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

Chen, Pablo; Martí Godia, Enric, dir. Disseny i implementació d'un joc de
puzle tipus Sudoku per a aprendre els númerosen Braille. 2022. (958 Enginyeria
Informàtica)

This version is available at <https://ddd.uab.cat/record/264193>

under the terms of the  license

Disseny i implementació d'un joc de puzzle tipus Sudoku per a aprendre els números en braille

Pablo Chen

Resum– En aquest article s'explica el disseny, implementació i test d'una primera versió d'un sudoku 3D utilitzant l'API d'OpenGL modern. El videojoc consisteix en un tauler en 3D on es pot col·locar fitxes amb números en braille respectant les regles d'un sudoku clàssic. El joc té com a finalitat ajudar als nens a aprendre els números en braille i per això s'ha implementat una sèrie de característiques que ajuden a la memorització del braille: visualització de números en braille i en números normals, pistes per a solucionar el trencaclosques, reproducció d'àudio en fer clic, entre d'altres. Per a una major disponibilitat, el joc s'ha desenvolupat per a Linux i Windows.

Paraules clau– Sudoku, braille, joc, OpenGL, gràfics 3D, multiplataforma.

Abstract– This article explains the design, implementation, and testing of an early version of a 3D sudoku using the modern OpenGL API. The video game consists of a 3D board where you can place tiles with numbers in braille respecting the rules of a classic sudoku. The aim of the game is to help children learn numbers in braille, which is why a series of features have been implemented that help them to memorize braille: visualization of numbers in braille and normal numbers, clues to solve the puzzle, audio playback on click, among others. For greater availability, the game has been developed for Linux and Windows.

Keywords– Sudoku, braille, game, OpenGL, 3D graphics, multiplatform.



1 INTRODUCCIÓ

EN el darrers anys, la tecnologia ha facilitat l'accés a jocs a persones invidents que poden desenvolupar capacitats en jocs com els escacs. El Sudoku és un trencaclosques on el jugador tracta d'omplir una graella parcialment emplenada de 9×9 caselles dividida en subgraelles de 3×3 (anomenades regions o blocs) amb un set de números de manera que cada fila, columna i subgraella contingui els números només una vegada. El sudoku modern va ser popularitzat en el 2004 quan Wayne Gould va desenvolupar un programa informàtic per produir trencaclosques únics ràpidament. El braille és un sistema d'escriptura i lectura tàctil basat en 6 punts disposats en una

graella de 3×3 , és utilitzat molt per a persones cegues o amb visió reduïda.

En l'actualitat, podem trobar sudokus en molts llocs: en els diaris, en llibres de passatemps, en aplicacions per al mòbil o ordinador. Curiosament, no s'ha pogut trobar un sudoku adaptat a persones invidents en braille o un sudoku que permeti aprendre els números en braille, només s'ha trobat un sudoku físic que funciona amb fitxes i un tauler[2], i això ens ha semblat una manca important. I el fet que estic interessat en el funcionament dels gràfics 3D i de la programació dels jocs ha motivat la realització d'aquest projecte.

L'objectiu principal del projecte consisteix a dissenyar i desenvolupar un joc en 3D des de zero, concretament un sudoku. El joc tracta d'una graella de 9×9 fitxes on es visualitzarà el valor numèric amb l'alfabet braille. Per una altra banda, mitjançant el teclat i el ratolí, el jugador podrà interactuar amb les caselles i les fitxes per veure el valor numèric de la fitxa. Es volen aconseguir els següents objectius i funcionalitats:

- Desenvolupar un model en 3D del sudoku.

• E-mail de contacte: pablo.chen@uab.cat

• Menció realitzada: Computació

• Treball tutoritzat per: Enric Martí Godia (Dept. Ciències de Computació)

• Curs 2021/22

- Incorporar ajudes per aprendre els números o l'alfabet en braille.
- Incorporar ajudes per la resolució del sudoku.
- Desenvolupar un joc per a Windows i Linux.

Respecte a les eines de desenvolupament, s'ha utilitzat principalment, l'entorn de desenvolupament multiplataforma CLion[5] per C/C++ per poder desenvolupar el joc tant en Windows com en Linux, RenderDoc[22] per a diagnosticar problemes de renderització i CMake[6] per al sistema de compilació multiplataforma.

Pel que fa a les llibreries, s'ha utilitzat les següents:

- ImGUI[17] És una llibreria que permet inserir interfícies d'usuari amb el pipeline d'OpenGL, s'ha utilitzat la versió amb viewports (branca anomenada docking).
- GLFW[1] Per gestionar la inicialització de la finestra i obtenir els estats dels perifèrics.
- OpenGL[19] API interactuar amb la GPU.
- glBinding[12] És una llibreria que permet carregar les funcions d'OpenGL. La llibreria utilitza característiques pròpies del llenguatge C++ que faciliten la detecció d'errors.
- GLM[13] És llibreria matemàtica especialment desenvolupada per OpenGL.

Els sistemes operatius amb els quals s'ha compilat i executat el joc són Windows 10 i Fedora 35.

Per a poder optimitzar al màxim els recursos, s'ha seguit una metodologia àgil, celebrant revisions periòdiques generals en finalitzar cada sprint on es van veure els resultats de la fase anterior, es van valorar els avenços i es va planificar el següent sprint. S'ha utilitzat una tècnica anomenada Kanban, com es pot veure en la figura 1, la tècnica tracta de dividir les tasques en subtasques i col·locar cada tasca segons el seu progrés (to do, in progress, done). Aquesta metodologia és ideal pel desenvolupament en solitari de projectes petits per la seva simplicitat; redueix la gestió de tasques a una simple llista. En particular, s'ha fet servir KanbanFlow[16] per gestionar les subtasques.

Finalment, per a gestionar el codi font, s'ha creat un repositori a GitHub[3] on s'ha anat pujant a ells els avenços fets utilitzant git[11].

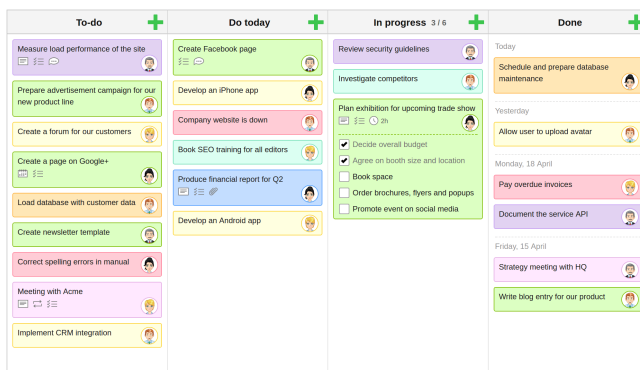


Fig. 1: Un tauler Kanban.

La planificació del projecte té una durada aproximada de tres mesos i consta de tres parts el disseny i la implementació de les classes que serviran per implementar el joc.

- **Disseny de les classes** En aquesta part s'han dissenyat les classes de renderització i els altres components del joc. També s'han dissenyat les diferents pantalles del joc: la pantalla d'inici, els controls del sudoku.

- **Implementació** Implementar les classes de renderització amb OpenGL (cub, esfera, pla), implementar les rutines de tractament de les accions dels perifèrics. A més d'implementar el *gameplay*.

- **Tests** Finalment, al final del projecte s'ha fet un *user-testing* per comprovar que el funcionament del joc és correcte i s'han fet testsos durant la fase d'implementació.

2 REQUISITS FUNCIONALS

El joc haurà d'implementar les següents funcionalitats:

- Guardar i carregar partides a partir d'un fitxer de text.
- Generar un sudoku aleatori d'una determinada dificultat.
- Donar pistes en la resolució del sudoku de manera interactiva.
- Aconseguir l'execució en diverses plataformes, principalment Windows i Linux.

2.1 Requisits de hardware

Per a executar el joc, el sistema ha de comptar amb una GPU, pot set tant una GPU integrada com dedicada, que suporti OpenGL mínim v3.3 o major.

3 DESENVOLUPAMENT

Per desenvolupar un joc, normalment, s'utilitza un motor de joc que prové les eines necessàries per al ràpid desenvolupament d'aquest, com per exemple Unity[27] o Godot[14].

3.1 Inicialització

Abans de res, necessitem una finestra i un context OpenGL per a poder renderitzar els objectes a la pantalla. Hi ha diverses funcions que s'encarreguen de la creació i de la destrucció del context OpenGL a més de la instal·lació dels *callback* dels esdeveniments per als perifèrics.

3.2 Sistema de renderització

El sistema de renderització, com es pot veure en la figura2, està format per dues classes (Object, Shader i derivats del Object) i exposa una interfície la qual permet dibuixar la figura que es desitja que permet dibuixar en pantalla.

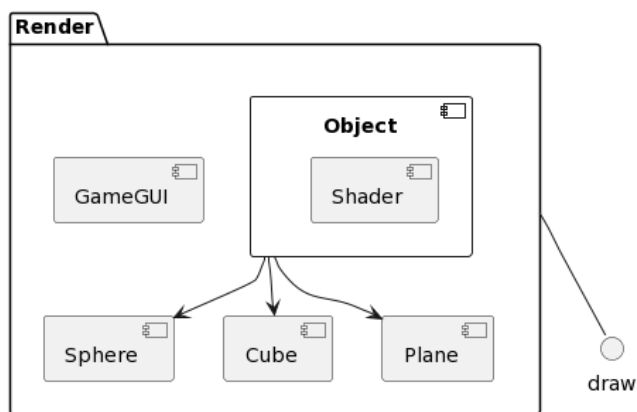


Fig. 2: Diagrama de components del sistema de renderització.

3.2.1 Classe Object

El procés de renderitzat té tres fases: la definició dels vèrtexs (o *quads* si inclou altres dades a més de la posició), la transmissió a la placa gràfica i el dibuix dels objectes. En la primera fase, es defineix en els vèrtexs de l'objecte i l'ordre dels vèrtexs a dibuixar, normalment utilitzant *buffers*. Després, és necessari comunicar aquestes dades a la GPU mitjançant les API que proporciona OpenGL[19]. Finalment, cal activar el programa *shader* i cridar una de les funcions de dibuix (les rutines més simples són *glDrawArrays* o *glDrawElements*).

La primera fase és específica a la forma que es vol dibuixar un cub, una esfera, mentre que les altres fases són independents de la forma. Això ens permet crear una classe abstracta que encapsuli les dues fases. La classe *object* és la classe base per a qualsevol objecte que es dibuixa a pantalla i conté dos mètodes principals: *setupVao* que serveix per crear els objectes i *buffers* necessaris per les dades i *draw* que serveix per dibuixar en pantalla. La figura 3 mostra el resultat final dels objectes.

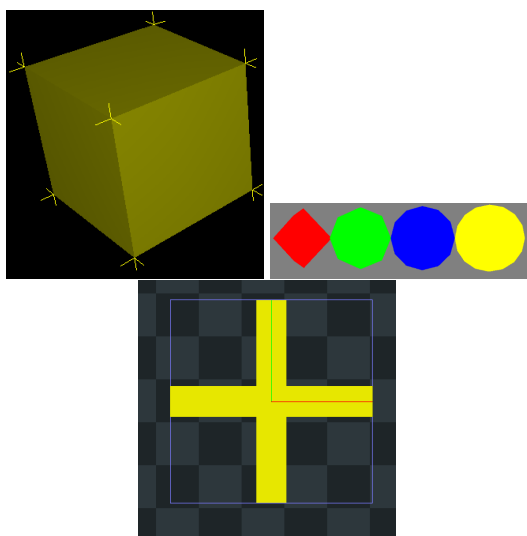


Fig. 3: Renderització dels diferents objectes: un cub amb il·luminació i les seves normals, esferes amb diferents paràmetres i la mira visualitzada amb RenderDoc.

3.2.2 Shaders

Els *shaders* són programes informàtics que s'executen per la GPU i en permet modificar l'escena 3D durant el renderitzat. Els programes *shaders* s'escriuen amb un llenguatge específic per a l'API que s'utilitza pel renderitzat; si s'utilitza l'API Direct3D[7] pel renderitzat s'ha d'utilitzar HLSL[29] (*High Level Shader Language*). En el nostre cas utilitzem GLSL[26] (*OpenGL Shading Language*) per a OpenGL. Hi ha diversos tipus de *shaders* que permeten modificar diferents característiques de l'escena: els vèrtexs amb el *vertex shader*, el color o la il·luminació amb el *fragment shader* o la forma del objecte amb el *geometry shader*.

La creació del *shader* consisteix en tres parts (igual que la creació d'un programa per a la CPU): codificació, compilació i enllaçat (*linking*). És un procés molt repetitiu i per això s'ha desenvolupat una classe per gestionar la creació del programa *shader*.

La classe *shader*, a partir del codi font, genera el programa *shader* i traca els errors que sorgeixen en les diferents parts del procés (errors de sintaxi i problemes d'enllaçat entre els diferents *shaders*). A més a més, els *shaders* permeten rebre dades de l'aplicació mitjançant variables anomenades *uniforms*, el procés també és molt repetitiu i és convenient tenir uns mètodes que permetin assignar les dades de manera més senzilla.

3.2.3 Cub

La renderització d'un cub (o de qualsevol objecte) es fa a partir de triangles, primer es defineix la posició dels vèrtexs dels triangles que el conformen, a més de les normals i de les coordenades de la textura i s'utilitza el mètode *setupVao* de la classe base per enviar les dades dels vèrtexs a la GPU.

3.2.4 Esfera

Igual que el cub, cal definir les posicions dels vèrtexs i les normals per a construir l'esfera, però pel fet que una esfera té una superfície llisa i té infinits punts, necessitem dividir la superfície en diferents sectors i estacs més petits de manera que quan fem infinites particions, obtenim una esfera perfecta.

La primera opció per generar els punts i les normals és utilitzar Blender o qualsevol altre programari per modelar objectes 3D per modelar una esfera llisa i exportar el model com a fitxer *wavefront* per més tard carregar el model al joc.

La segona opció és generar les els vertex i normals de forma algorítmica utilitzant les equacions de l'esfera [21].

El primer mètode comporta desenvolupar un carregador de fitxers *obj* si es vol implementar a mà o utilitzar una llibreria com ara Assimp[18] que podria ser útil si es vol crear objectes més complexos per al joc (per exemple: un model d'una mà per agafar les fitxes). Però en aquest cas s'ha optat en utilitzar el mètode algorítmic, ja que és més simple d'implementar.

3.2.5 Pla

Aquest objecte l'utilitzem per seleccionar el pla que es vol en la funció *glUnproject* per convertir la posició del ratolí

en la finestra a coordenades món. Si no s'utilitza un pla, la posició del món que s'obtingria seria la del pla *zFar*.

3.2.6 Mira

A diferència dels objectes anteriors, la mira és un objecte 2D pel fet que forma part de la interfície i per això la classe té un mètode per renderitzar especial que desactiva la il·luminació i fa l'ús de la projecció ortogràfica per col·locar l'objecte en coordenades de la finestra a més d'un *shader* no té implementat la matriu *lookAt* per la càmera com es pot veure en la figura 4.

```
// vertex uniform
basicShader.setMat4("uModel", glm::value_ptr(transform));
basicShader.setMat4("uView", glm::value_ptr(camera->getViewMatrix()));
basicShader.setMat4("uProjection", glm::value_ptr(camera->getProjectionMatrix()));

// vertex shader
void main(){
    gl_Position = uProjection * uView * uModel * vec4(aPosition, 1);
}
```

Fig. 4: Codi *shader* per a objectes 3D (a dalt) i 2D (a baix)

3.2.7 Fitxa del sudoku

Una vegada tenim implementat l'esfera i cub, es pot començar el modelatge de l'entitat del joc, és necessari un cub i sis esferes tot i que el braille només s'utilitza les quatre primeres esferes per representar els números del 0 al 9. El cos de la fitxa és un cub amb 1 unitat d'amplada i alçada i 0,375 unitats de profunditat, les esferes tenen un radi de 0,1 unitats.

Un problema que s'ha trobat en implementar aquesta classe és el renderitzat de les esferes en funció del valor de la fitxa. La primera opció és escriure el codi amb declaracions condicionals de manera que si el valor numèric de la fitxa és 1, dibuixar la primera esfera, si el valor és 2 dibuixar la primera i la tercera i així amb els altres valors.

Com es pot veure, si s'escriu d'aquesta manera, el codi serà molt difícil de mantenir o, si es vol més tard, implementar noves funcions com ara ampliar la classe perquè incorpori tot l'alfabet. És per això que s'ha optat per utilitzar una estructura anomenada *look up table* que és bàsicament un vector o una *hash table* que mapeja el valor de la fitxa i l'índex de les esferes a dibuixar.

3.3 Classes d'esdeveniments

En un principi, aquesta classe estava implementada com una cua de missatges amb classes polimòrfiques. Els esdeveniments eren classes que heretaven d'una classe principal comú a tots els esdeveniments que es produïen.

Els esdeveniments són introduïts a la cua de missatges mitjançant *callbacks* que s'instal·len al sistema d'esdeveniment de GLFW. El *callback* crea, de manera dinàmica, la classe corresponent al missatge i ho converteix en la classe base per a poder afegir-lo a la cua, finalment, els missatges són propagats mitjançant un bucle que s'executa al final de cada *frame*.

Per rebre els missatges, els mòduls han de subscriure's al sistema d'esdeveniments mitjançant *callbacks* i convertir els missatges a esdeveniment.

Aquesta implementació té diversos problemes o inconvenients: la llibreria GLFW només envia un missatge per

```
const std::vector<std::vector<int>> NUMBER_TO_BRAILLE = {
    {1, 2, 3}, // 0
    {0}, // 1
    {0, 2}, // 2
    {0, 1}, // 3
    {0, 1, 3}, // 4
    {0, 3}, // 5
    {0, 1, 2}, // 6
    {0, 1, 2, 3}, // 7
    {0, 2, 3}, // 8
    {1, 2}, // 9
    {0, 1, 2, 3, 4, 5} // 10 draw everything
};
```

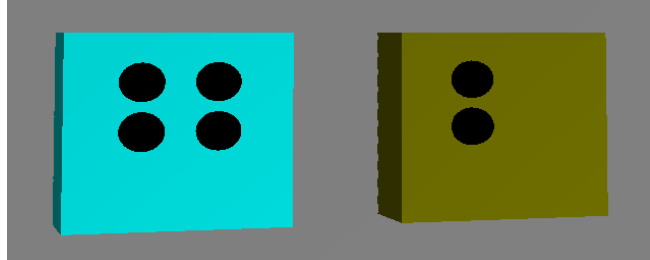


Fig. 5: *Look up table* indexat pel valor numèric de la fitxa i la fitxa amb el 7 i el 2 en braille.

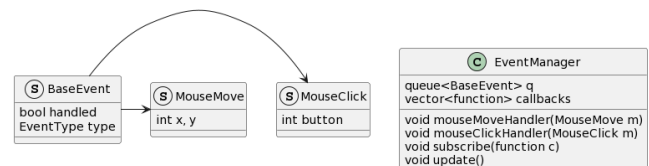


Fig. 6: Diagrama UML del sistema d'esdeveniment inicial.

esdeveniment, és a dir si es pressiona una tecla i es deixa premuda, el sistema només envia un missatge que serà vàlid per a un *frame*. Això vol dir que si es deixa pressionada, no s'enviaran els missatges que toquen; com es pot observar en la figura 7, en el *frame* 1, el *KeyPress* no es transmet. Aquest problema es pot solucionar guardant el missatge que s'ha rebut de manera local en cada mòdul, però això significa introduir un nou camp en cada mòdul cosa que incrementa la complexitat molt. El segon problema radica en assignar i alliberar memòria dinàmica (per poder utilitzar el polimorfisme) constantment pels missatges, cosa que disminueix el rendiment del sistema i afegeix fugues de memòria si no s'utilitza amb compte, a més a més, és necessari convertir el missatge al tipus d'esdeveniment.

Per això s'ha eliminat i creat una nova classe (*EventState*), que canvia el sistema de missatges per a un sistema d'enquesta (*polling*) i guarda l'estat de l'esdeveniment que s'ha produït. La classe inclou totes les dades dels esdeveniments (*KeyPress*, *KeyRelease*, *MouseMove*, *WindowResize*...) i uns *callbacks* que s'instal·len en el sistema d'esdeveniments de GLFW i actualitzen l'estat intern. Pel fet que la llibreria ImGUI[17] instal·la els seus *callbacks* en el mateix sistema, s'ha implementat lògica per permetre diferenciar si l'esdeveniment va dirigit per al joc o a la llibreria ImGUI[17].

Aquesta classe té dos tipus de mètodes: un per obtenir les dades de qualsevol esdeveniment (posicions *x, y* pel *MouseMove*) i un altre per sol·licitar si un esdeveniment ha canviat des del *frame* anterior.

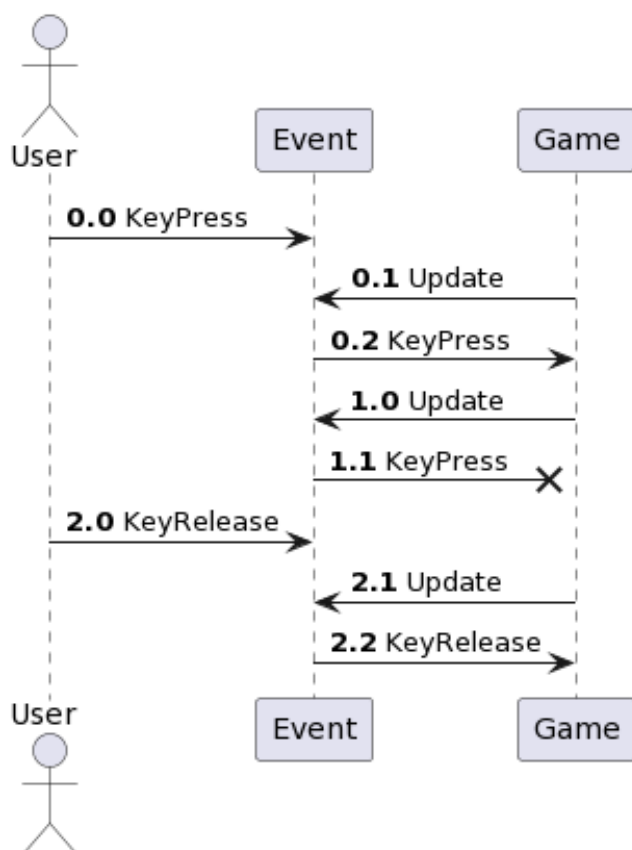


Fig. 7: En el frame 1, Event no envia KeyPress tot i que l'usuari segueix pressionant la tecla.

3.4 Càmera

La classe Camera conté les matrius que s'utilitzen per posicionar i transformar els objectes al món, més concretament, proveeix dues matrius dependentment si es vol una projecció ortogràfica o una projecció perspectiva.

Si l'objecte que es vol dibuixar en pantalla és un objecte 3D, s'utilitza la projecció perspectiva com és el cas de l'esfera o el cub. Si és un objecte 2D, s'utilitza la projecció ortogràfica per dibuixar els objectes a pantalla com és el cas de la mira.

La càmera és controlable utilitzant el ratolí i el teclat. En la implementació s'ha utilitzat els angles d'Euler, concretament el *pitch* i el *yaw* per a canviar l'orientació de la càmera. A més a més, existeixen dos modes de control, un mode que permet moure la càmera sense fer clic a un lloc buit i un altre que és necessari fer clic per a moure la càmera. Les opcions es poden activar i desactivar amb la interfície d'usuari o utilitzant els accessos directes, en aquest cas la tecla F1. Amb el teclat es pot moure la posició de la càmera i es pot restablir la posició amb la tecla F2.

3.5 Selecció d'objectes

Una de les funcions fonamentals de qualsevol joc és la selecció d'objectes, ja sigui per moure un objecte amb el ratolí (com és el nostre cas, moure una fitxa) o per identificar i realitzar accions sobre un objecte en concret (identificar l'enemic seleccionat per realitzar un atac). Existeixen diverses tècniques que tenen els seus avantatges i inconvenients. Per aquest projecte s'ha considerat dues tècniques:

La primera és implementar una tècnica anomenada *Ray Casting* que consisteix a projectar un raig des de la posició del ratolí amb la direcció de la càmera i buscar col·lisions entre totes les entitats del joc. Aquest és un mètode que no penalitza molt el rendiment del joc si utilitzem les estructures presents en un motor físic (*bounding box* i arbres binaris) com ara Bullet3[4].

La segona manera d'implementar la selecció és utilitzant una tècnica anomenada *Picking Object*. La idea principal és codificar l'identificador de l'objecte en un color únic i pintar els objectes amb aquest color, després, es recupera el color que està sota el ratolí i obtenim l'identificador a partir del color. L'avantatge d'aquest mètode és la seva simplicitat comparada amb altres tècniques que utilitzen un motor físic. Però en canvi, el principal desavantatge que té aquesta tècnica és que disminueix el rendiment del joc pel fet que s'ha de dibuixar l'escena dues vegades (una per obtenir l'objecte i una altra per l'escena normal). Un altre desavantatge és el fet que estem obligats a utilitzar un nombre finit de colors i per tant només podrem seleccionar un nombre finit d'objectes.

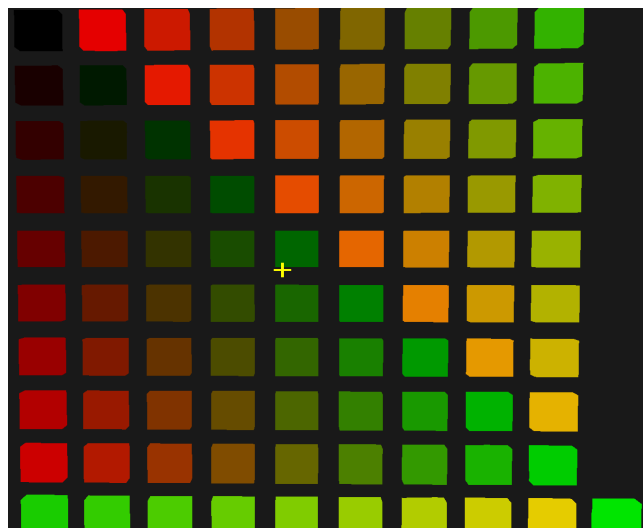


Fig. 8: Visualització dels colors assignats als objectes.

Per desenvolupament del projecte s'ha optat per utilitzar el segon mètode per la seva simplicitat. S'ha implementat una transformació que associa cada identificador amb un color RGB amb l'algoritme de la figura 9. Aquest algoritme té l'avantatge que codifica l'identificador en base 10 perquè es pugui obtenir fàcilment a mà. Per exemple si l'identificador és 123, obtenim que $R = 0.1$, $G = 0.2$, $B = 0.3$.

```

glm::vec3 tfg::genPickColor(int n) {
    float r = int((n % 10)) / 10.f;
    float g = int((n % 100) / 10) / 10.f;
    float b = int((n % 1000) / 100) / 100.f;
    glm::vec3 newColor(r, g, b);

    LOG_DEBUG("Pick color {}-", glm::to_string(newColor).c_str());
    return newColor;
}
  
```

Fig. 9: Funció de generació del color.

La funció *colorToId(color)* obté de manera trivial l'identificador a partir del color si el color no coincideix amb el color de fons.

```

int tfg::colorToId(glm::vec3 color) {
    if (color == glm::vec3(GlobalOptions.clearColor))
        return -1;

    int id = std::round(color.r * 10) +
             std::round(color.g * 100) +
             std::round(color.b * 1000);
    assert(id < Injector::EntitySize);
    return id;
}

```

Fig. 10: Funció de obtenció del identificador.

A més a més de les funcions de conversió, s'utilitza la funció `glReadPixels` que permet obtenir el color (entre altres variables canviant l'argument *type*) del buffer de la GPU en un punt (x, y), en aquest cas la posició del ratolí dintre de la finestra.

```

glm::vec3 color;
glReadPixels((int) win.x, (int) win.y, 1, 1, GL_RGB, GL_FLOAT, glm::value_ptr(color));

```

Fig. 11: Utilització de la funció `glReadPixels`.

3.6 Àudio

Per ajudar a la memorització dels números en braille, el joc diu el número en anglès quan fa clic a qualsevol fitxa, aquesta opció es pot desactivar en el menú d'opcions si no és desitjable.

En el fitxer `Audio.cpp` s'han implementat dues funcions que permeten la càrrega de fitxers d'àudio i la reproducció d'aquests en les diferents plataformes.

En Linux s'ha implementat utilitzant `Pipewire`[24] que és un framework de baix nivell per la reproducció de vídeo i àudio amb una latència mínima. `Pipewire` té un daemon que serveix com a servidor pel streaming de fitxers multimèdia i un client que dona ordres al servidor utilitzant el mòdul `Core API`[23]. El procés per la reproducció consta de quatre passos: la creació d'un objecte stream i d'un loop, la connexió del stream del client amb el servidor, l'enviament de dades per a reproduir en el bucle i la destrucció dels objectes inicialitzats[25].

En Windows la reproducció d'àudio té els mateixos passos (inicialitzar, configurar i destruir els components de `xAudio2`[30]) només canvia en l'enviament de dades: en Linux s'ha d'enviar paquets 5000 Bytes d'àudio i per això és necessari crear un nou fil d'execució, en canvi en Windows, l'API de `xAudio2`[30] accepta buffers complets.

Els fitxers d'àudio procedeixen de `FreeSound`[10]. Per carregar els fitxers multimèdia és necessari que tinguin format WAV, PCM, una freqüència de mostreig de 44,10 kHz, dos canals i per això s'han editat amb `Audacity`[15].

3.7 Il·luminació

Per millorar l'estètica del joc, per això s'ha implementat una llum direccional, bàsicament una llanterna posicionada en el centre de la càmera i es mou amb aquesta. Si es vol, es pot desacoblar de la càmera amb el botó *lock light* del menú.

3.8 Estructura de dades

Abans de passar a la implementació del sudoku i del *gameplay* s'ha de parlar sobre les estructures que s'han utilitzat. En el fitxer `DataStructs.h` podem trobar totes les estructures que s'utilitzen en el projecte, les més importants són les que estan presents en la figura 12

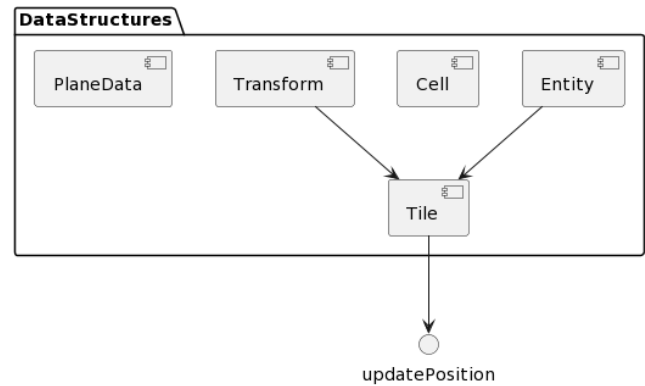


Fig. 12: Diagrama de components de les estructures de dades.

Primer tenim `PlaneData`, aquesta estructura s'utilitza per fer càlculs relacionats amb els plans; conté mètodes que permeten calcular el pla a partir de tres punts. Seguidament tenim `Transform`, s'utilitza per guardar les transformacions necessàries per a la renderització d'un objecte en l'escena: la posició, l'escala, la rotació i el color. L'estructura `Cell` guarda informació sobre la cel·la d'un sudoku: el valor inicial, si es pot escriure o no i la solució. L'estructura `Entitat` conté l'identificador i el color generat per l'algorisme explicat en la secció 3.5. Finalment, `Tile` guarda les dades de l'entitat del joc (si es pot moure, el número, etc.).

3.9 Sudoku

Aquesta classe té les següents funcions: inicialitzar el joc, comprovar que l'estat del joc és vàlid, generar i carregar sudokus.

La inicialització col·loca les fitxes en la posició correcta i inicialitza l'estructura de dades `Tile`.

Per comprovar que el joc s'ha completat correctament, sumem totes les cel·les i es comprova que és igual a 405 $((1 + 2 + \dots + 9) * 9)$.

La classe permet generar sudokus amb tres nivells de dificultat, cada nivell té un nombre diferent de caselles buides (3, 5 i 7 caselles buides); per generar un sudoku, primer, emplenem les subgraelles de la diagonal que són independents entre si, després, utilitzem un algorisme de *backtracking* per arribar a una solució i treiem valors d'una fila i columna aleatòria fins a arribar a la dificultat desitjada.

3.10 Classe Món

És la classe que conté el bucle del joc i controla les accions permeses del jugador o sigui el *gameplay* del joc. La classe `Món` depèn de totes les classes anteriors, com es pot veure en la figura 13, exposa dues interfícies: el bucle principal (`startGame`) i la funció d'inicialització. El bucle principal

renderitza tots els objectes de la pantalla i actualitza el sistema d'esdeveniments de GLFW que la vegada actualitza el nostre sistema d'esdeveniments.

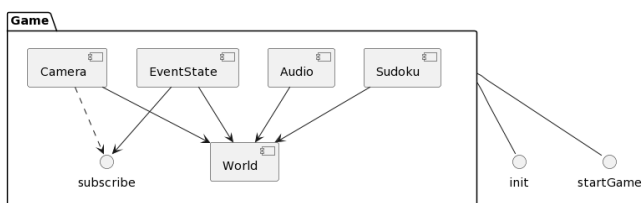


Fig. 13: Paquet game.

La classe conté referències a la fitxa seleccionada, la fitxa que sota el ratolí i la fitxa més pròxima al ratolí (*selectedTile*, *hoveredTile* i *nearestTile*). Amb aquestes tres variables podem implementar totes les accions requerides pel jugador.

La variable *nearestTile* conté la fitxa que està més a prop del ratolí, s'actualitza per cada *frame* utilitzant la funció *nearestTile*,

La variable *hoveredTile* s'obté mitjançant la funció *colorTold* i representa la fitxa que se selecciona si es fa clic.

Si es fa clic, la variable *selectedTile* s'assigna amb el valor de *hoveredTile*, podem moure la fitxa pel pla del si es manté el botó del ratolí pulsat. Amb la fitxa seleccionada deixar-la anar en la fitxa indicada per la variable *nearestTile* que es permetrà assignar números (si és correcte) als llocs buits.

Les fitxes permeten donar pistes en fer un clic (mostra la solució parcial en braille fins a un màxim de 3 esferes) i reproduïx el número en anglès.

La interfície està implementada amb ImGUI[17], una llibreria que permet crear GUI de forma declarativa de la mateixa manera que el *framework* Flutter[9]. La llibreria té varis *widgets* com ara: *checkboxes*, *sliders* per a escalars i vectors, *modal popups* ... que permeten la creació d'interfície fàcilment.

El menú principal conté botons per a carregar sudokus i generar sudokus de diferents dificultats i diversos *checkboxes* per a activar i desactivar opcions i finestres de control (control de la càmera, de la il·luminació, etc.) del joc.

4 RESULTATS I TESTS

Una vegada finalitzat la fase d'implementació, s'ha compilat el joc en mode *release* per a diferents plataformes i s'ha testejat que el joc pugui assolir 60 FPS en les dues plataformes i s'ha convidat a usuaris externs per al *user-testing*.

El joc, com es pot veure en la figura 15, està format per dues parts: el tauler i la baralla de fitxes (en blau marí). Les fitxes tenen 6 esferes incrustades que formen el seu valor en braille. En el tauler, es pot veure que hi ha dos tipus de fitxes amb colors diferents: en groc, les fitxes inicials del sudoku i en violeta, els espais en buit del sudoku. Les fitxes del tauler no es poden moure de la seva posició inicial amb el ratolí, però sí poden reproduir el valor de la fitxa si es fa clic sobre aquesta. Si es fa clic sobre una fitxa violeta (una fitxa buida), el nombre d'esferes canvia i dona pistes sobre quin número en braille ha d'anar la casella buida per solucionar el sudoku, concretament, es pot obtenir el menor

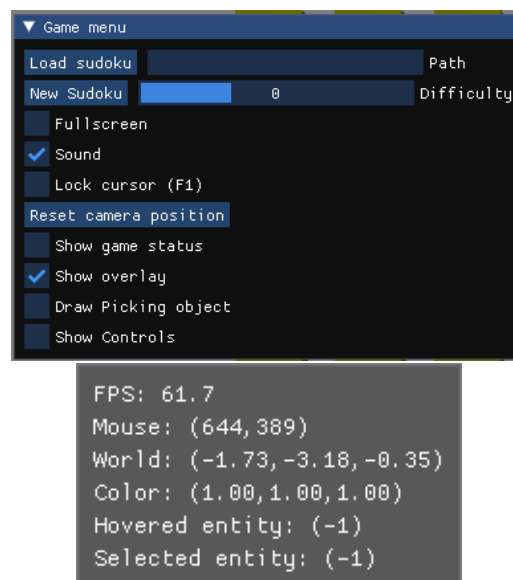


Fig. 14: Menú i overlay del joc.

de 3 esferes o el nombre d'esferes de la solució menys 1, per exemple, si la solució és 7, pot donar fins a 3 pistes, en canvi si la solució és 2, només donarà 1 esfera.

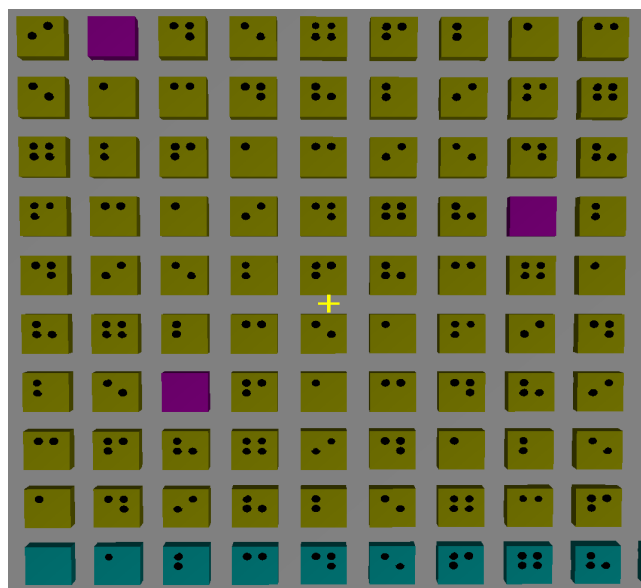


Fig. 15: Estat inicial de joc.

Per resoldre el sudoku és necessari arrossegat les fitxes de la baralla fins a una fitxa buida, si el número és correcte, la fitxa buida canvia al número de la fitxa arrossegada, si no, la fitxa no canvia. Una vegada completat el sudoku, es mostra un missatge que es pot veure en la figura 16; fent clic al botó es pot començar una nova partida.

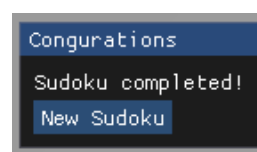


Fig. 16: Missatge de final de la partida.

El joc conté diferents opcions (figura 17) que per-

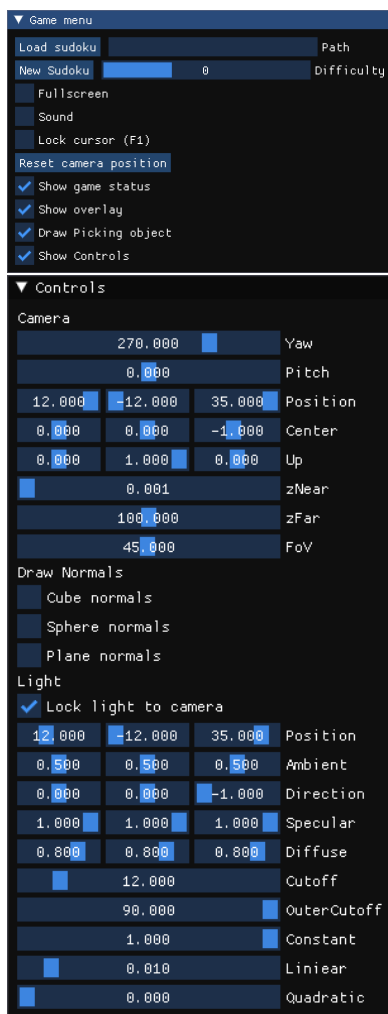


Fig. 17: Menú d'opcions.

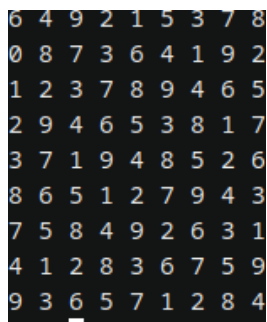


Fig. 18: Format del sudoku per carregar.

met personalitzar l'experiència de joc, es pot canviar la il·luminació, la posició de la càmera, etc. Una opció clau és canviar la dificultat del sudoku, el nombre de caselles buides augmenta amb la dificultat. També hi ha una entrada de text per introduir la ruta d'un sudoku en un fitxer text. El format del sudoku ha de ser el mostrat en la figura 18.

Per la part de tests, al llarg del desenvolupament, s'han afegit en funcions clau del codi clàusules *asserts* que permeten afirmar supòsits per la correcta execució del programa, per exemple, durant la construcció d'un objecte 3D els *buffers* per guardar les dades han d'existir, si no, es dona un error com es pot veure en la figura 19 i es para l'execució del joc. A més a més, s'ha utilitzat la llibreria *spdlog*[28] per registrar els errors o alertes que es produeixen en el joc

i així poder solucionar els errors més fàcilment.

```
int Shader::getLocation(const std::string &name) const {
    auto a = location.find(name);
    assert(a != location.end());
    return a->second;
}

glCreateBuffers(1, &indexBufferId);
assert(vertexArrayId_ != 0);
assert(positionId_ != 0);
assert(indexBufferId_ != 0);
```

TFG: /home/.../TFG/src/render/Object.h:10: Assertion 'indexBufferId_ != 0' failed

Fig. 19: Si no existeix la variable en el codi *shader* o si no existeix el *buffer*, es para l'execució del joc.

5 CONCLUSIONS I MILLORES

Aquest projecte ha sigut una primera experiència al desenvolupament de jocs amb OpenGL amb la utilització de shaders, s'ha pogut expandir el coneixement sobre els gràfics 3D i al desenvolupament de programes en diferents plataformes.

Un dels problemes sorgits al llarg del projecte és la compilació entre diferents plataformes: en sistemes Linux les dependències s'instal·len de manera trivial mitjançant la terminal, en canvi, en Windows s'ha de descarregar la versió correcta instal·lar-lo manualment (configurar la compilació, l'enllaçat de les llibreries, etc). Això s'ha solucionat utilitzant el mòdul *FetchContent*[8] de CMake que permet descarregar de manera automàtica les dependències necessàries.

Hi ha molts aspectes que es poden millorar, sobretot en l'estètica: afegir textures a les fitxes, afegir una textura pel fons, implementar la càrrega de models 3D i afegir una mà, etc.

Un altre àmbit a millorar és el sistema de renderitzat. En aquest joc, s'utilitza moltes crides a *glDraw* que si més no, l'impacte al rendiment no és apreciable en un joc, ja que no hi ha molts objectes, es podria veure afectat en afegir més objectes. La solució seria implementar la tècnica del *batch rendering*[20] que permet dibuixar totes les figures només utilitzant una crida a *glDraw* i així millorar el rendiment del joc.

AGRAÏMENTS

Voldria expressar el meu agraïment més sincer al meu tutor Enric Martí Godia que m'ha estat ajudant des de l'inici del projecte, també gràcies a la meua família i amics que han mostrat el seu suport i han estat testejant l'aplicació. Finalment, a l'usuari JoeyDeVries per crear la pàgina <https://learnopengl.com/> que ha sigut un recurs molt útil durant tot el projecte.

REFERÈNCIES

- [1] «An OpenGL library», GLFW. <https://www.glfw.org/> (data darrer accés 27 febrer 2022).
- [2] «Braille Sudoku Set». <https://lssproducts.com/braille-sudoku-set/> (data darrer accés 4 abril 2022).

- [3] «Build software better, together», GitHub. <https://github.com> (data darrer accés 27 febrer 2022).
- [4] Bullet Physics SDK. Bullet Physics SDK, 2022. [En línia]. Disponible a: <https://github.com/bulletphysics/bullet3> (data darrer accés 4 abril 2022).
- [5] «CLion: A Cross-Platform IDE for C and C++ by JetBrains». <https://www.jetbrains.com/clion/> (data darrer accés 27 febrer 2022).
- [6] «CMake». <https://cmake.org/> (data darrer accés 27 febrer 2022).
- [7] GrantMeStrength, «Direct3D - Win32 apps». <https://docs.microsoft.com/en-us/windows/win32/direct3d> (data darrer accés 25 juny 2022).
- [8] «FetchContent - CMake 3.24.0-rc2 Documentation». <https://cmake.org/cmake/help/latest/module/FetchContent.html> (data darrer accés 27 febrer 2022).
- [9] «Flutter - Build apps for any screen». <https://flutter.dev/> (data darrer accés 19 juny 2022).
- [10] «Freesound - "Counting Numbers Normal Voice 0-12" by dersuperanton». <https://freesound.org/people/dersuperanton/sounds/434733/> (data darrer accés 22 maig 2022).
- [11] «Git». <https://git-scm.com/> (data darrer accés 27 febrer 2022).
- [12] «glbinding». <https://glbinding.org/> (data darrer accés 27 febrer 2022).
- [13] «GLM SDK contribution». <https://www.opengl.org/sdk/libs/GLM/> (data darrer accés 27 febrer 2022).
- [14] G. Engine, «Godot Engine - Free and open source 2D and 3D game engine», Godot Engine. <https://godotengine.org/> (data darrer accés 19 juny 2022).
- [15] «Home», Audacity ®. <https://www.audacityteam.org> (data darrer accés 22 maig 2022).
- [16] «KanbanFlow - Lean project management. Simplified.» <https://kanbanflow.com/> (data darrer accés 27 febrer 2022).
- [17] «ocornut/imgui at docking», GitHub. <https://github.com/ocornut/imgui> (data darrer accés 27 febrer 2022).
- [18] Open Asset Import Library (assimp).. Open Asset Import Library, 2022. [En línia]. Disponible a: <https://github.com/assimp/assimp> (data darrer accés 5 maig 2022).
- [19] «OpenGL - The Industry Standard for High Performance Graphics». <https://www.opengl.org/> (data darrer accés 27 febrer 2022).
- [20] «OpenGL Batch Rendering», GameDev.net. <https://gamedev.net/tutorials/programming/graphics/opengl-batch-rendering-r3900> (data darrer accés 19 juny 2022).
- [21] «OpenGL Sphere». http://www.songho.ca/opengl/gl_sphere.html (data darrer accés 2 abril 2022).
- [22] «RenderDoc». <https://renderdoc.org/> (data darrer accés 27 febrer 2022).
- [23] «PipeWire: Core API». https://docs.pipewire.org/group__api__pw__core.html (data darrer accés 22 maig 2022).
- [24] «PipeWire: PipeWire API». https://docs.pipewire.org/page_api.html (data darrer accés 22 maig 2022).
- [25] «PipeWire: Tutorial - Part 4: Playing A Tone». https://docs.pipewire.org/page_tutorial4.html (data darrer accés 22 maig 2022).
- [26] J. Kessenich, «The OpenGL Shading Language 4.4», <https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4> (data darrer accés 19 juny 2022).
- [27] «Unity Real Time Development Platform 3D, 2D VR and AR Engine». <https://unity.com/> (data darrer accés 19 juny 2022).
- [28] G. Melman, spdlog. 2022. [En línia]. Disponible a: <https://github.com/gabime/spdlog> (data darrer accés 27 febrer 2022).
- [29] stevewhims, «Writing HLSL Shaders in Direct3D 9 - Win32 apps». <https://docs.microsoft.com/en-us/windows/win32/direct3dhls/dx-graphics-hlsl-writing-shaders-9> (data darrer accés 25 juny 2022).
- [30] drewbatgit, «XAudio2 programming guide - Win32 apps». <https://docs.microsoft.com/en-us/windows/win32/xaudio2/programming-guide> (data darrer accés 22 maig 2022).

APÈNDIX

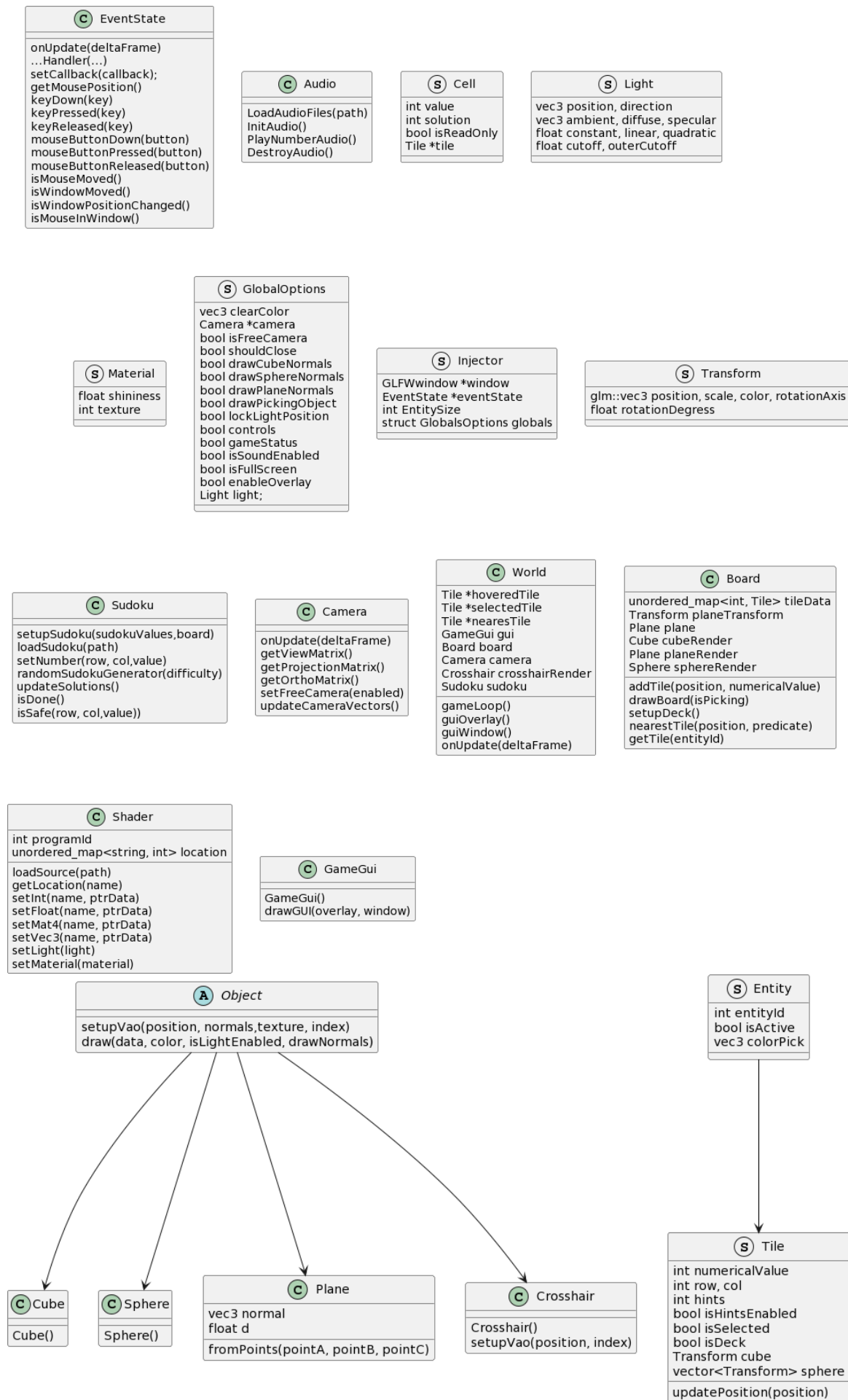


Diagrama UML del projecte.

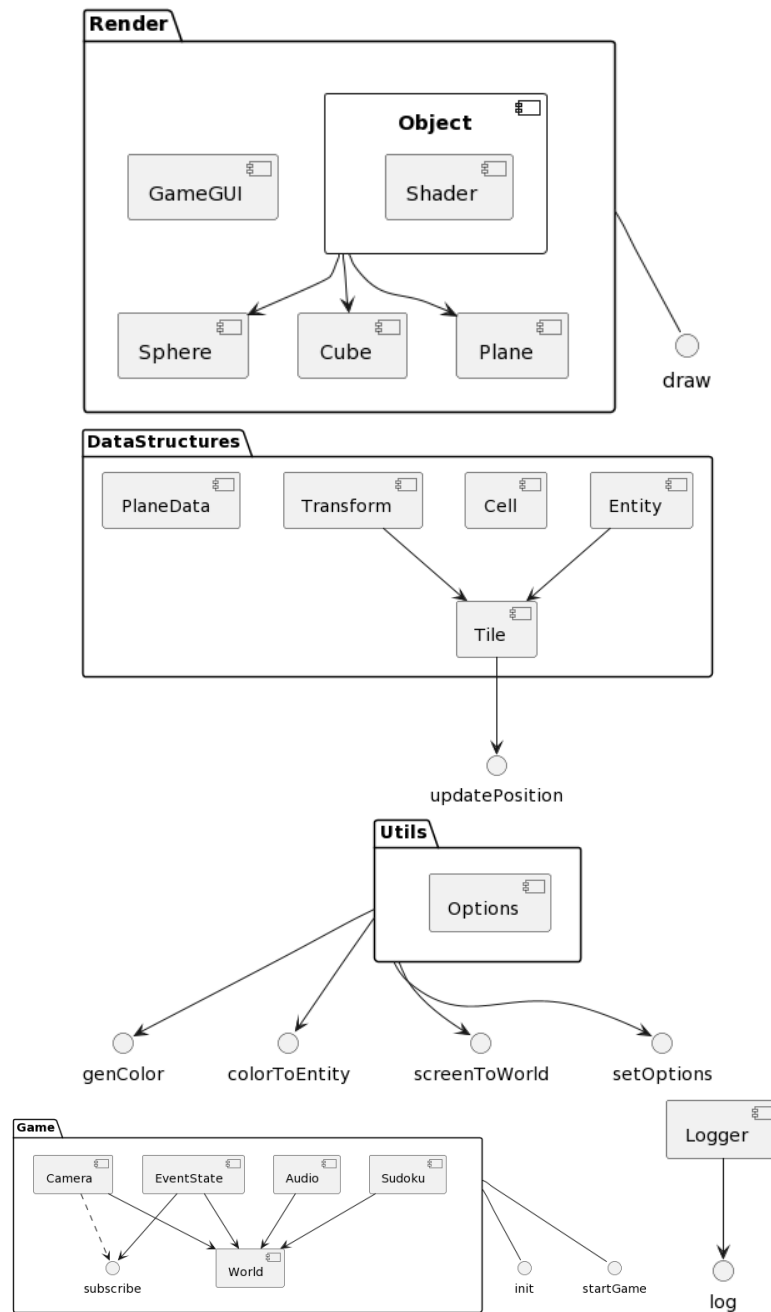


Diagrama de components del projecte.