near * front, glm::normalize(front));
8      frustum.farFace = Pla(pos + frontMultFar, glm::normalize(-front));
9      frustum.rightFace = Pla(pos, glm::normalize(glm::cross(frontMultFar - right * halfHSide, cameraUp)) );
10     frustum.leftFace = Pla(pos, glm::normalize(glm::cross(cameraUp, frontMultFar + right * halfHSide)) );
11     frustum.topFace = Pla(pos, glm::normalize(glm::cross(right, frontMultFar - cameraUp * halfVSide)) );
12     frustum.bottomFace = Pla(pos, glm::normalize(glm::cross(frontMultFar + cameraUp * halfVSide, right)) );
13 }
14
```

Figura 5.62: codi de la funció actualitzaPlans

5.13. Jugador

Per representar al jugador farem servir una classe que encapsuli tota la informació, incloent una instància d'una Càmera.

5.13.1. Atributs

- `mode` (`int`): representa el mode actual del jugador, `ESPECTADOR` o `CREATIU`.
- `enTerra` (`bool`): `true` si el jugador està tocant el terra.
- `aabb` (`AABB`): representa la *bounding box* del jugador.
- `velocitat` (`float`): la velocitat normal del jugador. Per defecte 12, es pot editar per anar més ràpid o més lent.
- `vel` (`vec3`): la velocitat a la que s'està movent el jugador.
- `càmera` (`Camera*`): una instància de la classe Càmera. Té un *getter.*
- `inventari` (`Inventari*`): una instància de la classe Inventari.
- `SALT`: quant d'alt ha de saltar el jugador.
- `GRAVETAT`: a quina velocitat cau el jugador.

5.13.2. Constructor

El constructor s'encarrega de crear una instància d'Inventari i especificar la mida del jugador a `aabb`.

5.13.3. `obtPos`, `obtPosBloc`, `obtPos2D` i `chunkActual`

Aquestes funcions retornen la posició del jugador en contexts diferents. `obtPos` simplement retorna la posició de la càmera i `obtPos2D` retorna aquesta posició en un `vec2` sense el component y, mentre que `obtPosBloc` retorna la posició del vòxel on es troba el jugador, i es pot retornar el que té al cap o als peus. `chunkActual` retorna la posició del chunk on es troba el jugador.

5.13.4. `canviaMode`

Permet canviar el mode del jugador. `ESPECTADOR` dona llibertat total per moure's, mentre que en `CREATIU` estarà sotmès a la gravetat i col·lisionarà amb altres vòxels. Tots dos tenen accés a l'inventari amb vòxels infinits.

5.13.5. `caminar`, `correr` i `parar`

Totes tres funcions tenen a veure amb el moviment del jugador. La funció `caminar` fa que la velocitat actual sigui igual a la velocitat normal, mentre que `correr` multiplica aquesta velocitat per 1.5. `parar` fa que el jugador pari en sec, fent que el vector `vel` es posi a 0.

5.13.6. `moure`

Per moure el jugador, necessitarem saber quines tecles està pressionant (que les obtenim a la funció `key_callback`, explicada a l'Apartat 5.1.9). També haurem de calcular dos vectors per saber la direcció del jugador fent ús dels vectors `front` i `right` de la Càmera. Depenent de la tecla (o tecles) que s'hagi pres, actualitzarem el vector de velocitat,

principalment la `x` i la `z`, amb els vectors de direcció que hem calculat. Si volem saltar, només hem d'igualar el component `y` de la velocitat a `SALT` només si som a terra.

5.13.7. `update`

La funció principal del jugador. Per fer que es mogui de veritat, hem d'actualitzar la posició de la càmera fent servir la funció `teletransporta` (Apartat 5.12.8) i passant `pos+vel`. A més, hem de multiplicar `deltaTime` per tal que, en comptes de moure `vel` metres per **frame**, es mogui `vel` metres per **segon**. Així la càmera s'actualitzarà amb la nova posició d'acord amb la velocitat.

Abans, però, hem de detectar si som al mode `CREATIU`, ja que haurem de treballar la detecció de col·lisions. Per això actualitzarem `aabb` amb la posició i velocitat actuals del jugador. Iterarem pels blocs que tingui al voltant per saber si ha col·lidit amb algun d'ells. La funció `sweptAABB` de l'struct `AABB` informarà si ha hagut alguna col·lisió i en quins eixos.

Si l'eix ha estat a la `y`, vol dir que som a terra, així que actualitzem `enTerra` amb `true` i el component `y` de `vel` amb un `0`. Si ha estat en la `x` o en la `z`, simplement posarem un `0` al component corresponent.

Independentment de si hi ha hagut una col·lisió o no, aprofitarem per comprovar si hi ha un bloc justament a sota del jugador. Si no és així, posarem `enTerra` a `false`. Al final de la funció, si `enTerra` és `false`, hem d'aplicar `-GRAVETAT*deltaTime` al component `y` de vel.

La funció `sweptAABB` que hem mencionat abans comprova si dos `AABB` estan xocant avaluant les posicions, les mides i les velocitats que hi ha als `structs`. Detectarà la posició del jugador dins del bloc que estiguem avaluant i, si hi ha hagut una col·lisió, la funció retornarà la normal on hi ha hagut la col·lisió.

5.14. Sprite

Ara que ja hem renderitzat el món en 3D, passarem a la part en 2D: les interfícies d'usuari. En comptes d'utilitzar una llibreria externa per encarregar-se d'aquesta part, és molt més interessant pel projecte implementar les nostres pròpies classes que tinguin com objectiu el renderitzat de les interfícies.

Ja vam explicar a l'Apartat 4.2.8 que un sprite es tracta d'una textura fàcilment manipulable. Per tant, la unitat mínima per renderitzar qualsevol imatge 2D en pantalla serà la classe Sprite.

5.14.1. Atributs

- `pos`, `tamany` i `escala` (`vec2`): atributs per les transformacions del sprites. Tenen *getters*.
- `centrat` (`bool`): si l'sprite hauria de renderitzar-se centrat o ancorat a la cantonada superior esquerra.
- `nom` (`string`): nom de l'sprite. Igual que les textures, s'identificaran gràcies a això.
- `posicioMapa` i `tamanyMapa` (`vec2`): si la textura és un mapa, informa d'on s'ha de retallar. Podem especificar-los directament al codi.
- `color` (`vec4`): el color que tindrà l'sprite.
- `model` (`mat4`): la matriu amb la transformació final de l'sprite.
- `indexZ` (`float`): amb això sabrem l'ordre de renderitzat dels sprites.
- `visible` (`bool`): si false, no es renderitzarà.

5.14.2. Constructor

Al constructor se li ha de passar una sèrie de paràmetres: la textura que tindrà l'sprite, el nom (que hauria de ser únic), una posició en pantalla en píxels i, opcionalment, l'escala i si ha de ser centrat o no. Per defecte no es centra l'sprite. Després cridarà la funció `transformar`, que calcularà la matriu `model` de l'sprite. Recordem que la textura ha d'estar precarregada a la classe Recursos. A la Figura 5.63 trobem un exemple de com podem crear un sprite.

```
1 Sprite* logo = new Sprite(Recursos::obtTextura("logo.png"), "Logo", glm::vec2(renderer->width / 2, renderer->height / 2), glm::vec2(0.4), true);
```

Figura 5.63: a dalt, línia per crear l'sprite, a baix, el resultat

5.14.3. `teletransportar` i `moure`

Aquestes dos funcions serveixen per fer una translació de l'sprite. Podem teletransportar l'sprite a una nova posició o podem afegir-li un *offset* a la posició original. Després es crida `transformar`.

5.14.4. `escalar`

La funció canvia l'escala de l'sprite i crida `transformar`.

5.14.5. `centrar`

La funció canvia centrat i crida `transformar`. Si està centrat, tindrà `pos` com a centre i, si no, es renderitzarà a partir de la cantonada esquerra superior.

5.14.6. `transformar`

Aquesta funció és la més important de l'sprite. Hem vist que té un atribut model, que és la matriu que, com ja hem explicat, representa transformacions al model. Un cop especificats els atributs de l'sprite, `transformar` aplicarà les transformacions adients a la matriu model: primer fent la translació i després escalant-lo. Si està centrat, es farà una segona translació per moure'l on toca.

5.15. SpriteRenderer

La classe Sprite no té cap funció render, i tampoc tenim una manera de guardar-los i gestionar-los còmodament. Per això tenim la classe SpriteRenderer, que tindrà com objectiu guardar els sprites que es creïn i renderitzar-los correctament.

5.15.1. Atributs

- `Sprites` (`map<string,Sprite*>`): un `map` que es guarda tots els sprites que es renderitzaran. Amb `obtSprite` podem aconseguir el punter a l'sprite que vulguem.
- `SpritesOrdenats` (`multimap<int,Sprite*>`): igual que `Sprites`, però ordenats per `indexZ`.
- `queda` i `VBO` (`int`): els ids corresponents per renderitzar.
- `shader` (`ShaderProgram*`): el *shader* que es farà servir per renderitzar.

5.15.2. Constructor

El constructor crearà i carregarà un ShaderProgram a partir dels arxius `VertexSprite` i `FragmentSprite` i cridarà a la funció `initRenderData`. Podem notar que aquest nom es repeteix bastant al projecte, i és perquè qualsevol model que sapiguem que tindrà sempre els mateixos vèrtexs (al contrari que un chunk), podem construir els vèrtexs un cop i no canviar-los en tota la vida del motor.

Així, `initRenderData` construirà un quadrat amb els mateixos vèrtexs que els mostrats a la Figura 5.24. També haurà de generar el *buffer* del VBO i cridar les funcions pertinents (`glGenBuffers`, `glBufferData`, etc). Els vèrtexs dels sprites, com els núvols, només tindran informació de posició i d'UVs.

El destructor eliminarà el VAO i el VBO, la referència al shader i tots els sprites que tingués emmagatzemats.

5.15.3. `afegirSprite`

A la funció se li ha de passar un `Sprite*` per paràmetre. Si ja existia un sprite amb el nom que té el del paràmetre, s'eliminarà i es substituirà amb el nou. Quan acaba la funció, l'sprite queda afegit a `Sprites` i ordenat adequadament a `SpritesOrdenats`.

5.15.4. `eliminaSprite`

Aquest cop la funció rebrà un nom. El nom es buscarà a `Sprites` i, si el troba, eliminarà l'sprite associat.

5.15.5. `canviaIndex`

La funció rep el nom de l'sprite al que se li ha de canviar l'`indexZ` i el nou índex. La funció reorganitzarà l'sprite per tal que `SpritesOrdenats` sigui coherent.

5.15.6. `render`

La funció renderitza tots els sprites marcats com visibles que tingui la classe guardats. Per cada Sprite a SpritesOrdenats, cridarem la funció `DrawSprite`. Aquesta activa el *shader* i col·loca la matriu model de l'sprite que estem renderitzant i la posició i el

tamany del retall del mapa de textures si la textura és un mapa al *shader*. També ha d'activar la textura de l'sprite cridant el mètode `use` de la classe Textura. Al final, ha de cridar `glDrawArrays` i dibuixar el pla que contindrà l'sprite.

5.16. TextRenderer

Apart de dibuixar textures en pantalla, per tenir una interfície entenedora pel jugador necessitem també mostrar text. Per això utilitzarem la classe TextRenderer, que s'ocuparà de proporcionar la funció ideal per aquest problema.

5.16.1. Atributs

- `VAO` i `VBO` (`int`): com ja estem acostumats, ids per renderitzar.
- `face` (`FT_Face`): la font que estem utilitzant per renderitzar text.
- `Characters` (`map<char,Character>`): un `map` amb tots els caràcters que es poden renderitzar.
- `shader` (`ShaderProgram*`): com sempre que volem renderitzar, necessitem un shader.

5.16.2. `Character`

`Character` és un `struct` que es guardarà la informació d'un sol caràcter. Renderitzar una font no és tan senzill com sembla, hem de tenir en compte la mida del caràcter, on comença i on pot anar el següent, etc. A la Figura 5.64 podem veure tot el que comporta renderitzar-ne un. L'`struct` es guardarà la mida, el *bearing* i l'*advance* (a la figura tenim remarcats què són aquests dos últims) i una textura per tal de ser renderitzat.

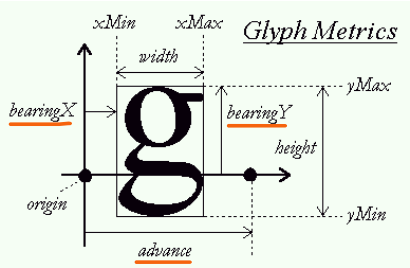


Figura 5.64: mètriques d'un caràcter

5.16.3. Constructor

El constructor, com sempre, haurà de crear el *shader* i generar el *buffer* del VBO. En aquest cas també especificarem que els vèrtexs de cada caràcter tindran 4 valors (posició i UVs).

5.16.4. `Load`

Aquesta funció serveix per carregar una font. Ha de rebre el nom de la font i la mida a la que la volem carregar. És millor carregar-la amb una mida gran i després escalar amb nombres petits que al revés.

Primer hem de carregar la llibreria de freetype amb `FT_Init_FreeType` i després crear una nova font amb `FT_New_Face`, al que li hem de passar el nom de la font. Després, per cada caràcter de la font, hem de crear una textura única per cada caràcter. Per sort, una `FT_Face` proporciona un *bitmap* que es pot utilitzar per això. No crearem una classe Textura, sinó que farem servir `glGenTextures` i `glTexImage2D`. Aquesta textura, juntament amb informació que podem aconseguir de la `FT_Face`, construirem el

caràcter. Un cop hem construït tots els caràcters i guardats al `map`, podem destruir la llibreria. Amb aquesta funció permetem que l'usuari pugui utilitzar accents.

5.16.5. `RenderText`

La funció principal del TextRenderer i la que es cridarà constantment. Necessita saber què text renderitzar, la posició en pantalla, l'escala, si tindrà un fons o no, el color del text, si tindrà ombra i un índex `Z` per ordenar.

Iterarem sobre l'`string` del text per obtenir cada caràcter i aconseguir la textura adient. Per cada caràcter construirem els vèrtexs en aquesta funció. Si tornem a la Figura 5.64, xMin, yMin i xMax i yMax determinen els vèrtexs que s'han de crear. Es calculen fent servir el *bearing* i així alinear tots els caràcters. Posarem els vèrtexs al VBO i després hem de cridar `glDrawArrays`. Per tal que tots els caràcters no es quedin al mateix lloc, anirem sumant l'*advance* a una `x` que determinarà on a la línia col·locar el següent caràcter.

Si decidim utilitzar un fons, obtindrem els vèrtexs en els dos eixos més petits i més grans que haguem renderitzat. Així podrem renderitzar un rectangle que cobrirà tot el text, amb un marge de 3 píxels (editable per l'usuari).

Si utilitza ombra, la funció tornarà a cridar `RenderText`, però amb un *offset* de 3 píxels (també editable per si es vol que sigui més o menys) per tal que es renderitzi a un costat i amb un color un 75% més fosc que l'original.

A la Figura 5.65 trobem un exemple de tres `strings` renderitzats amb diferents colors, un amb ombra i un altre amb fons.

```
1 textRenderer->RenderText("Normal", 5, 5, 0.2, false, {1,1,0.75}, false);
2 textRenderer->RenderText("Ombra", 5, 40, 0.2, false, { 1,0.5,0 }, true);
3 textRenderer->RenderText("Amb fons", 5, 80, 0.2, true, { 1,1,0 }, false);
```

Figura 5.65: tres string renderitzats en pantalla

5.17. HUD

Ara que sabem tant renderitzar sprites com text, podem començar a construir les interfícies d'usuari. A la classe HUD l'usuari pot col·locar qualsevol imatge que vulgui renderitzar com part de les interfícies, ja sigui la vida, text amb informació, etc.

Per defecte veurem que el HUD només té un sprite i text.

5.17.1. Atributs

- `visible` (`bool`): determina si es renderitza o no el HUD. Alternable amb `alternaVisibilitat`.
- `debug` (`bool`): determina si es renderitza el text amb informació o no. Alternable amb `modeDebug`.
- `renderer` (`SpriteRenderer*`): una instància de l'SpriteRenderer.
- `textRenderer` (`TextRenderer*`): una instància del TextRenderer.

5.17.2. Constructor

El constructor crearà les instancies dels renderers i iniciarà la font `Minecraft.otf` amb la funció `load`. Es pot posar la font que es desitgi i s'ha de col·locar a la carpeta `/Fonts`. Al constructor li passarem un punter a l'inventari per tal que pugui compartir les instàncies dels renderers.

També crida a la funció `iniciaSprites`, que és on crearem tots els sprites que necessitem per renderitzar. Per defecte només es crea el sprite *crosshair*, que és el que podem veure a la Figura 5.66 per tal que el jugador sàpiga a quin vòxel està mirant.

Figura 5.66: el crosshair

5.17.3. `render`

Aquesta funció crida al `render` de l'SpriteRenderer, renderitzant els sprites que tingui emmagatzemats. A més, si `debug` és `true`, renderitzarà text amb informació rellevant: la posició del jugador i el cub al que està mirant (veure Figura 5.67).

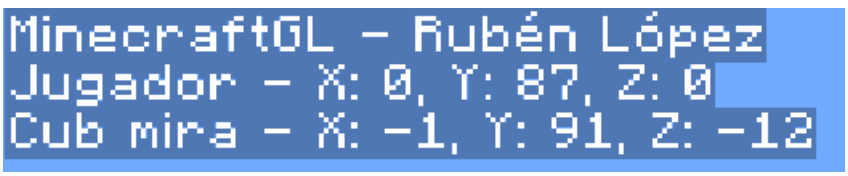


Figura 5.67: text debug

5.18. Inventari

Una de les mecàniques més importants de *Minecraft* i que sovint els jocs que surten a partir d'aquest s'obliden d'incorporar és la capacitat de poder fer construccions variades. Per això, s'ha de permetre al jugador accedir a una varietat important de vòxels a la vegada que fem aquest accés intuïtiu. La classe Inventari s'encarregarà de brindar al jugador amb tot tipus de blocs.

El jugador tindrà una barra a sota de la pantalla mostrant els blocs que té actius, és a dir, que activament pot agafar i col·locar, i un inventari al que podrà accedir i intercanviar aquests blocs.

5.18.1. Atributs

- `creatiu` (`bool`): determina si es renderitza o no el HUD. Alternable amb `alternaVisibilitat`.
- `visible` (`bool`): determina si es renderitza el text amb informació o no. Alternable amb `modeDebug`.
- `dintre` (`bool`): determina si som a dins de l'inventari gran.
- `itemsInicials` (`vector<string>`): els items amb els que comença el jugador a la barra.
- `spriteSlot` (`Sprite*`): representa una caixa que mostra el bloc actiu.
- `caixeta` (`Sprite*`): representa una caixa on posarà el nom del bloc.
- `sobre` (`Slot*`): l'Slot on es troba el ratolí a sobre.
- `ultim` (`Slot*`): l'últim Slot clicat.
- `slotMouse` (`Slot*`): l'Slot que té el ratolí sempre, diferent de `sobre`.
- `slotSeleccionat` (`int`): el bloc actiu actualment. `obtenirItemActual` actua com un *getter* per l'Item que hi ha a l'Slot seleccionat.
- `inventari` (`vector<Slot*>`): representa l'inventari de la barra de sota.
- `inventariGran` (`vector<vector<Slot*>>`): representa l'inventari amb tots els blocs.
- `mapaBlocs` i `mapaItems` (`Textura*`): per poder renderitzar blocs i items.
- `renderer` (`SpriteRenderer*`): una instància de l'SpriteRenderer.
- `text` (`TextRenderer*`): una instància del TextRenderer.
- `Shader` (`ShaderProgram*`): per renderitzar sempre necessitem un *shader*.
- `VAO i VBO` (`int`): com sempre, els ids dels *buffers*.

5.18.2. Slot

Als atributs trobem diferents referències a la classe Slot. Aquesta serveix per identificar cadascun dels objectes de l'inventari. El ratolí sempre tindrà un Slot que el seguirà per donar la sensació de que estem agafant un objecte, quan en realitat el que s'està fent sempre son intercanvis entre els Slots dels dos inventaris amb aquest del ratolí.

Un Slot té un id per diferenciar-lo de la resta, un Sprite per poder visualitzar-lo i la quantitat que hi ha d'un mateix Item. També conté una funció `render` per poder renderitzar un bloc (veurem què significa això a l'Apartat 5.18.3), una funció `obtItem` que retorna l'Item que conté l'Slot i una funció per actualitzar l'Sprite.

Aquesta última funció actualitzarà l'Sprite segons l'Item que conté l'Slot. Si no hi ha res, posarà `visible` a `false`. Si és un bloc, posarà la textura `mapaBlocs` amb les textures dels blocs i posarà el color que toqui si es tracta d'un bloc com `GESPA`, i si no, posarà la textura `mapaItems`.

En resum, la classe Slot representa un objecte a l'inventari i servirà per intercanviar objectes entre els inventaris, per saber quin tenim seleccionat, etc. Hem de tenir en compte que, tot i que parlem d'Items, un bloc també és un Item, ja que sinó no podríem representar-lo a l'inventari.

5.18.3. Constructor

El constructor crearà tots els Slots necessaris tant a inventari com `inventariGran`. Podem determinar quants Slots crearà en cada cas amb `MAX_ITEMS` i `MAX_FILES`, però per defecte estan amb els valors necessaris per tal que quadrin amb les textures que s'han seleccionat per les interfícies. També crearà l'Slot que segueix contínuament al ratolí i emplenarà `inventari` amb els Items especificats a `itemsInicials`.

A la Figura 5.68 podem veure 9 Slots amb Items diferents. Podem observar que els blocs es renderitzen d'una manera i la flor taronja de l'Slot 4 d'una altra. Això és perquè la vegetació es renderitza agafant directament una part del mapa de textures i renderitzant-lo com si fos un Sprite. Els blocs, però, tindran uns vèrtexs determinats per poder visualitzar-los d'aquesta manera.

Figura 5.68: inventari "petit" amb el bloc de terra seleccionat

El constructor cridarà a la funció `initRenderData`, que començarà creant el *shader* necessari per aquesta part (`VertexBlocItem` i `FragmentBlocItem`). Per poder renderitzar aquests blocs, només necessitem tres cares. No hem de confondre aquests blocs amb els vòxels renderitzats al món: aquells estan optimitzats per poder representar-los en 3D, i aquests no tenen profunditat alguna. Els vèrtexs d'aquests tindran informació de posició, d'UVs i un valor de foscor que tenyirà la cara (que és el que dona la sensació de que són vòxels del món posats en perspectiva). Com sempre que iniciem vèrtexs, hem de cridar les funcions d'OpenGL corresponents (`glGenBuffers`, `glBufferData`, `glVertexAttribPointer`, etc).

5.18.4. `iniciaSprites`

La funció rebrà l'SpriteRenderer i el TextRenderer creats a la classe HUD i afegirà els Sprites necessaris per poder renderitzar l'inventari. Recordem que SpriteRenderer serà el que renderitzi els Sprites visibles que s'afegeixin. Amb el TextRenderer podrem renderitzar un nombre amb la quantitat d'objectes que hi ha en un Slot.

També aprofita i col·loca els Sprites on toca i emplena tot l'inventari (usant la funció `emplenarInventariGran`) amb tots els tipus de vòxels que es puguin fer servir si `creatiu` és `true` (veure Figura 5.69).

Figura 5.69: l'inventari creatiu amb tots els blocs

5.18.5. `render`

El render d'aquesta classe s'encarregarà de cridar al **`render`** de cada Slot visible, és a dir, que tingui un Item. Si no som a dintre de l'inventari creatiu, simplement cridarem **`render`** de cadascun dels Slots de l'inventari petit i **`renderText`** per mostrar la quantitat d'Items.

Si som a dins de l'inventari gran, renderitzarem els dos inventaris. A la Figura 5.69 podem veure que l'inventari petit s'acobla al gran per poder intercanviar entre ells. Si el ratolí es troba a sobre d'algun Slot, es mostrarà una caixa (posant **`visible`** de **`caixeta`** a **`true`**) amb el nom de l'Item que hi ha en aquell Slot (veure Figura 5.70). Si l'Slot que segueix sempre al ratolí té un Item, també el renderitzarem.

Figura 5.70: el ratolí a sobre d'un bloc de síndria

Aquesta funció rebrà la posició del ratolí en pantalla per tal de saber sobre quin Slot es troba i guardar-lo a **`sobre`**.

5.18.6. `obrir`

Obre i tanca l'inventari gran posant **`visible`** a **`true`** o **`false`** respectivament els Slots d'aquest. Un cop dins, mou tot l'inventari petit al mateix lloc del gran per facilitar l'intercanvi de blocs. Si tanquem l'inventari, ocultarem l'Slot que segueix al ratolí i, si tenia algun bloc agafat, el deixarem on estava per mantenir la coherència de l'inventari. També es posarà a `NULL` el punter **`sobre`**.

5.18.7. `agafarItem`

Si GLFW detecta un clic del ratolí i l'inventari està obert, mirarem si el ratolí es troba a sobre d'algun Slot (mirant si `sobre` no és `NULL`).

Si el ratolí està a sobre d'un Slot amb un Item i el ratolí no en té cap, aleshores tota la informació rellevant de l'Item, com l'id i la quantitat que hi ha, passa a l'Slot del ratolí. Visualment pel jugador sembla que el ratolí està agafant l'Slot que hi havia abans, però només s'ha traspassat la informació. També passa el mateix al revés, si el ratolí tenia un Item i l'Slot que hi ha a `sobre` està buit, aleshores es deixa l'Item en aquell Slot. Si passa que tots dos tenen un Item, la informació d'aquests s'intercanvia.

5.18.8. `canviaSeleccionat` i `canviaSeleccionatPer1`

Per poder canviar el bloc actiu a l'inventari petit es faran servir les tecles de l'1 al 9. Així la caixa gris que es veu a sobre del bloc de terra de la Figura 5.68 es mourà cap a l'Slot que es trobi en la tecla corresponent. Per això canviaSeleccionat teletransporta l'Sprite de la caixa allà on toqui.

Amb la roda del ratolí podem fer el mateix: canviaSeleccionatPer1 mou la caixa en un costat o un altre segons si el jugador gira la roda cap a dalt o cap a baix. Si la caixa es passa del límit de l'inventari, torna pel costat contrari.

5.18.9. `afegirItem`

La funció rep l'id de l'Item que es vol afegir i la quantitat. Només podrà afegir un Item nou en cas que l'inventari tingui com a mínim un Slot buit per poder posar-lo.

6. Proves i resultats

6.1. Captures i resultats

Ara que ja hem vist com el motor és capaç de generar i visualitzar un món infinit fet amb vòxels, és hora de comprovar que tot estigui al seu lloc. Les captures que es troben a les figures que hi ha a continuació mostraran com totes les classes del motor s'uneixen per permetre a l'usuari i al jugador la llibertat de poder confeccionar el món com vulguin.

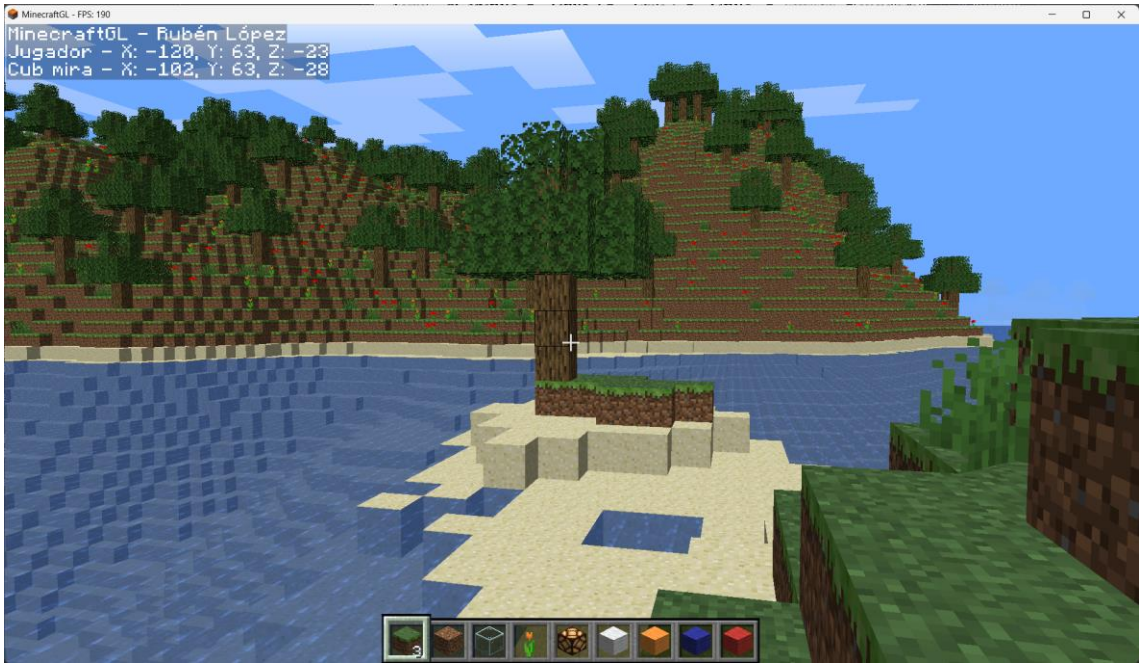


Figura 6.1: un arbre solitari en mig d'una platja

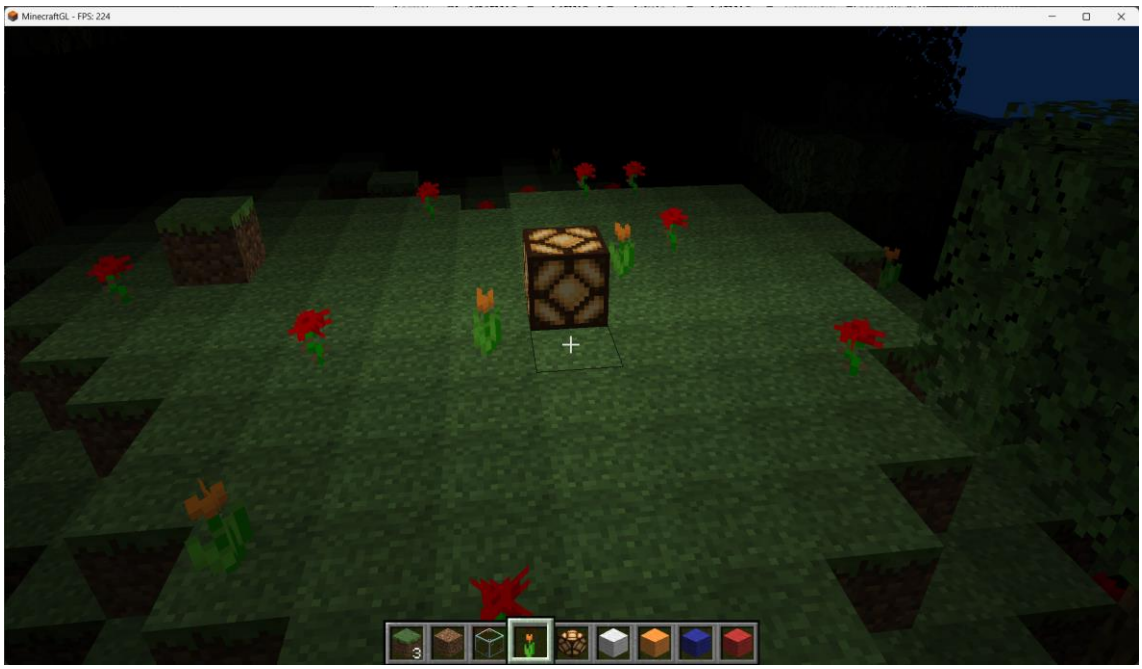


Figura 6.2: llum il·luminant un camp de flors

Des de la Figura 6.3 fins a la Figura 6.7 podem veure que, modificant els punts dels `structs` Soroll i barrejant diferents tipus, podem aconseguir terrenys amb característiques molt peculiars. Per exemple, per obtenir el resultat de la Figura 6.3 hem de comentar el Soroll `erosion` i posar els següents punts a `continentalness: { {-1,120}, {0.5,64}, {0.5,0} }`.

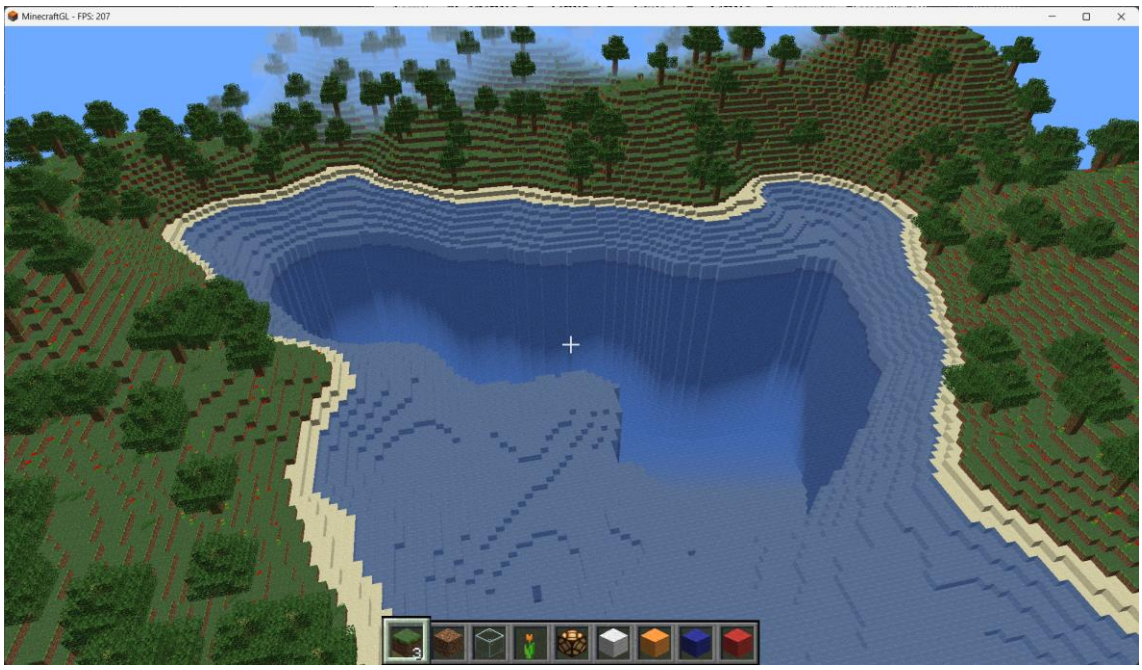


Figura 6.3: una cala amb un fons abissal

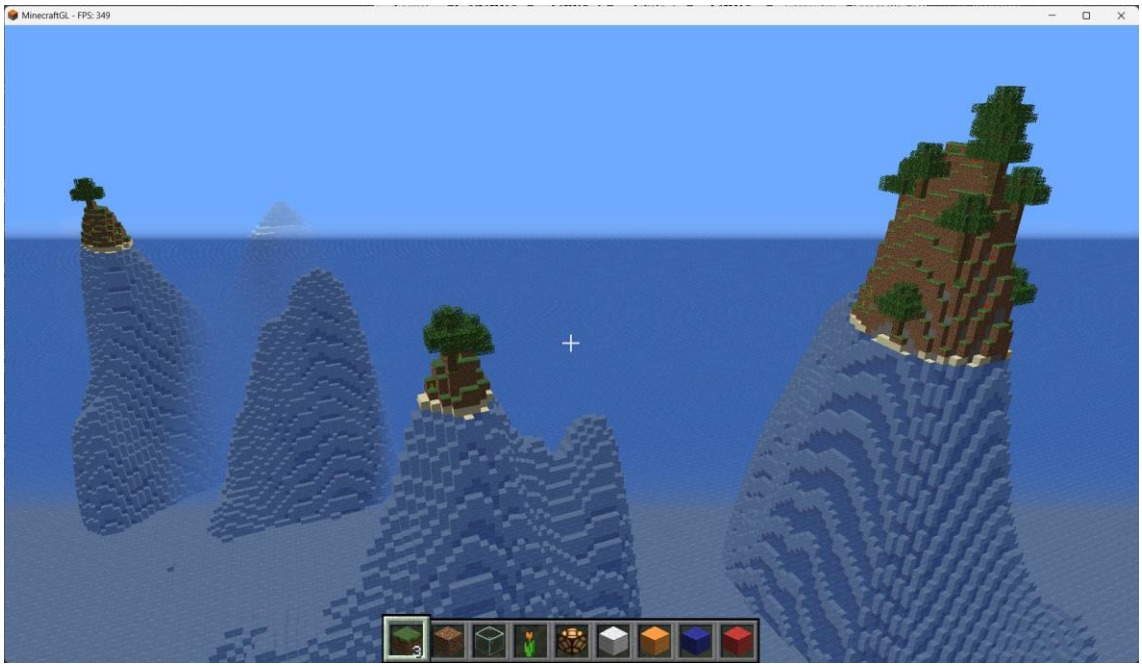


Figura 6.4: un món fet amb illes

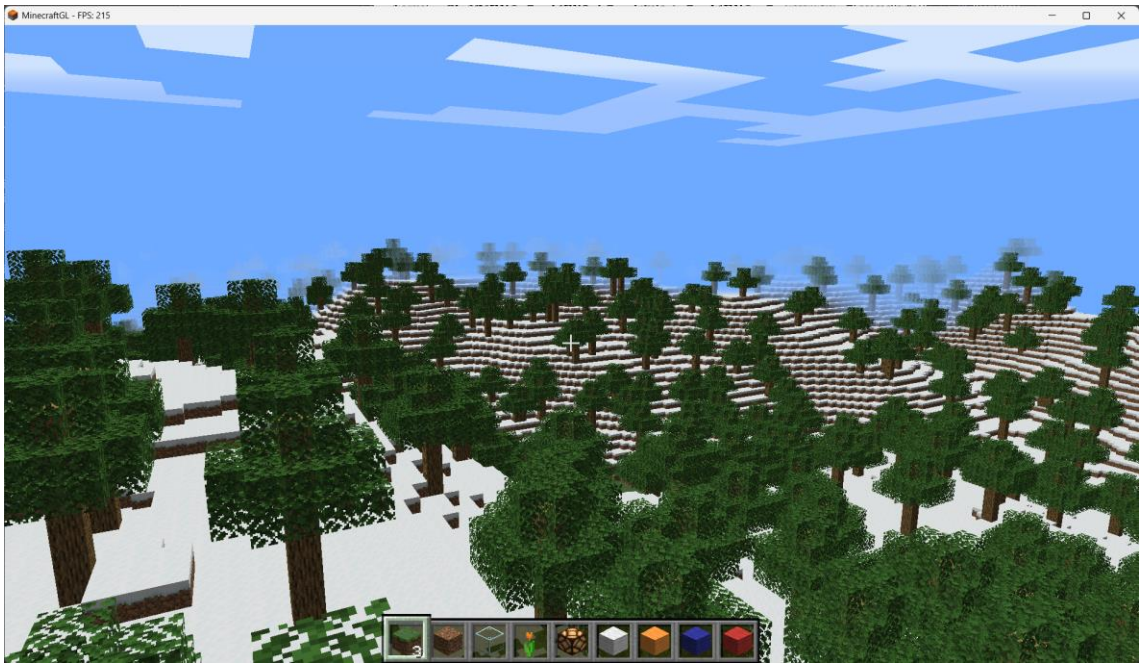


Figura 6.5: un món totalment nevat

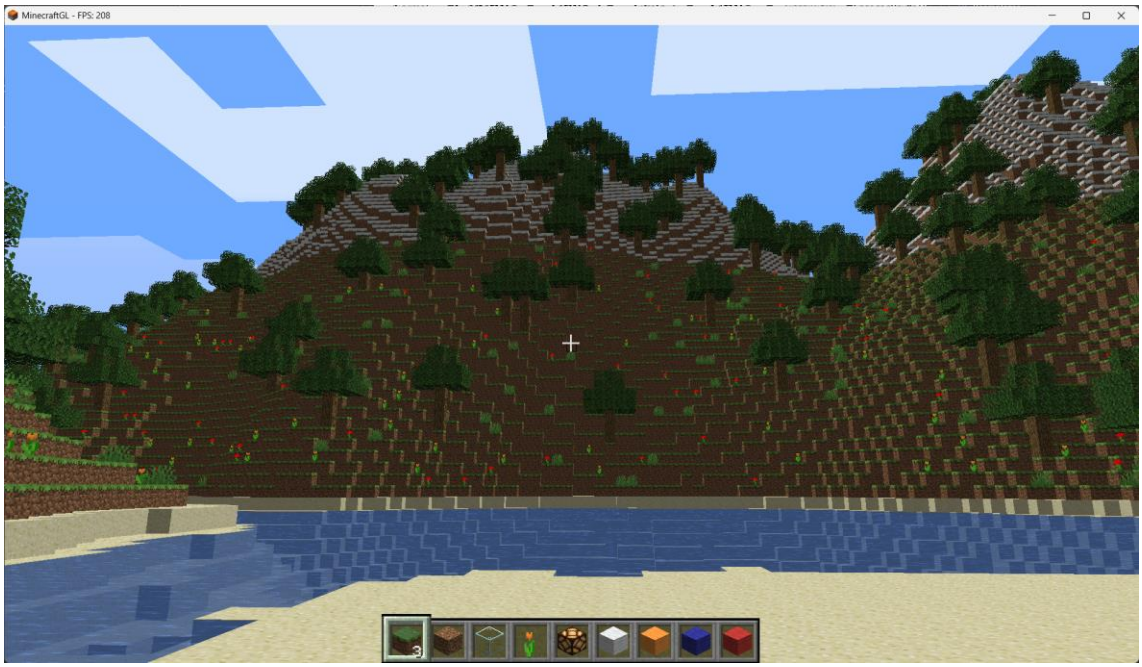


Figura 6.6: una muntanya vista des d'una platja

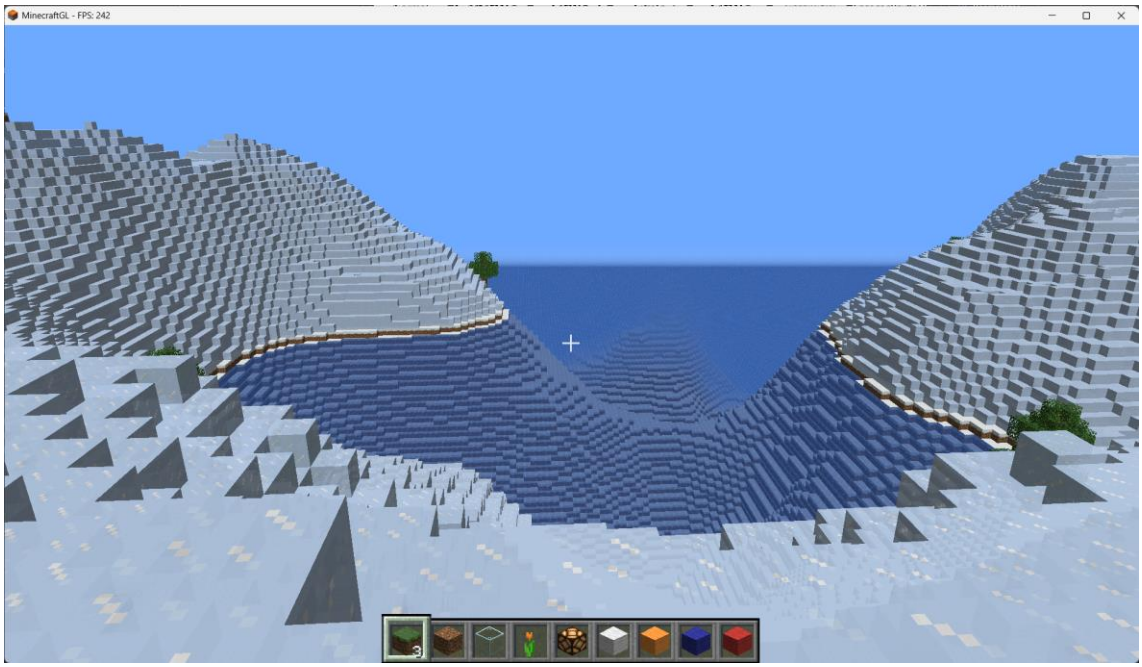


Figura 6.7: un món glacial

Figura 6.8: una casa construïda en una platja

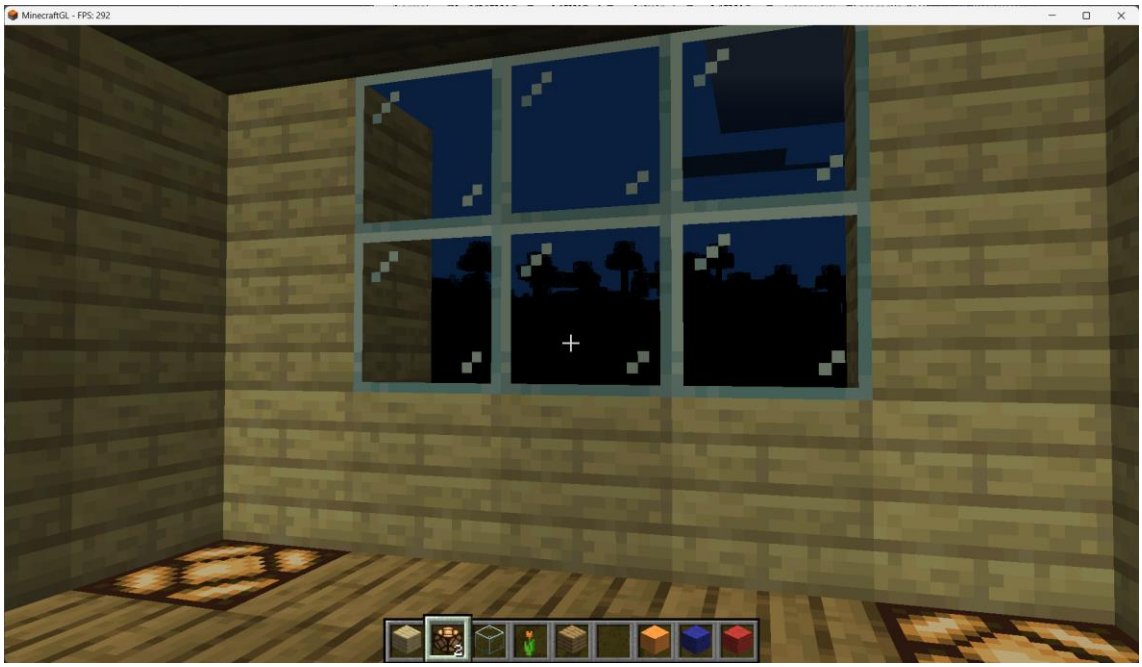


Figura 6.9: interior de la casa de nit

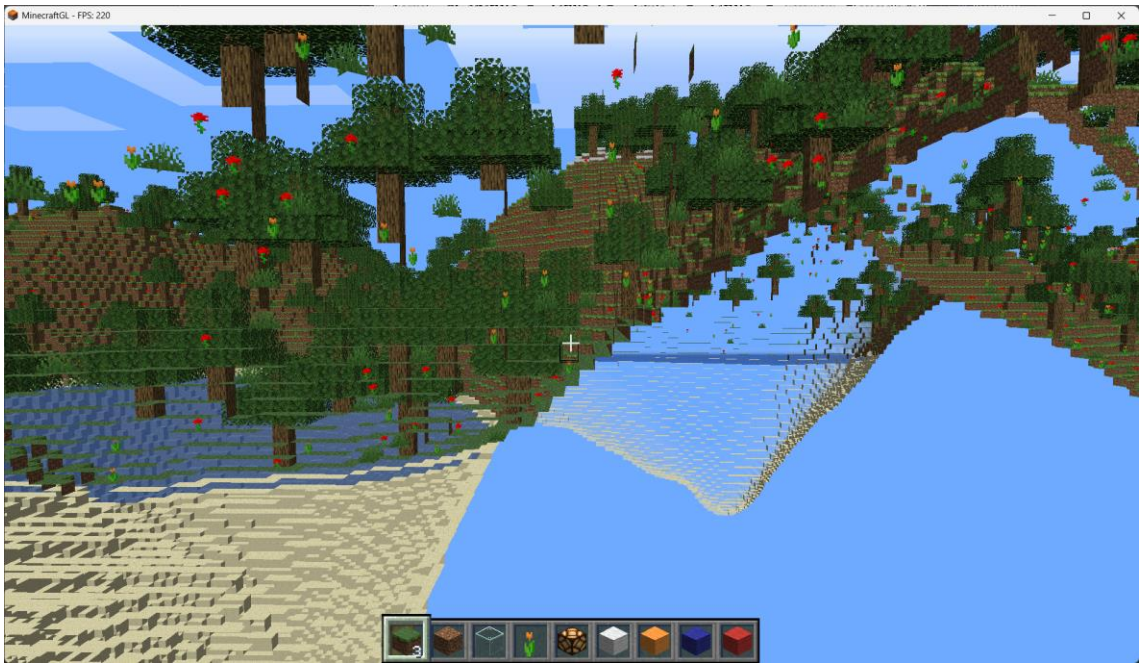


Figura 6.10: el món vist des de dins del terreny. Podem observar que no hi ha cares internes.

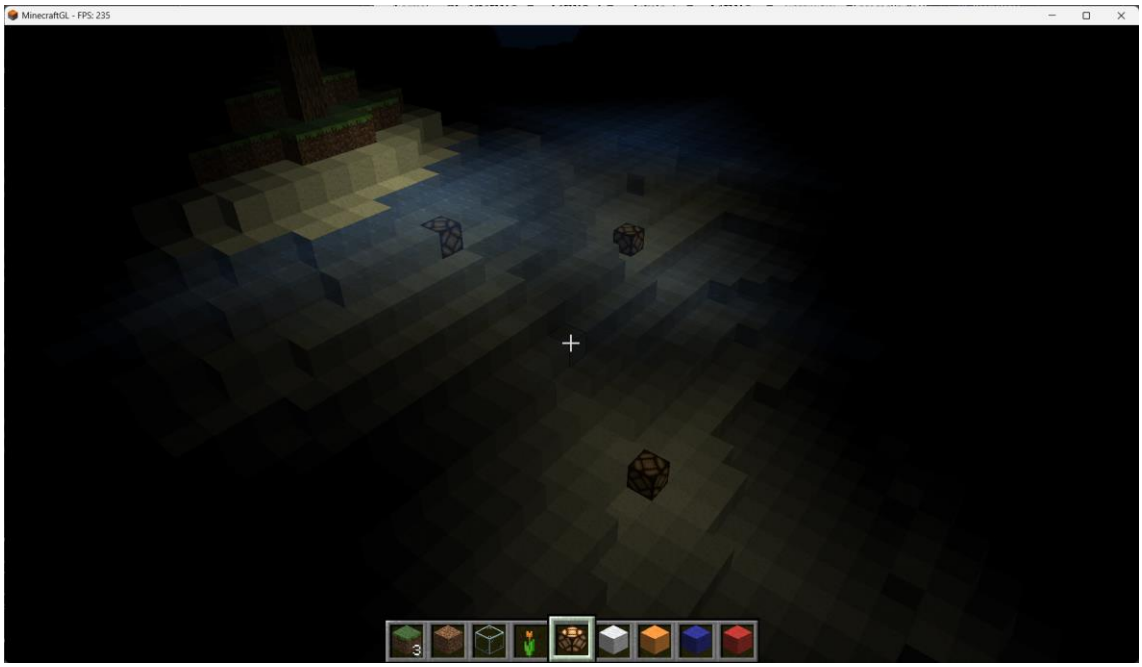


Figura 6.11: llums il·luminant el fons marí

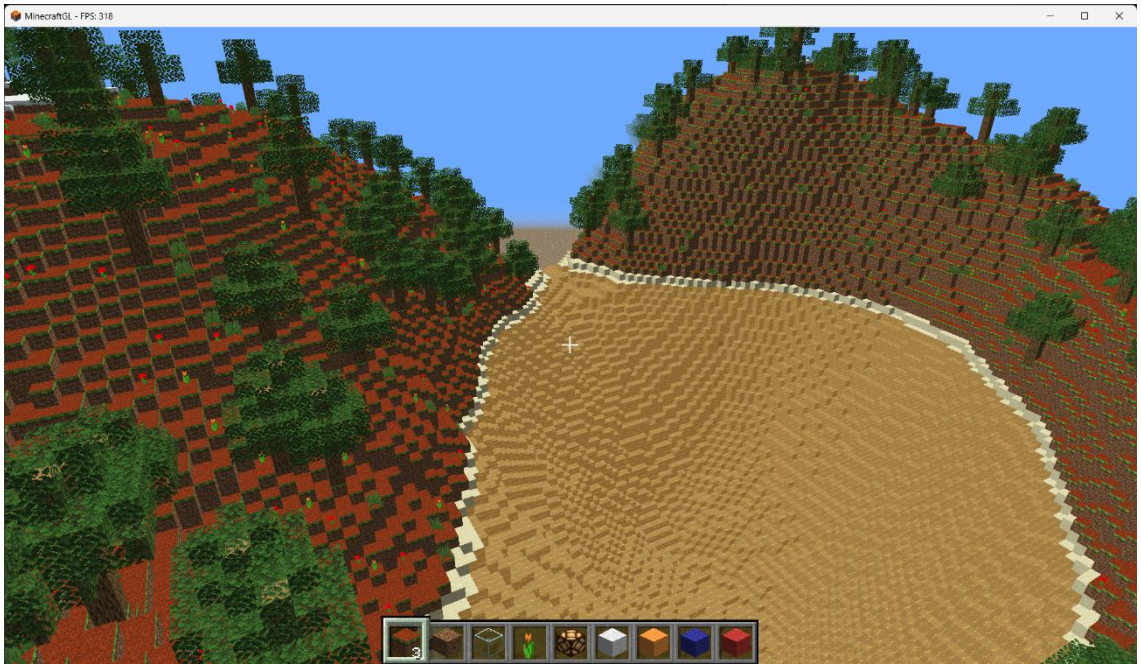


Figura 6.12: el món amb colors canviats

Una de les proves més importants és veure fins a on som capaços d'arribar en aquest món infinit. La majoria de jocs amb vòxels que s'inspiren en *Minecraft* tenen un límit bastant reduït de blocs fins a on el joc permet al jugador arribar. Per això, hem estat volant una estona pel món generat i hem arribat fins a la coordenada (1148, -10702) i el món seguia carregant.

Aquesta prova té diferents significats positius pel motor: primer, que pot seguir generant fins i tot havent arribat a una coordenada tan llunyana sense repetir terreny. Segon, que el món pot generar correctament chunks amb coordenades negatives sense problemes. I tercer, hem demostrat que podem arribar a -10702, per tant el contrari també es pot

demostrar. Això vol dir que el món, com a mínim, va de -10702 a 10702, és a dir, una distancia de 21404 blocs totalment modificables. I si continuéssim avançant, perquè hem parat a aquella coordenada, trobaríem que és encara molt més gran del que sembla (probablement igual de llarg que el mapa de soroll que puguem generar). A la Figura 6.13 podem veure aquesta demostració.

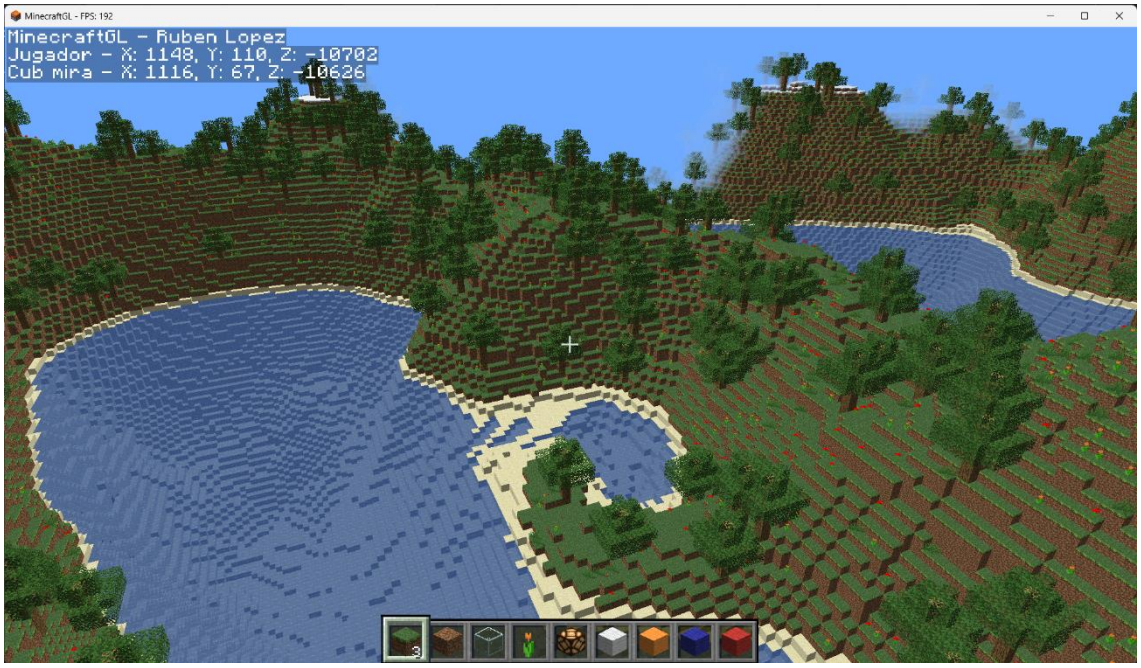


Figura 6.13: el jugador a la posició (1148, -10702)

Com ja s'ha comentat, totes les figures mostrant el joc i les proves s'han realitzat en un món on la llavor sempre és la mateixa (874) per tal que, qualsevol persona que vulgui comprovar la veracitat d'alguna figura, pugui recrear-la al mateix món. Això significa que qualsevol jugador pot anar a les coordenades de la Figura 6.12 o 6.1 i trobarà el mateix paisatge.

6.2. Proves en un entorn diferent

Totes les proves i captures fins al moment s'han fet amb l'ordinador especificat a l'Apartat 2.2.1, però per tenir evidència

Hem utilitzat un portàtil comprat a l'any 2018 amb especificacions bastant més baixes, l'entorn ideal per comprovar si les optimitzacions implementades al motor realment serveixen per permetre córrer el joc en dispositius més antics.

Les especificacions del portàtil utilitzat per les proves que veurem a continuació són les següents:

- **Sistema Operatiu:** Windows 11
- **CPU**: Intel i5-8300H
- **RAM**: 8GB
- **GPU**: NVIDIA GeForce GTX 1050 4GB

El valor per defecte de `DISTANCIA` al SuperChunk és de 15, perquè l'ordinador amb el que es va desenvolupar el projecte podia córrer sense problema aquesta distància, però per aquest portàtil hem de baixar-la a 10. Tot i així, amb 15 els FPS es mantenien estables a 30. Amb 10, aconseguim entre 60 i 70. A més, hem hagut de reduir la mida de la finestra per poder visualitzar millor el joc en una pantalla més petita.

Hem deixat també que el propietari de l'ordinador jugui una mica per poder obtenir captures de pantalla del joc. Mentre es movia pel món i construïa, els FPS es mantenien estables sense gairebé cap caiguda, inclòs quan es generava el món. El mínim aconseguit era aproximadament 50.

De la Figura 6.14 a 6.16 trobem algunes captures fetes durant les proves. A la part superior de la finestra podem veure els FPS que hi havia en el moment de fer-les. Podem observar que el jugador ha pogut fer una construcció i moure's pel món lliurement (podem veure que la Z va arribar a 342).

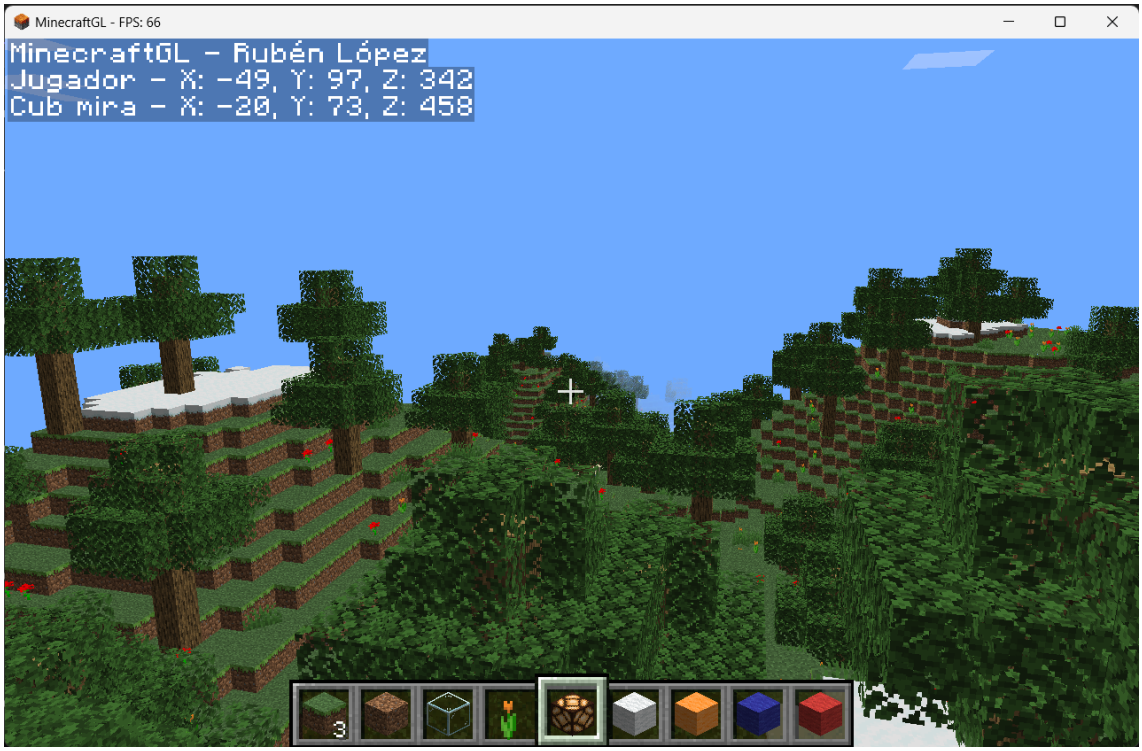


Figura 6.14: el jugador arribant a (-49, 342)

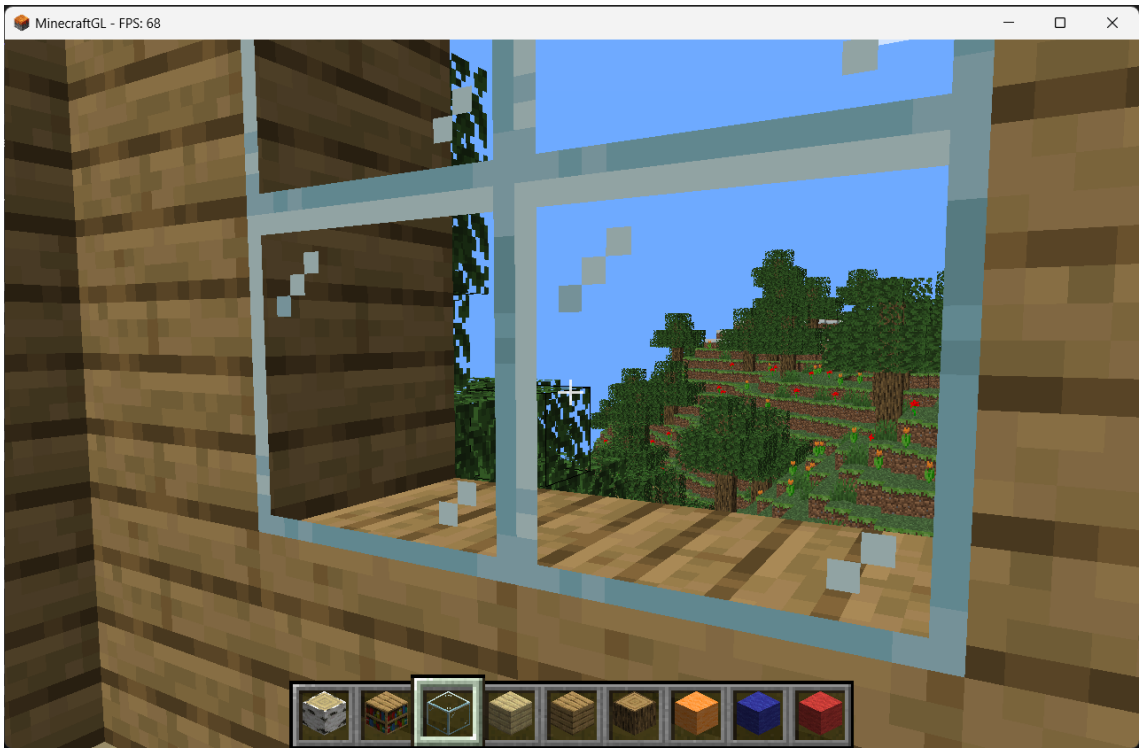


Figura 6.15: l'interior d'una casa

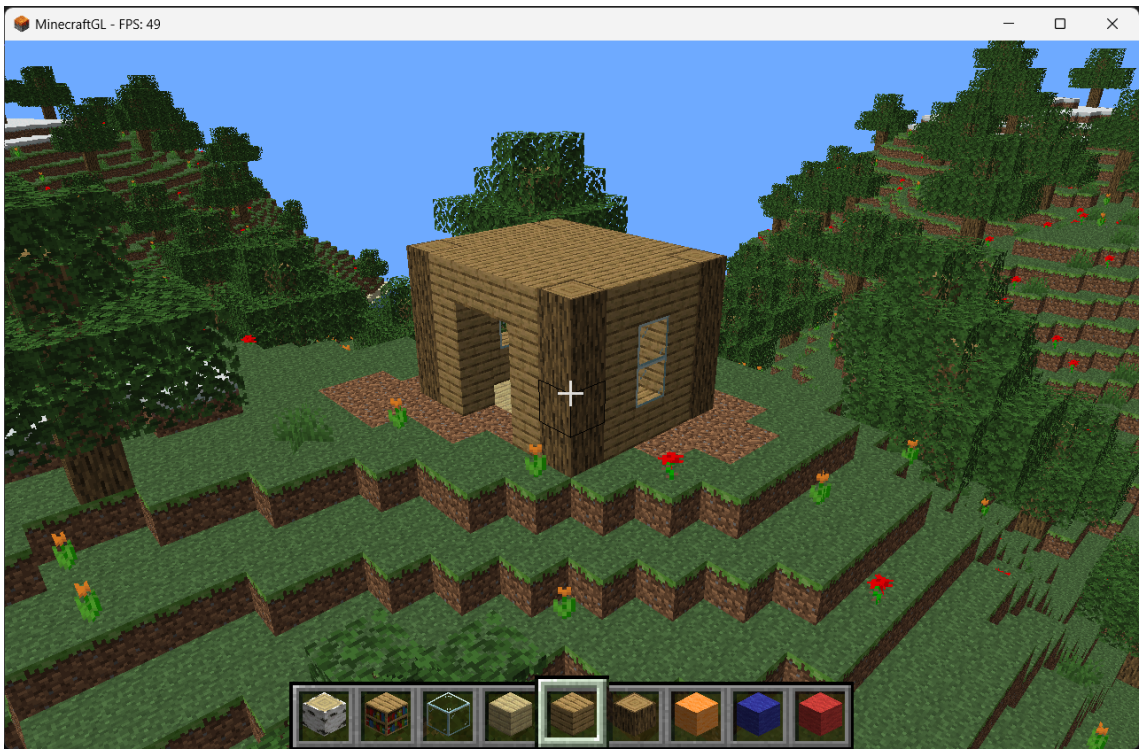


Figura 6.16: l'exterior de la mateixa casa

També hem fet una prova per saber si canviar el nombre de chunks que es gestionen en una funció (recordem, el valor `NCHUNKS` del SuperChunk) fa que canviï el rendiment del joc. A l'ordinador on s'ha desenvolupat el projecte no veiem cap canvi significatiu, potser un o dos FPS, però el segon ordinador sí mostrava un canvi: amb valors més alts s'obtenia una millora d'FPS. La millora potser no és tan significativa, però són 10 FPS

de diferència a la mateixa posició, arribant a 80 FPS. A la Figura 6.17 podem veure aquesta diferència visualitzant la mateixa escena amb `NCHUNKS` igual a 5 per la captura de dalt i 100 per la de baix.

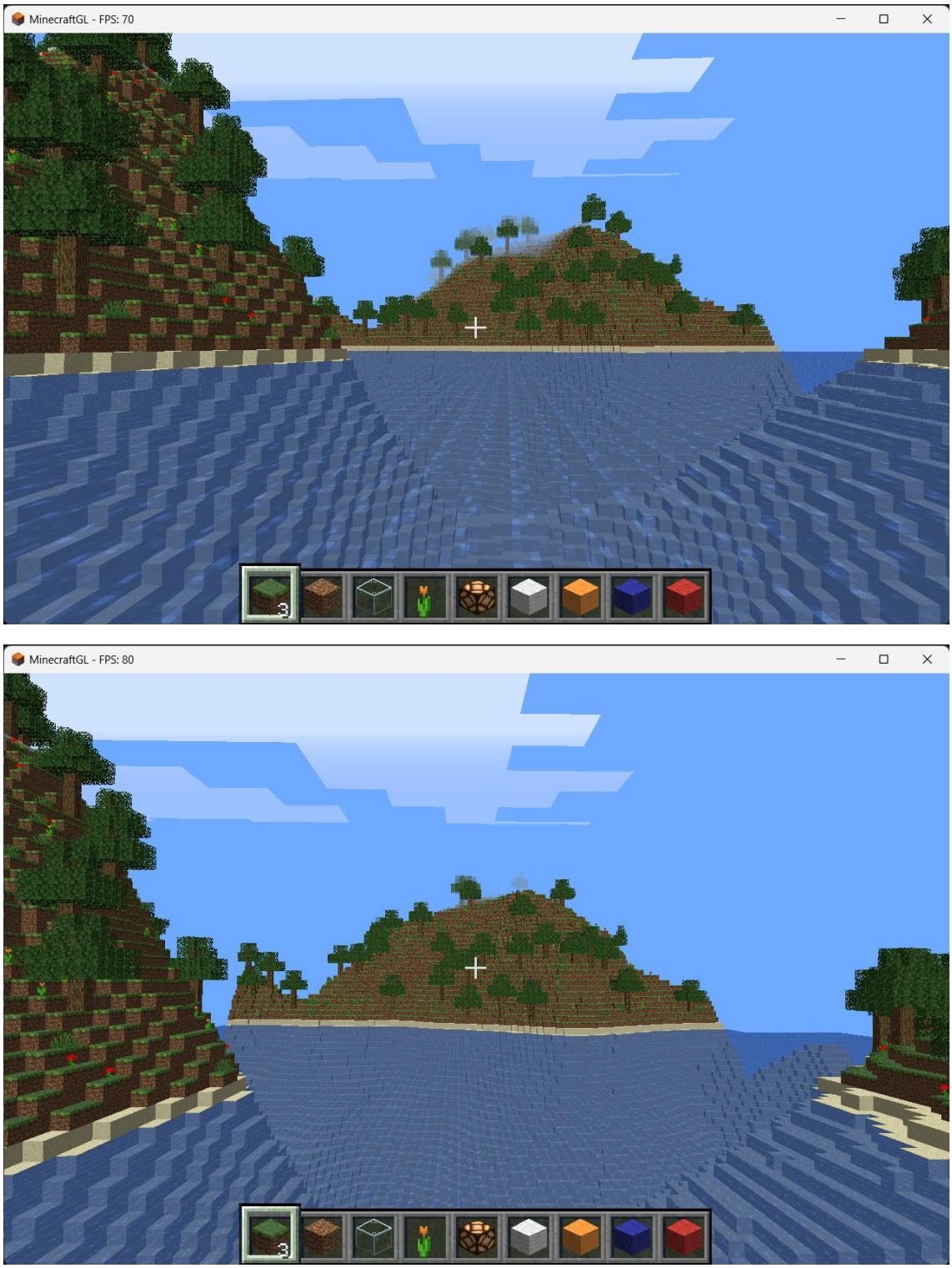


Figura 6.17: comparació d'FPS a la mateixa posició amb NCHUNKS prenent valors diferents

7. Conclusions

Hem desenvolupat un motor que permet la visualització d'un món dinàmic on el jugador és lliure de fer el que vulgui, ja sigui moure's lliurement volant pel món o caminar per ell, a més de construir qualsevol estructura que s'imagini. L'usuari també és capaç d'editar el codi del motor com més li agradi per personalitzar el món guiant-se amb les directrius d'aquest document. Hem creat les nostres pròpies classes per poder visualitzar les interfícies d'usuari d'una manera ràpida i senzilla i hem aplicat tècniques d'optimització que han fet del rendiment una part destacable del motor. Podem dir amb seguretat que els objectius del projecte han estat assolits i que hem superat a la majoria de còpies del joc original, tot i que encara hi ha lloc per continuar millorant el motor i afegint més mecàniques.

A l'enllaç a GitHub que es troba al Capítol 10 es pot veure tot el treball que s'ha fet i com ha anat evolucionant, així com tot el codi pròpiament documentat.

Personalment, crec que aquesta experiència m'ha ajudat a posar-me a prova a mi i als coneixements que he adquirit al llarg del curs. He aprofundit molt en la programació gràfica i sobretot m'ha permès dissenyar el meu propi motor, que es tracta d'un treball que al principi no semblava gens fàcil i ha comportat incomptables hores de treball. Crec que aquest projecte pot avançar encara més i em pot ajudar a demostrar les meves aptituds.

A continuació es presenta el diagrama de Gantt actualitzat amb com realment s'ha desenvolupat el projecte. En verd fosc, les tasques que s'han pogut seguir, en verd més clar les que s'han hagut d'allargar i blau les que s'han endarrerit per prioritzar altres tasques. En general s'ha seguit bé el diagrama plantejat, tot i que algunes tasques s'han trigat més en completar.

Tasca	Octubre	Novembre	Desembre	Gener	Febrer	Març	Abril	Maig	Juny	Juliol	Agost
Investigació											
Pràctica											
Inici											
Conjunt de *chunks*											
Modificar terreny											
Texturització											
Il·luminació											
Base de dades											
Món infinit											
Generació vegetació											
Generació arbres											
Generació de terreny											
Text											
Inventari											
Informació *debug*											
Frustum culling											
Col·lisions											
Decoracions											
Documentació											

8. Treball futur

El projecte ara mateix permet al jugador entrar en un mode creatiu i construir el que vulgui en un món infinit, però encara hi ha espai per més mecàniques i opcions de visualització:

- Mode supervivència: el jugador recollirà els seus propis recursos per poder fabricar els blocs en comptes de tenir-los tots ja a l'inventari.
- Ja que tenim una classe Inventari, aprofitar-la i crear inventaris externs.
- Programar *shaders* més realistes.
- Acabar de perfeccionar la il·luminació i permetre una il·luminació global.
- Afegir animals i altres NPCs per tal que el jugador pugui interactuar amb altres éssers vius al món.

Totes aquestes millores són totalment opcionals, ja que el motor ara per ara funciona correctament i té un bon rendiment, però li poden donar encara més vida i pot ser una bona oportunitat per continuar treballant.

9. Bibliografia

1. De Vries, J. (2020). *Learn OpenGL: Learn Modern OpenGL Graphics Programming in a Step-by-step Fashion.* https://learnopengl.com/book/book_pdf.pdf

2. De Vries, J. (s. d.) *Learn OpenGL, extensive tutorial resource for learning modern OpenGL.* https://learnopengl.com/

3. Khronos Group. (s. d.). *OpenGL - the industry standard for high performance graphics.* https://www.opengl.org/

4. Khronos Group. (s. d.). *OpenGL Wiki.* https://www.khronos.org/opengl/wiki/

5. *GLFW* (s. d.). https://www.glfw.org/

6. G-Truc. (14 abril de 2010). *GitHub - G-Truc/GLM: OpenGL Mathematics (GLM).* GitHub. https://github.com/g-truc/glm

7. Auburn. (28 de març de 2016). *GitHub - Auburn/FastNoiseLite: Fast Portable Noise Library.* GitHub. https://github.com/Auburn/FastNoiseLite/

8. Lohmann, N. (2013). *JSON for Modern C++ - JSON for Modern C++.* https://json.nlohmann.me/

9. *The FreeType project.* (s. d.). https://freetype.org/index.html

10. Fandom. (s. d.*). Minecraft Wiki – the ultimate resource for minecraft.* https://minecraft.fandom.com/wiki/Minecraft_Wiki

11. Brendan, L. K. (30 abril de 2013) *Swept AABB Collision Detection and Response.* https://www.gamedev.net/tutorials/programming/general-and-gameplay-programming/swept-aabb-collision-detection-and-response-r3084/

12. Henrik Kniberg. (6 de febrer de 2022). *Minecraft terrain generation in a nutshell* [Vídeo]. YouTube. https://www.youtube.com/watch?v=CSa5O6knuwI

13. Benjamin. (s. d.) *Fast Flood Fill Lighting in a Blocky Voxel Game.* https://www.seedofandromeda.com/blogs/29-fast-flood-fill-lighting-in-a-blocky-voxel-game-pt-1/

10. Manual d'usuari

10.1. Controls del joc

El jugador pot realitzar les següents accions prement les tecles corresponents:

- **WASD**: moure's pel món.
- **Control esquerre**: si es manté polsat, córrer.
- **Shift esquerre**: baixar (en mode `ESPECTADOR`).
- **Barra espaiadora**: pujar o saltar.
- **ESC**: surt de l'inventari o tanca el joc.
- **C**: alterna mode `ESPECTADOR` i mode `CREATIU`.
- **E**: obrir l'inventari. El jugador pot reorganitzar l'inventari com li sembli i pot agrupar blocs del mateix tipus. Els blocs que hi ha a baix del tot són els que pot fer servir per construir.
- **Roda del ratolí** o **nombres de l'1 al 9**: canviar bloc actual.
- **Clic dret**: col·locar bloc actual al món.
- **Clic esquerre**: destruir el bloc que hi ha a la mira.

A més, hi ha una sèrie de tecles especials que fan accions per canviar com es renderitza el món:

- **F1**: alterna la visibilitat del HUD.
- **F2**: activa i desactiva el *culling* d'OpenGL.
- **F3**: alterna la visibilitat del text *debug*.
- **F4**: activa i desactiva el límit d'FPS.
- **F6**: activa i desactiva el *frustum culling*.
- **F7**: alterna entre dia i nit.
- **F10**: activa i desactiva el mode *wireframe*.

10.2. Instal·lació del motor

En realitat no fa falta instal·lar res, sinó que s'ha de descarregar o clonar el repositori que es troba a *https://github.com/RubyEnoshima/MinecraftGL*. Per poder iniciar el joc es necessita tenir Visual Studio Community 2022 instal·lat, però un cop descarregat el projecte només s'ha d'obrir amb el Visual Studio i compilar-lo per jugar. Es pot modificar tot el que es vulgui del motor i personalitzar-lo com ja s'ha anat explicant al llarg del document.