TecnoCampus

Centres universitaris adscrits a la

upf. Universitat Pompeu Fabra Barcelona

Doble titulació en Enginyeria Informàtica i Disseny i Producció de Videojocs

Creació i desenvolupament d'una aplicació interactiva en línia a través de l'API de Discord

Memòria Final

Arnau Reig Méndez
Tutor: Rafael González Fernández
2022-2023

TecnoCampus Mataró-Maresme

Abstract

Discord is an application to chat that allows the creation of simple games or tools where text is the primary focus of interaction. The goal of this investigation was to experiment with the new Discord API which was updated recently and check the viability of dynamically generated images in this type of programs. The result was Discland, an RPG like game very similar to Questron or MUD II.

Resum

Discord és una aplicació per a parlar amb altres usuaris que permet crear jocs senzills o eines on el seu principal mètode d'interacció és mitjançant text. L'objectiu del treball ha sigut experimentar amb la nova actualització de l'API per comprovar la viabilitat d'incloure imatges generades de forma dinàmica en aquests programes. El resultat ha sigut Discland, un joc RPG molt similar a Questron o MUD II.

Resumen

Discord es una aplicación para chatear que permite crear juegos muy sencillos o herramientas donde principalmente se interactúa mediante texto. El objetivo de este trabajo ha sido experimentar con su recién actualizada API para estudiar la viabilidad de usar imágenes generadas de forma dinámica en estos programas. El resultado ha sido Discland, un juego RPG al estilo de aventuras como Questron o MUD II.

Taula de continguts

Índex d'imatges

Índex de taules

1 Introducció

Actualment estem vivint en una era on la comunicació és el centre de les nostres vides. Les aplicacions de missatgeria instantània com *WhatsApp* o *Telegram* permeten actualitzar-nos amb els nostres contactes al moment. En el sector del *gaming*, *Discord* es manté com a líder indiscutible amb un creixement anual tant de beneficis com d'usuaris (Curry, 2023).

Discord és una aplicació que permet la comunicació via text, vídeo o veu entre els seus membres, afavorint la creació de comunitats en línia per a grups de persones de forma totalment gratuïta. Aquest model de negoci ha sigut un dels seus principals factors d'èxit (Miriam, 2019), però tot aquest ecosistema tan viu a la vegada genera unes necessitats: la creació de contingut i la personalització d'aquests espais virtuals. Aquí és on entren en joc els *bots*.

Els *bots* són scripts que, a partir d'una API creada per Discord, permeten reaccionar a esdeveniments i respondre a missatges automatitzats com si fossin usuaris normals. Gràcies a aquestes aplicacions els usuaris poden, per exemple, integrar software de gestió i planificació com Trello dins dels seus servidors, moderar amb més facilitats les seves comunitats o col·leccionar cartes com a recompensa per la seva activitat. La part interessant de tota aquesta moda és també la capacitat de monetització que els desenvolupadors han pogut extreure d'aquests programes. Aquesta funcionalitat és relativament nova i promet un futur encara més brillant per aquelles persones que es volen guanyar la vida a partir del desenvolupament de bots.

Un cas que és d'especial interès pel treball són els bots d'entreteniment. Els jocs creats dins d'aquest ecosistema estan subjectes a grans limitacions com la interacció majoritàriament via comandes. A part, els bots només poden respondre amb un text o imatge, cosa que dificulta el procés de disseny a l'hora de la creació d'interfícies. Molts referents que s'estudien en el treball eviten com poden aquesta limitació aprofitant al màxim els emojis o taules per tal de tenir més usabilitat.

2 Objectius

Principals:

- **Creació d'un videojoc fent servir l'API que proporciona Discord:** Aquest és l'objectiu més important i d'on neixen la resta de punts. Per tant, tot allò que sigui necessari per a realitzar un MVP (*Minimum Viable Product*) forma part del conjunt d'objectius principals.
- **Generar imatges de forma dinàmica:** Cal investigar i implementar un sistema de generació d'imatges, ja que és el principal punt de diversificació amb la resta de bots.
- **Creació d'una base de dades:** Donat l'entorn multiusuari, no només cal tenir una base de dades sinó una implementació que garanteixi múltiples partides paral·leles a la vegada.
- **Lectura i modelització d'un sistema DDL (*Data Definition Language*):** Tota la informació del joc ha d'estar continguda en uns arxius i ha de poder ser editada de forma senzilla.
- **Disseny del joc:** Essencial per a tenir un joc amb un sentit i mecàniques escaients dins l'entorn de Discord.
- **Art funcional:** Per a poder fer ús d'imatges dinàmiques cal crear un art que estigui adaptat i sigui funcional.
- **Arquitectura que garanteix flexibilitat:** Donat que no hi ha cap framework o estructura preestablerta, cal dissenyar tot el sistema des de zero i amb una gran flexibilitat en ment. El joc pot requerir canvis o addicions no planejades.

Secundaris:

- **Sistemes de combat:** Per aplicar les funcionalitats en línia típiques dels Multi User Dungeons s'hauria de dissenyar un sistema de combat entre dos jugadors.
- **Deployment:** S'hauria de pujar el codi a un servidor online per poder mantenir el bot obert les 24 hores del dia.

3 Referents

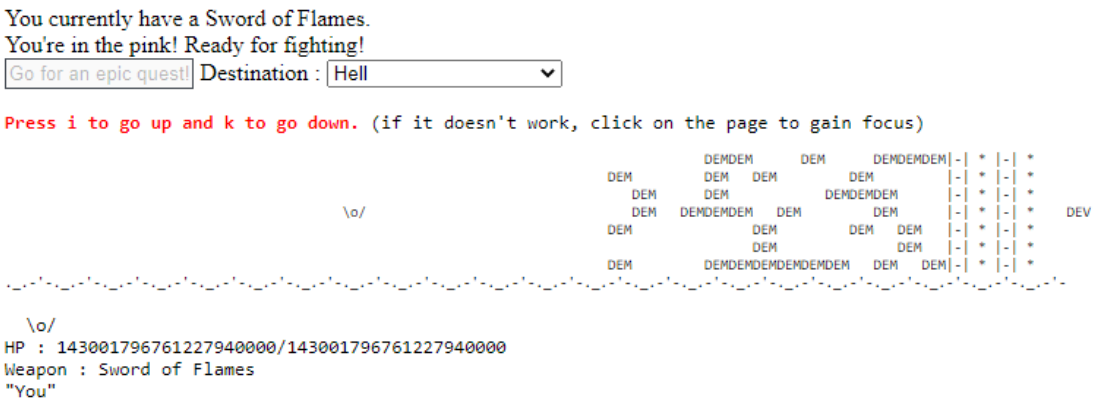
Per entendre els objectius que es volen assolir i en quin entorn s'està treballant cal una primera introducció amb algunes aplicacions similars que han servit com a inspiració a l'hora de realitzar el treball.

3.1 Candy Box

Candy Box és un joc per navegador creat purament a partir de caràcters i botons bàsics del mateix HTML. L'objectiu principal és guanyar cada cop més caramels a partir de millores i noves zones. Hi ha moltíssimes interaccions interessants fent que part de l'entreteniment tracti en intentar descobrir-ho tot.

La referència principal que cal agafar d'aquest joc és el sistema de millora i recol·lecció. El fet d'anar obtenint millores permanents que et permetin a la vegada obtenir més recursos és una idea senzilla que es pot anar expandint a mesura que el disseny evoluciona.

Una referència que no es pot fer servir és el sistema de combat. *Discord* no permet interaccions a temps real i, per tant, no es poden fer servir dissenys que siguin "*time-sensitive*".

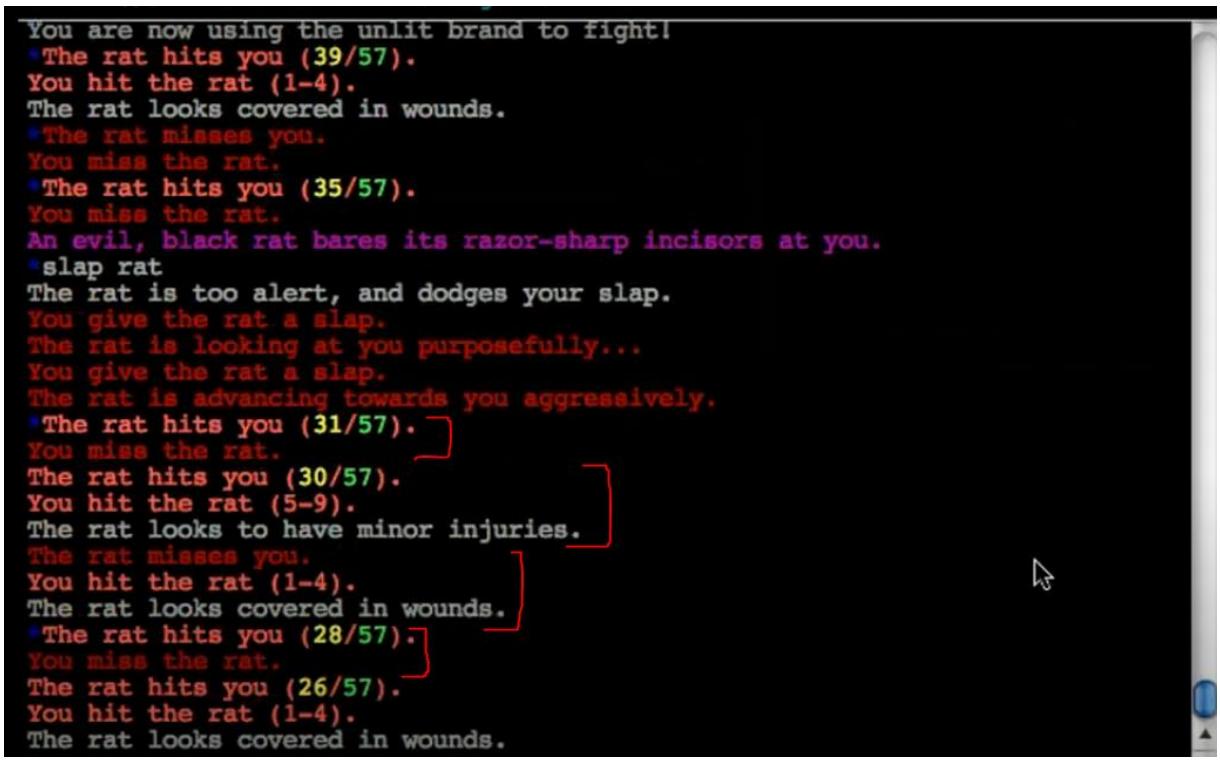


Imatge :1 Captura del sistema de combat de *Candy Box*. També es pot veure com totes les icones i gràfics estan aconseguits purament per mitjà de text. Elaboració pròpia.

3.2 MUD2

Desenvolupat per Richard Bartle i llençat l'any 1985, la saga *MUD2* barreja els components dels jocs basats en text amb un sistema online. D'aquesta forma el món evoluciona amb la interacció dels jugadors. Cada 105 minuts el joc torna a començar de nou per a tothom, fent que una mecànica important és la constant millora pel que fa al coneixement de l'entorn.

La característica principal a tenir en compte pel treball és com utilitza el factor del online per enriquir el món de joc i les seves mecàniques. Altres jugadors poden matar el teu personatge per aconseguir punts, però a la vegada més jugadors implica millors tresors. També és important destacar el seu sistema de combat basat en torns però que s'actualitza en temps real. Amb cada "*update*" el joc calcula el resultat segons les teves característiques i les de l'enemic i actualitza els punts de vida nous. Pots realitzar accions entre aquests "updates" per escapar-te o curar-te.



Imatge 2: Combat del MUD2. Captura del vídeo "*mud2 introduction*" de *curiousattemptbunny*

3.3 Questron

Questron és un *top-down* RPG on l'objectiu principal és sobreviure l'entorn hostil i millorar a poc a poc les teves armes per a finalment lluitar contra un enemic final. El joc no et dona cap mena de mapa i fa entendre que és el jugador qui ha de descobrir a poc a poc com navegar pel món. *Questron* va ser desenvolupat per *Strategic Simulations* l'any 1984 per les consoles *Commodore* 64*, Apple II* i *Atari 8-bit*.

Aquest joc conté sistemes que poden ser molt importants pel desenvolupament del treball. Primer de tot, la forma de representar el mon de joc és discreta. Cada cop que el jugador canvia de casella i es mou la simulació avança. Per exemple, variables com el menjar es veuen reduïdes a mesura que el temps augmenta. A part l'art bàsic dibuixat en vista d'ocell dona una visió molt àmplia de la posició del jugador.

A més a més, com es pot veure en la,Imatge 3 el joc permet fer ús de diverses accions. Aquestes permeten canviar l'estat del teu personatge o interactuar amb el món. Segons la referència de *Questron* (Strategic Simulations, 1984), "xamine" serveix per descobrir trampes ocultes en les masmorres, "*hold item*" per posar un utilitzable a la mà, "*unlock*" per a intentar forçar panys de portes…

Imatge 3: Captura del joc Questron. A l'esquerra del personatge es pot veure la icona d'un poblat. A l'esquerra de la imatge el llistat d'accions que pot realitzar. Imatge per CRPG Addict

3.4 Stardew Valley

Stardew Valley va sortir l'any 2016 com un simulador de vida rural. El core loop consisteix a guanyar diners i millorar la teva vida a la granja a mesura que passen les estacions, venent els resultats de la teva collita o altres troballes dins la mina. A part, hi ha un sistema de progressió que recompensa la interacció constant amb els diferents elements del joc: Augmentar el nivell de "*Farming*" dona més eficiència a l'hora de conrear plantes i augmentar el nivell de "*Foraging*" dona més destresa en utilitzar la destral.

Un dels factors claus que motiva el jugador a continuar millorant és l'exploració. Durant el transcurs del joc es poden anar observant zones bloquejades que requereixen diferents materials per a poder passar. Desbloquejar-les no només imbueix una sensació de satisfacció sinó que a més aporten millores substancials a la teva partida o la possibilitat d'obtenir nous materials.

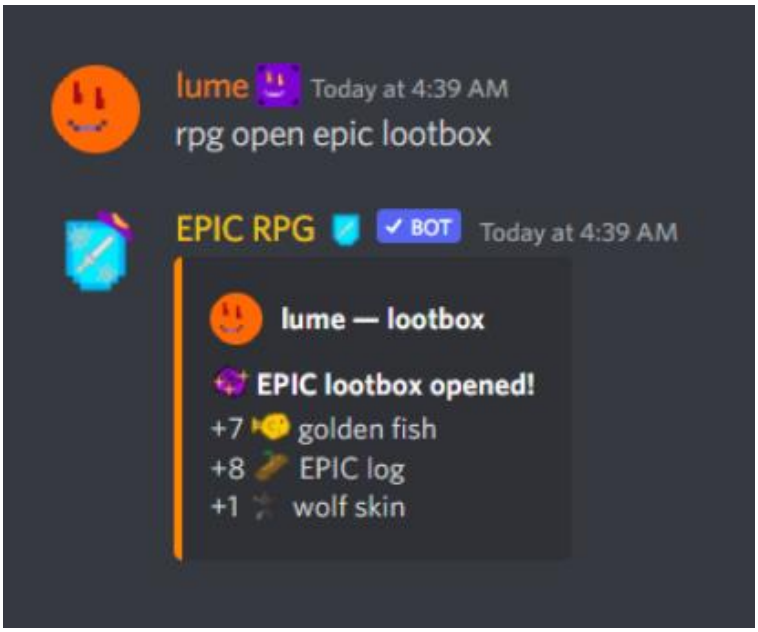
Aquest joc ha estat escollit com un referent pels seus sistemes de progressió i exploració, ja que es vol crear un joc amb temàtica RPG amb diferents zones per descobrir.

Imatge 4: Captura de pantalla de *Stardew Valley*. Imatge per Mashable SE

3.5 EPIC RPG

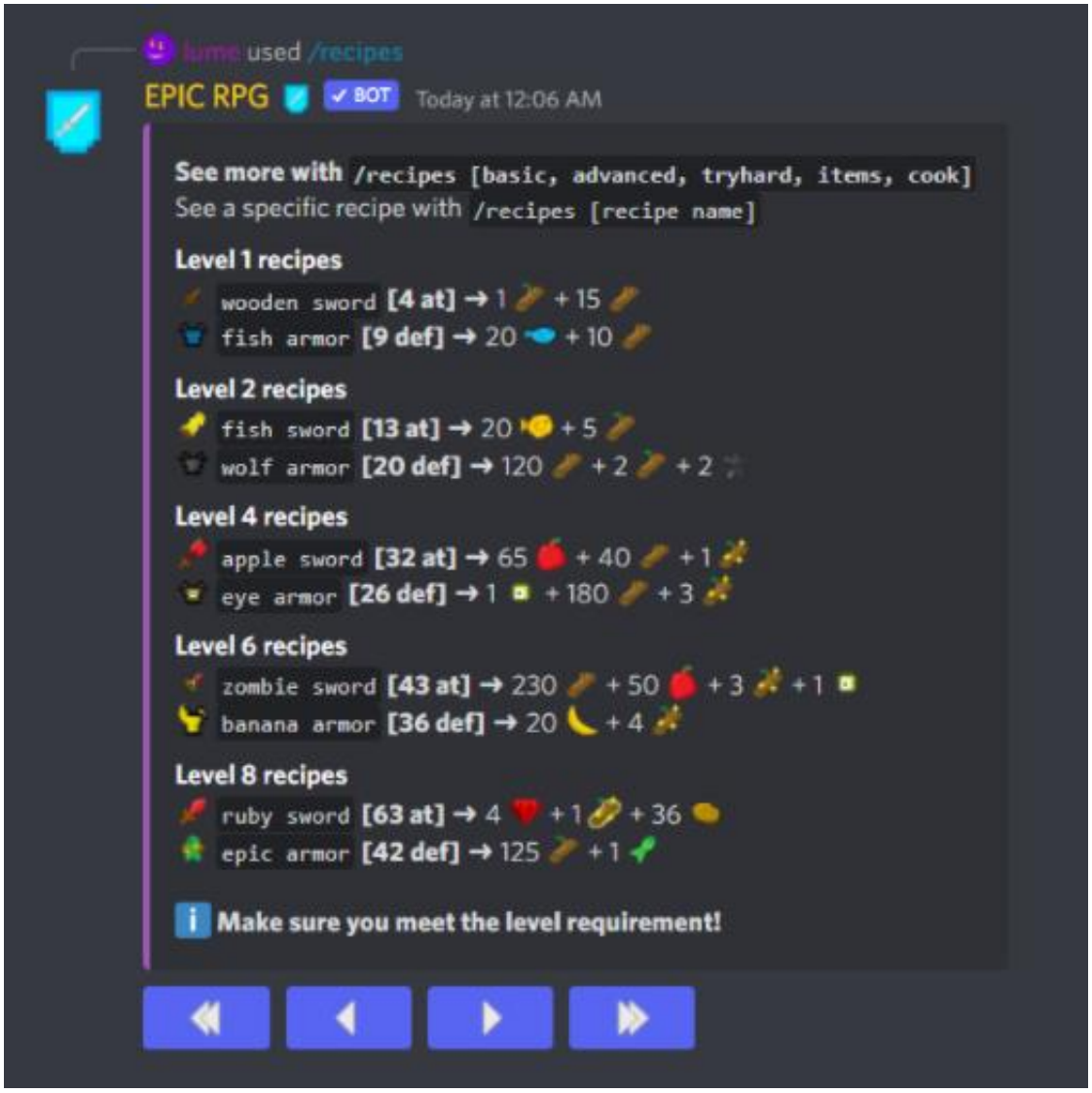
Aquest és un dels bots amb el rànquing més alt d'usuaris i servidors en la llista oficial de Discord. És un exemple interessant d'examinar donat que contempla algunes interaccions bastant similars amb l'aplicació que es vol construir.

Segons la pàgina oficial (Epic RPG Wiki, 2022) "*Epic RPG* és un bot d'economia basat en característiques RPG com masmorres, modalitats de jugador contra jugador, *gambling*, *loot boxes* i nivells".



Imatge 5: Usuari obrint una caixa per aconseguir nous objectes. Elaboració pròpia.

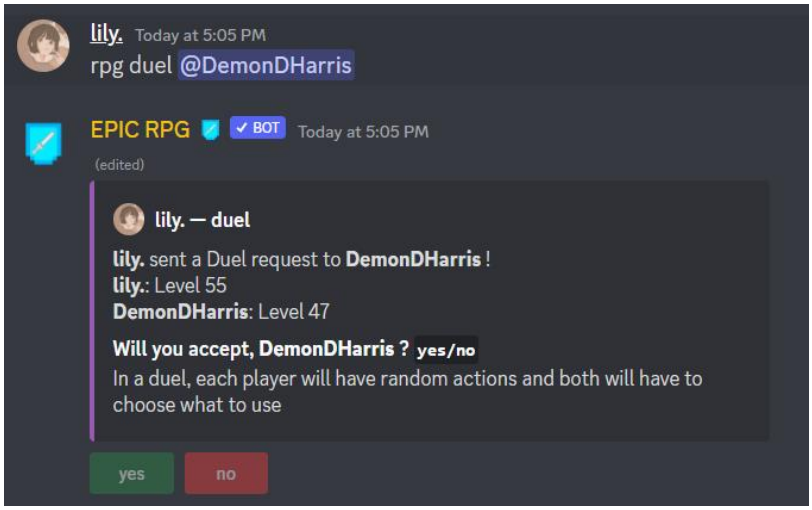
En la Imatge 5 es pot veure el funcionament estàndard d'un bot. Un usuari escriu una comanda que el bot és capaç d'interpretar per executar una acció programada. En aquest cas el bot revisa si l'usuari té una lootbox amb la propietat "epic" en el seu inventari. Si ho confirma, resta una caixa i calcula el seu contingut a partir de taules aleatòries. Tota aquesta informació ha d'estar guardada en la base de dades.



Imatge 6: Exemple de la comanda "/*recipes*". Elaboració pròpia.

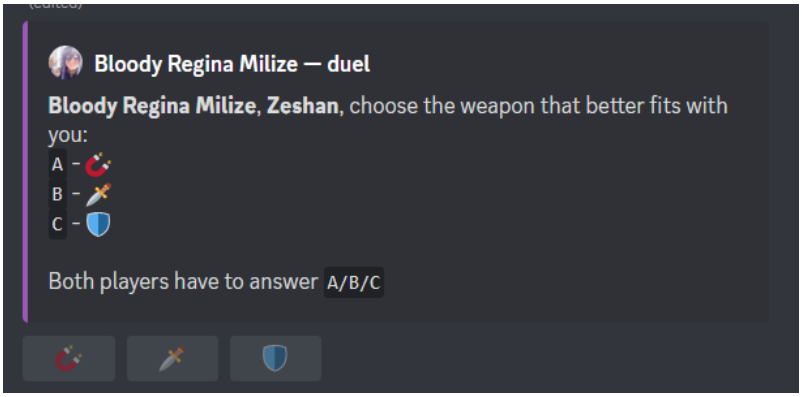
La Imatge 6 mostra un cas de com s'utilitzen els emojis i els botons per mostrar informació als usuaris. La comanda "/recipes" indica al bot que ha d'imprimir una llista per pantalla de les receptes per a crear armadures i espases.

Ara es mostrarà un exemple més complex pel qual dos usuaris poden realitzar un combat i aconseguir una sèrie de recompenses.



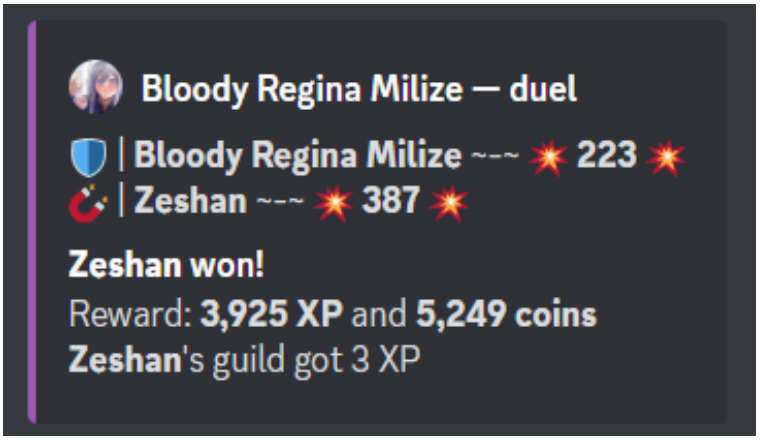
Imatge 7: Inici del combat, l'usuari *lily* demana un repte a l'usuari *DemonDHarris*. Elaboració pròpia.

L'usuari 1 comença escrivint una comanda que conté el nom de l'usuari 2, convidant-lo a un duel (Imatge 7). El bot envia un missatge al canal públic amb dos botons. Aquests botons s'asseguren de només acceptar l'esdeveniment si l'ID és igual al de l'usuari2 (Altres usuaris no poden acceptar o denegar un combat que no hi formen part). En el cas que s'està examinant, l'usuari 2 decideix acceptar el duel i el següent missatge apareix en pantalla:



Imatge 8: Exemple de la primera fase d'un combat entre dos jugadors en EPIC RPG. Elaboració pròpia.

De manera aleatòria apareixen tres icones on els dos usuaris hauran de triar només un (Imatge 8). Cada icona està lligada a una estadística com defensa, atac o diners. En total hi ha set icones i es posen tres de manera aleatòria per evitar que un jugador pugui guanyar tots els combats només centrant-se en una estadística.

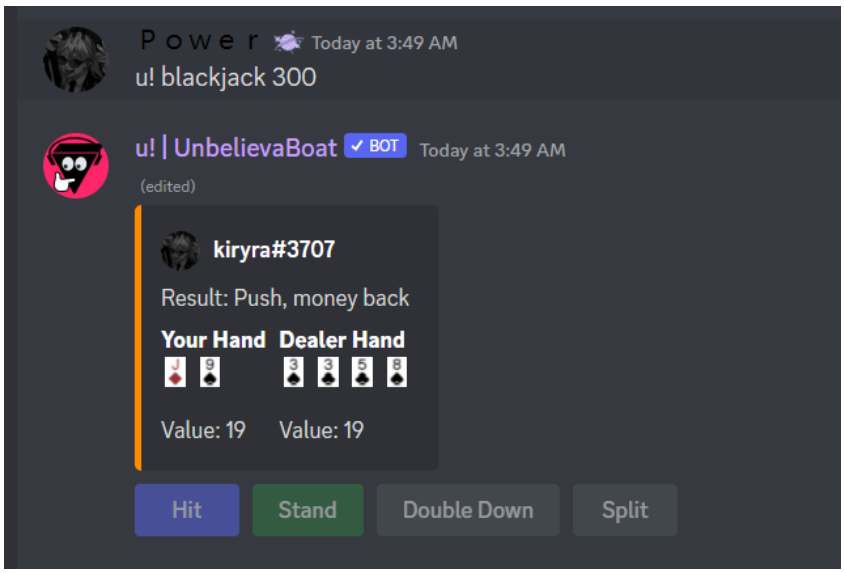


Imatge 9: Resultat d'un combat. Elaboració pròpia.

Un cop els jugadors han fet la seva elecció el bot calcula el resultat. Qui té la puntuació més alta guanya el combat i conseqüentment experiència i diners.

3.6 UnbelievaBoat

UnbelievaBoat és un bot d'ús genèric que té diverses comandes d'utilitat i jocs senzills. És un cas interessant a examinar perquè permeten veure algunes de les interfícies més usuals pel que fa a jocs i les seves interaccions. També és important per entendre les *affordances* en què han estat sotmesos els usuaris. Segons *Donald Norman* a *"The Design of Everyday Things"* (2013) una *affordance* és "la relació entre les propietats d'un objecte i les capacitats d'un agent que determinen com aquell objecte pot ser utilitzat". Si es treballen bé, permeten que l'usuari pugui reconèixer i entendre, per exemple, una interfície gràfica sense cap mena d'esforç.



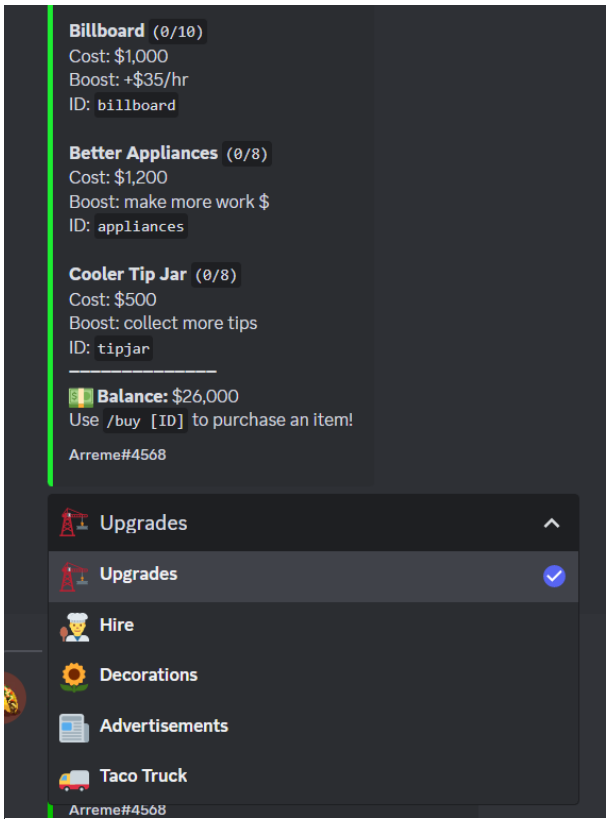
Imatge 10: Un usuari jugant a Blackjack. Elaboració pròpia.

En la Imatge 10 es pot veure un exemple de joc senzill: Una implementació de *Blackjack*. El programa comença una partida després que l'usuari decideixi una quantitat de diners virtuals a apostar. A partir d'aquí es trien una sèrie de cartes aleatòries i deixa a l'usuari decidir l'última acció.

Imatge 11: Exemple de recompenses a comprar per un servidor en específic. Elaboració pròpia.

Per acabar, aquest és un exemple de les recompenses que un bot pot donar als seus jugadors. En la Imatge 5 es pot veure un cas de recompensa interna, on es donen objectes inherents al bot. En aquest cas (Imatge 11) la recompensa és externa: Un usuari pot comprar rols de colors i diferenciar-se de la resta del servidor. És una de les poques recompenses extradiegètiques que un bot pot donar dins de l'entorn de Discord.

3.7 TacoShack



Imatge 12: Sistema de millora de *TacoShack*. Elaboració pròpia.

Recentment Discord ha estat publicant actualitzacions que milloren l'experiència d'usuari a l'hora de tractar amb les comandes. *TacoShack* ha estat triat per exemplificar-ne aquests canvis.

La temàtica principal de *TacoShack* és obrir el teu propi local per a vendre *tacos* i millorar-lo per obtenir més beneficis.

El sistema de millora en la Imatge 12 és justament un cas molt bo a examinar donat el seu ús de les llistes, un nou component que permet elegir una opció entre diverses definides. Quan l'usuari elegeix una, el contingut del missatge canvia amb es noves dades. Això permet donar molta informació nova sense obligar a esciure constantment comandes pel xat.

En general la metodologia nova sol prioritzar l'organització de les comandes en una forma molt similar a la dels menús. D'aquesta forma es pot maximitzar la navegació mitjançant botons i no per comandes. També es prioritza el fet d'editar missatges ja existents com és en el cas de la Imatge 12, així el xat del servidor no s'omple tant.

4 Marc Teòric

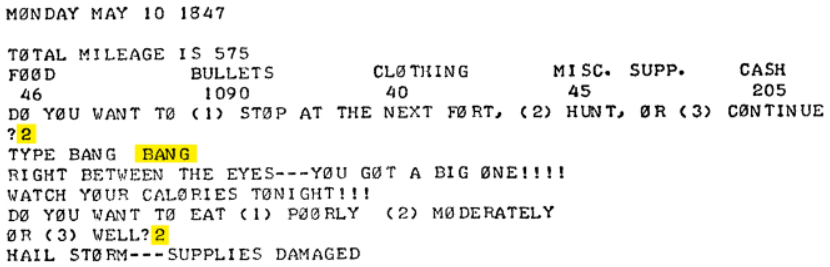
Per a poder explicar de forma estructurada l'objecte del treball, es dividirà en tres grans parts. La primera part es centra en Discord i el seu funcionament. La segona part es centra a explicar els fonaments del disseny de jocs basat en text. Finalment, la tercera part explica quines tecnologies i metodologies utilitzarà la programació del bot, l'objectiu principal del treball.

4.1 El joc

Donada la naturalesa altament basada en text en la que s'ha de treballar, és important detectar videojocs que van estar sotmesos a les mateixes condicions per analitzar els atributs importants que cal tenir presents.

4.1.1 Conceptes i definició

Una primera definició del que significa "interacció basada en text" ve d'un vessant històric. Durant la dècada dels 60 els ordinadors eren molt poc comuns i sovint es localitzaven en institucions de recerca o en universitats en forma de grans estructures anomenades *mainframes* (Wolf M. J., 2012). Donat que aquests ordinadors no permetien una interacció visual amb pantalles els programes creats en aquest hardware s'havien d'adaptar dins d'aquestes limitacions: Introducció de dades mitjançant targetes programables i resultat imprès a paper. Un exemple de programa que utilitzava aquesta tecnologia era *"The Oregon Trail".* En aquest joc educatiu l'usuari es posa en la pell del líder d'un grup d'exploradors travessant els Estats Units, fent front a adversitats, enemics i gestionant els recursos. Cada decisió que fa l'usuari representa un canvi d'estat i una progressió de dues setmanes en el temps de joc (Bouchard, 2017). Sota aquesta conjuntura un programa basat en text és una interacció que es fa mitjançant una terminal on hi ha un agent que interpreta les dades i les transforma en un estat nou.



Imatge 13: Exemple d'una impressió del "The Oregon Trail", per *Died Of Dysentery*

Si es vol tractar el tema en un context més acadèmic, la primera pregunta que s'ha de respondre és si aquests jocs es poden considerar com a gènere. King i Krzywinska defineixen en el seu llibre sobre l'anàlisi dels videojocs des d'una perspectiva cinematogràfica (2002) quatre termes que ajuden a aquesta categorització. Una obra té una 'Plataforma' que defineix per quina consola o sistema es pot jugar. Un 'Gènere' els quals defineix el tipus de mecàniques presents i com afecten la jugabilitat. 'L'Ambient' com a les condicions socials i físiques en la que s'ha desenvolupat el joc i finalment un 'Mode' que explica la forma en la qual presentes el món als jugadors. Aquesta última propietat és la que quadra millor amb els jocs basats en text.

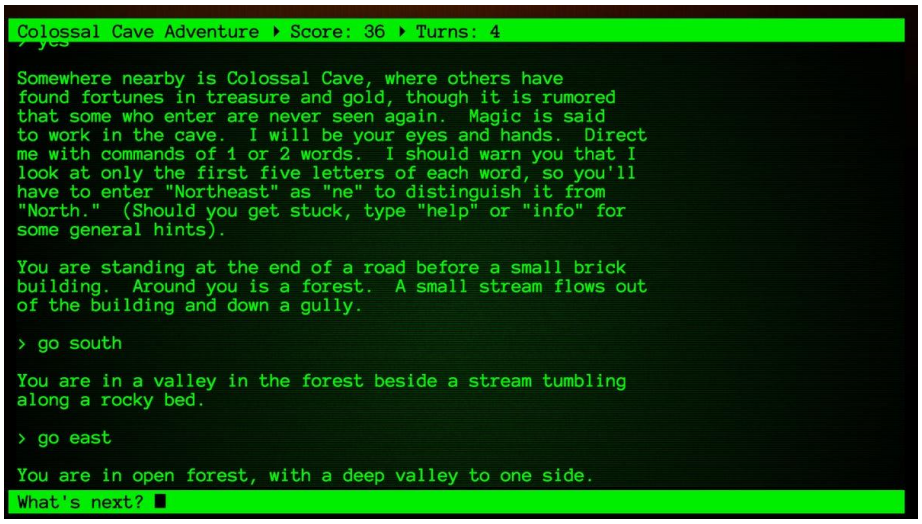
De fet, segons una anàlisi dels diferents gèneres que els videojocs han mostrat al llarg de la història (Wolf M. J., 2000) els jocs basats en text són representats en diferents categories. Els més famosos i els que val la pena analitzar són les Text Adventures i els MUDs.

4.1.1.1 Text Adventures o Interactive Fiction

Molts articles acadèmics debaten de forma exhaustiva en intentar definir aquest gènere. El problema principal és que el concepte de *interactive fiction*, que sovint s'utilitza com a sinònim per a les *text adventures*, és un terme complicat i que suposa unes característiques molt específiques. Per exemple, l'autor Nick Montfort estipula que la *interactive fiction* és un programa interactiu on la seva forma d'interacció és principalment textual (Montfort, 2013). En el mateix article es defineixen més endavant algunes de les propietats que suposadament són importants pel gènere: Un agent que tradueix el llenguatge escrit en accions per l'ordinador anomenat "*parser"* i un model de món virtual. Altres obres del mateix autor (Monfort, 2011) defineixen que aquests mons estan dividits de forma discreta en "habitacions" o "àrees". Finalment, una altra

propietat que es destaca sovint fa referència al jugador, que pren el rol de "*interactor*". El "*interactor*" és qui porta el control del temps de joc. Un programa de *interactive fiction* es pot quedar hores sense canviar el seu estat si no rep cap comanda del seu usuari.

Un joc que es destaca com la primera obra dins del gènere es titula "*Colossal Cave Adventure*" o "*Adventure*". Publicat l'any 1977, el joc situa al *"interactor"* dins d'un sistema de coves a Kentucky a la recerca de tresors. Les accions del joc sorgeixen a partir de petites frases que l'ordinador s'encarrega d'interpretar com "*Go North*" (Ves cap al Nord) o "*Take Hammer*" (Agafa el martell). Al ser una de les primeres expressions del gènere, el seu disseny ha estat força criticat (Sorolla, 1996) per la seva dificultat en alguns dels seus puzles, poder-se quedar en un estat on és impossible guanyar o accions sense context que ningú et diu que pots realitzar.



Imatge 14: Captura d'una sessió de joc de "*Colossal Cave Adventure*". Elaboració pròpia.

4.1.1.2 Multi-User Dungeons

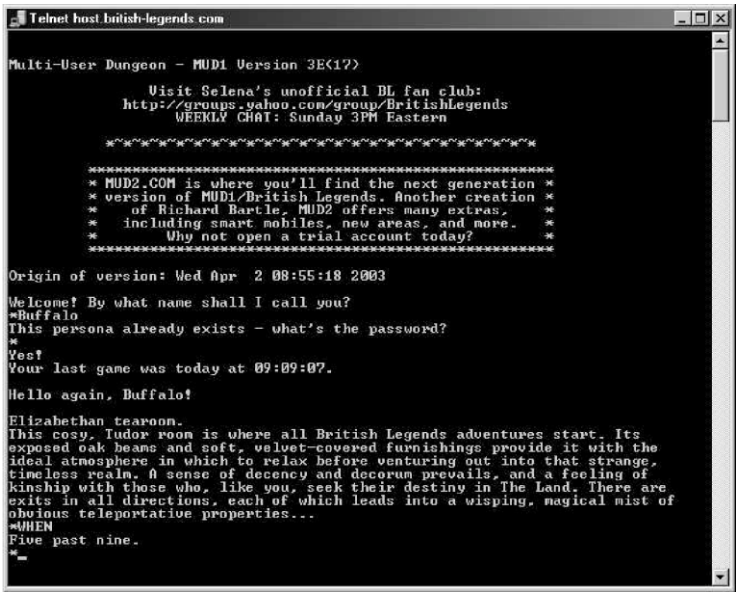
Alan Cox defineix els *Multi-User Dungeons* (*MUDs*) com programes que contenen un món virtual accedit via text i que accepten connexions simultànies de diferents usuaris (Cox & Campbell, 1994). La paraula món virtual obté una gran rellevància en aquest gènere, ja que no només ha d'oferir una sèrie de regles sinó que ha de permetre

aquesta pluralitat de jugadors. Bartle, màxim exponent i cocreador del primer *MUD,* anomena algunes propietats d'aquests mons virtuals (Bartle R. , 2010):

- Automatitzats: El món té un set de propietats (físiques) que determinen què poden canviar els jugadors.
- Compartits: Més d'un jugador pot estar en un mateix món virtual i ha de permetre que es puguin comunicar
- Persistents: Si surts del joc el món segueix estan allà.
- En temps real: Genera esdeveniments o canvia el seu estat en temps real.

L'inici d'aquest gènere i el seu nom es va donar l'any 1978 quan Richard Bartle juntament amb Roy Trubshaw van desenvolupar "*MUD*". Després de múltiples intents i passar el codi de MACRO-10 a BCPL van aconseguir un prototip bastant exitós i que estudiants de la universitat de Essex anhelaven per a poder jugar. Això va inspirar altres desenvolupadors a crear els seus propis *MUDs*, com és l'exemple de "*Shades*" o *"MirrorWorld*", però el punt d'inflexió no es va produir fins a la creació de "*AmberMUD*". Aquest joc estava escrit en el llenguatge C, cosa que permetia poder executar-lo en màquines UNIX i que a part començaven a estar connectades a Internet. A partir d'aquí els *MUDs* van seguir expandint-se, amb títols molt importants com "*TinyMUD"* o "*DikuMUD".* La part més interessant és que cada desenvolupador emfatitzava diferents característiques en la seva simulació de món, creant així diversos estereotips de *MUDs* (Bartle R. A., 2003):

- *DikuMUDs*: Orientats a l'aventura i en el combat contra enemics controlats per ordinador. Mons generalment estàtics
- *MUCKs*: Orientats a l'aspecte social i al *role play*.
- *MUSHes*: També orientats a l'aspecte social, però no tant al *role play*.
- *LPMUDs*: Orientats a l'aventura però sense un èmfasi als combats.
- *MOOs*: Orientats a la creació i al *scripting* de nou contingut. Genera experiències més originals però no tan destinades a jugar.



Imatge 15: Exemple de la interfície de “MUD” o “MUD1” creat per Bartle i Trubshaw. Imatge per P. Fuhrer.

4.1.2 Aspectes de disseny

Ara que s’han examinat dos dels possibles gèneres on els jocs basats en text són representats, el següent pas és elegir quin encaixa més tenint en compte l’entorn en el qual treballa Discord.

Per una banda, els dos gèneres tenen una forta motivació per a la creació de mons virtuals, però els *text adventures* presenten unes importants limitacions. De fet, l’article *Command Lines* (Douglas, 2007) argumenta que els MUDs no poden ser considerats *text adventures* per dues simples raons: La primera és que el gènere es basa principalment en el diàleg constant entre *interactor* i el parser. Els MUDs tenen un fort component social i sovint molta interacció ve donada a través d’altres usuaris. La segona és que les *text adventures* no tenen un vincle tan important al temps com els *MUDs*. Una persona pot deixar en espera un *text adventure* de forma indefinida i aquest estarà esperant resposta sense avançar el temps de joc.

Com a clausura d’aquest apartat, el concepte dels *MUDs* cobra molt més de sentit en un ambient social com *Discord*. Això dona una visió molt específica sobre quins patrons de disseny de jocs cal seguir que ajudarà durant la fase pràctica, com la teoria dels tipus de jugadors de Bartle (1996).

4.2 L'entorn

Aquest apartat té com a objectiu contextualitzar a les persones que no estan familiaritzades amb *Discord* i altres conceptes de caràcter més tècnic. D'aquesta manera es podran entendre més endavant els requisits i limitacions de l'aplicació que es vol desenvolupar.

4.2.1 Definició i interfície

Discord és una aplicació que permet connectar amb desenes de milions d'usuaris amb veu, xat i vídeo (Discord, 2023). La principal característica es la seva capacitat per a crear servidors: comunitats de gent que comparteixen algun hobby com pot ser la programació, les sèries de terror o algun videojoc.

Imatge 16: Captura de la interfície de *Discord* en un servidor. Elaboració pròpia.

La imatge 1 ens mostra com és l'aplicació per dins. Està dividida en dues seccions principals: Missatges privats i *guilds.* Els missatges privats són xats personals amb altres usuaris o amics. Les *guilds* o servidors són una col·lecció d'usuaris i de canals dirigida a una comunicació més pública (Discord, Discord Developer Platform, n.d.). La llista de servidors en què un usuari forma part està situada en el quadre vermell a l'esquerra.

Cada servidor es pot estructurar de forma diferent segons les seves necessitats. Discord ofereix diferents tipus de canals com els de text, veu o fòrum (Semblant a

l'estructura de posts com Reddit). El quadre verd mostra un exemple d'aquesta estructuració d'un servidor en concret. Es poden veure diferents noms de canal que donen certa temàtica, com un canal de dubtes o de feedback. En obrir un canal de text aquest ens mostra el contingut de la conversa. D'altra banda, en obrirun canal de veu ens uneix de forma automàtica a una sala on podem parlar i obrir la webcam. En la imatge 1 es pot veure com el quadre blau fosc presenta la informació del canal *"sala-común"*.

Finalment, en la mateixa imatge el quadre blau cel ens mostra la llista d'usuaris i el seu rol. Un rol no és més que una col·lecció de permisos lligats a un usuari (Discord, Discord Developer Platform, n.d.). Un exemple molt comú són els canals d'administradors on només poden ser vistos per aquella gent amb el rol d'administrador i que s'encarreguen de moderar i vigilar els canals públics de gent que pot importunar a altres usuaris.

4.2.2 Tecnologies

Per entendre el funcionament a més alt nivell de les llibreries que es faran servir i què és el que solucionen, és important mencionar algunes de les tecnologies que utilitza Discord per funcionar:

Tecnologia	Quin problema resol?	Per a què s'usa a Discord?
REST	Adquisició de recursos estandarditzada	Registre de comandes i altres peticions que no requereixen connexió persistent.
WebSockets	Comunicació persistent bidireccional via TCP entre un client i un servidor	Connexió a la "Gateway", una API que gestiona esdeveniments que succeeixen dins de Discord (missatges nous, mencions, contestacions)
Oauth2	Seguretat i identificació de clients en la utilització de recursos	Enregistrament de bots dins de Discord i gestió de permisos
WebRTC	Comunicació en temps real de veu i vídeo	Implementació dels serveis d'audio i vídeo

Taula 1: Breu contextualització de les tecnologies i el seu ús a Discord. Elaboració pròpia.

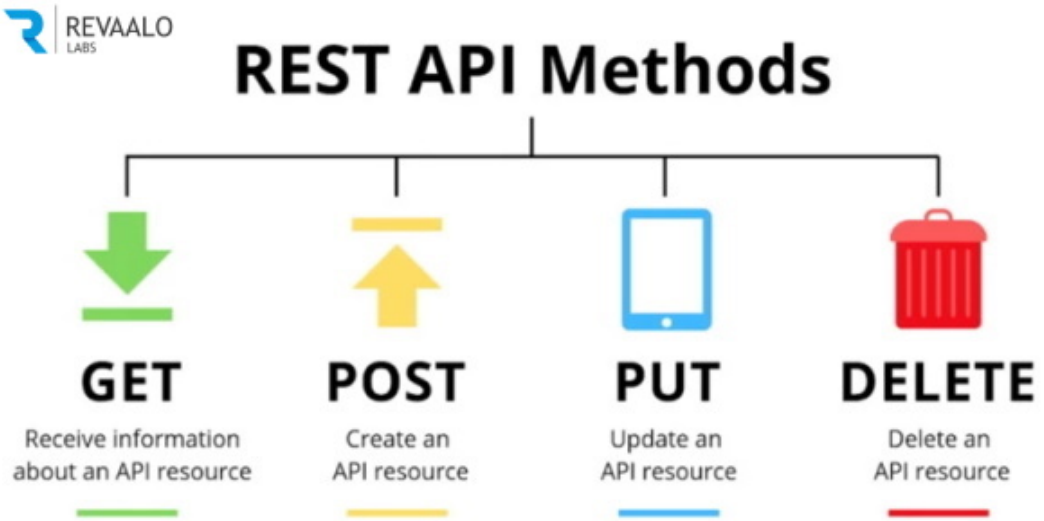
4.2.2.1 REST

El protocol REST (*Representational State Transfer*) és una arquitectura que permet l'adquisició estandarditzada de recursos entre un client i un servidor (Gupta, 2022). Per garantir una bona comunicació l'API REST ha de permetre, entre moltes altres coses, una connexió *stateless* (Tota la informació és enviada en una sola comanda, no hi ha estats entremitjos) i una arquitectura per capes.

Quan un client fa una petició a una API que utilitza el protocol REST, generalment està demanant realitzar una acció sobre un recurs en concret. Per exemple, en el cas de Discord un bot pot demanar l'últim missatge enviat en un canal en específic. Demanar és una acció i el recurs és el contingut del missatge.

Algunes de les accions més utilitzades són:

- GET: Adquirir informació d'un recurs
- POST: Generar un nou recurs
- PUT: Actualitzar la informació d'un recurs
- DELETE: Eliminar un recurs

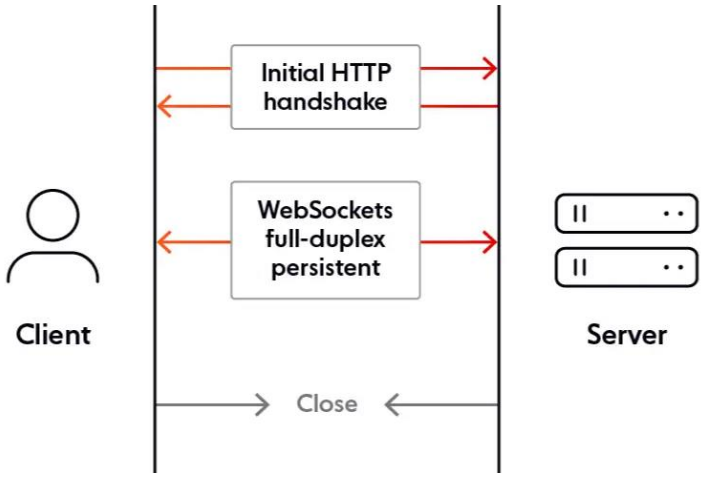


Imatge 2: Els diferents tipus de mètodes en una API REST. Imatge *per Revaalo Labs*.

4.2.2.2 WebSockets

En un article del RFC (Fette & Melnikov, 2011) es defineix els WebSockets com un protocol de comunicació que permet una connexió a bidireccional persistent entre un client i un servidor basat en TCP (*Transmission Control Protocol*). Bàsicament, permet una connexió estable on els dos agents poden enviar-se alertes i notificacions en temps real.

Per establir una comunicació via WebSockets, la documentació defineix que primer cal realitzar un procés de *handshake* amb HTTP. Durant aquest procés, el client pregunta al servidor si està operatiu i en cas afirmatiu aquest li respon amb un enllaç per a començar la connexió.



Imatge 17: Establiment d'una connexió via WebSockets. Imatge per Ably Realtime.

4.2.2.3 Oauth2

Oauth2 és un protocol que permet als clients obtenir permisos d'aplicacions externes mitjançant "tokens" (Hardt, 2012). D'aquesta forma no cal demanar ni guardar contrasenyes específiques per a cada aplicació. Oauth2 és el protocol de seguretat estàndard de la indústria (Oauth, n.d.).

Per fer servir aquest protocol i resumit a partir de la documentació oficial (Hardt, 2012), l'aplicació primer ha de demanar utilitzar un recurs privat. En el cas de Discord, una aplicació externa pot fer una petició a la seva API per demanar el correu electrònic d'un usuari. Posteriorment, el servidor d'autentificació demana a l'usuari si vol concedir

a l'aplicació externa amb els permisos que demana. En l'exemple anterior està demanant un permís d'obtenció de correu electrònic. Si l'usuari accepta els permisos el servidor d'autentificació atorga a l'aplicació d'una clau especial per a poder fer ús de l'API.

Imatge 18: Diagrama de flux del protocol Oauth2. Imatge per D. Hardt

4.2.2.4 WebRTC

WebRTC o Web Real Time Communication és una tecnologia que permet afegir a les aplicacions la capacitat d'una comunicació en temps real (Google, n.d.). Està disponible a tots els buscadors moderns i permet l'enviament de dades a partir d'una connexió directa d'ordinador a ordinador (*Peer to Peer).*

A través de la documentació oficial (World Wide Web Consortium, 2021) podem entendre a grans trets com funciona aquest protocol:

4.2.2.4.1 Fase 1: Signaling Stage (Establiment de la connexió):

Un dispositiu anuncia que vol començar una connexió, com per exemple una trucada per *Zoom*. Per fer-ho, penja en un servidor una "oferta" amb la informació necessària per a que altres clients es puguin connectar a ell. Aquesta informació també està estandarditzada i s'anomena SDP (Session Description Protocol).

Quan un altre dispositiu veu l'oferta i decideix unir-se a la connexió, aquest penja també la seva informació. D'aquesta forma el servidor s'encarrega de mantenir un pont entre els dos clients i no li cal tocar les dades que s'envien. Aquest últim punt és

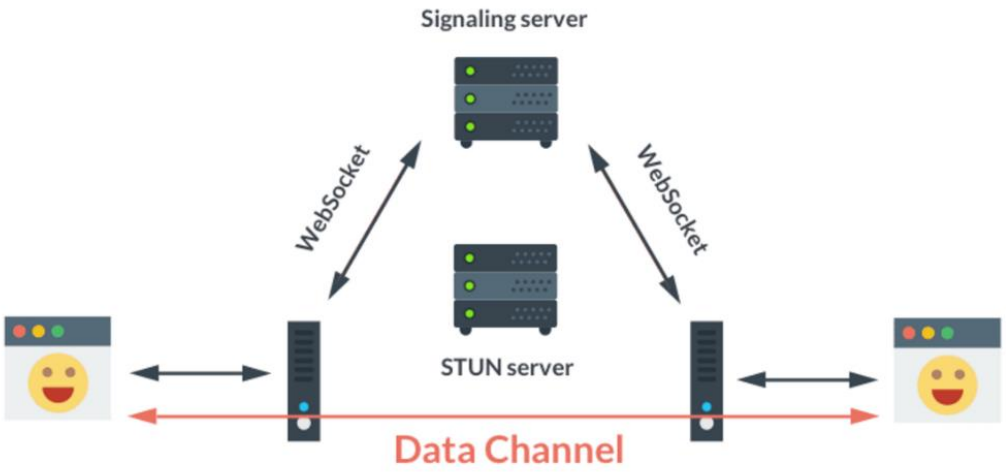
important, ja que les dades que se solen enviar mitjançant WebRTC són d'un pes considerable i enviades en temps real.

4.2.2.4.2 Fase 2: ICE Gathering Stage (Interactive Connectivity Establishment):

Per a mantenir una connexió estable en un entorn com Internet, calen protocols que permetin avançar-se a possibles canvis de IP (Ocasionats per problemes de ruta o dispositius NAT, per exemple). L'estàndard ICE (Interactive Connectivity Establishment) fa justament això.

Per començar cada client s'encarrega de mantenir una llista de les seves possibles IP públiques i el seu port. Aquesta informació és aconseguida gràcies a uns servidors especials anomenats STUN. Quan un client llença una petició, aquests els hi responen amb la seva IP pública actual.

La llista de cada client amb les seves IP actualitzades és enviada al servidor de *Signaling* perquè tothom pugui saber com enviar la informació de la retransmissió de forma directa.



Imatge 5: Esquema d'una connexió via WebRTC. Imatge per Francois J Rossouw

4.2.3 Funcionament intern

Ara que s'entenen alguns dels protocols que apareixen sovint quan es revisa la documentació oficial de *Discord*, és moment d'entrar en les seves implementacions específiques dins de l'aplicació

4.2.3.1 Bots

Segons el llibre *Bots* (Monaco & Wolley, 2022) el terme prové originalment a partir d'una taquigrafia de la paraula robot i que significa "*objecte que realitza tasques humanes i que és, a la llarga, programat per una persona*".

Durant el començament de l'era de l'Internet, els dos termes van anar agafant a poc a poc connotacions diferents. Mentre que un robot té una presència física i interactua mitjançant un hardware, el bot és un programa i actua principalment per mitjà de software. Un exemple històric són els anomenats *daemons*, processos interns del sistema operatiu que mantenen l'ordinador fent tasques vitals (Monaco & Wolley, 2022).

4.2.3.1.1 Diferència entre usuaris normals i usuaris bots

La classe usuari és l'entitat més bàsica dins de l'aplicació i els principals generadors d'esdeveniments i notificacions (Discord, Discord Developer Platform, n.d.). La documentació categoritza els usuaris normals i els bots com la mateixa entitat, però amb diferències remarcables:

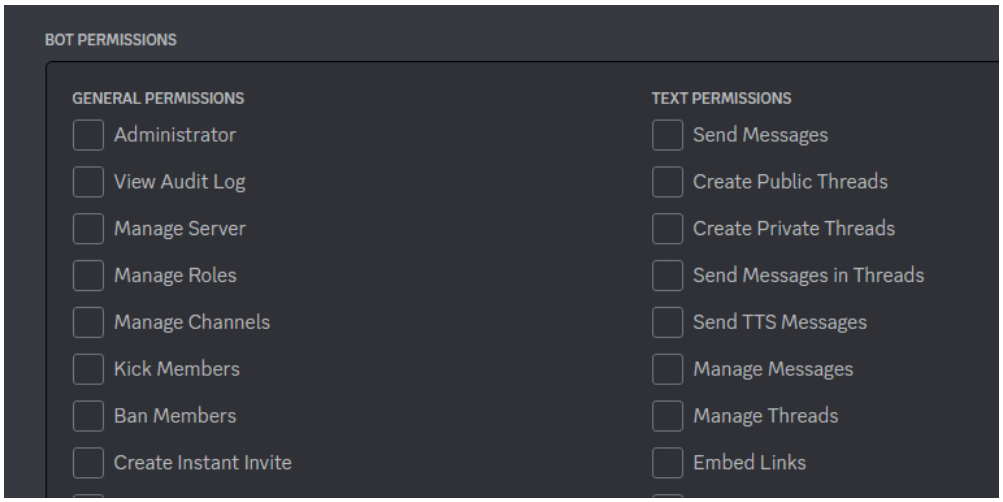
- A diferència dels usuaris normals, un bot que està verificat no té una limitació màxima de servidors als quals pot entrar.
- Un bot és sempre propietat d'un usuari normal de *Discord*.
- Els bots no poden entrar en un servidor mitjançant mètodes normals. Han d'entrar a partir d'un procediment amb *Oauth2*.
- Els bots tenen límits diferents de tràfic (*rate limits*) que els usuaris normals.
- Finalment, els usuaris normals utilitzen un client o aplicació gràfica per interactuar amb l'API mentre que els bots hi interactuen directament.

4.2.3.1.2 Registre de l'aplicació i permisos

Per tal de crear un bot, *Discord* ofereix un formulari especialitzat per a gestionar les aplicacions en la seva pàgina oficial. Una aplicació no és un bot com a tal, és un

programa amb una sèrie de permisos i que a partir d'un procediment *Oauth2* és garantit l'accés a l'API. El bot el que fa és crear un pont perquè usuaris dins de la mateixa aplicació de *Discord* puguin interactuar amb la teva aplicació. Per exemple, no caldria crear un bot en el cas que es volgués realitzar una pàgina web d'estadístiques, ja que no requereix cap interacció dins de la mateixa aplicació de *Discord*.

A part dels permisos que cedeixes a l'aplicació mitjançant el procediment *Oauth2*, si l'aplicació té un bot aquest entrarà en el servidor que tu decideixis (donat que tens els drets d'administrador). Ja que un bot forma part de l'entitat *user*, aquest també necessita uns permisos per poder operar (un bot que dinamitza la gestió de canals de veu ha de tenir permisos per a poder-los crear i eliminar). Com a desenvolupador pots marcar què necessita tenir el teu bot per poder funcionar a partir d'un número on cada bit està relacionat amb un permís (si el bit està a 1 el bot tindrà aquell permís en concret).



Imatge 19: Breu llistat d'alguns permisos que pots assignar a un bot. Elaboració pròpia.

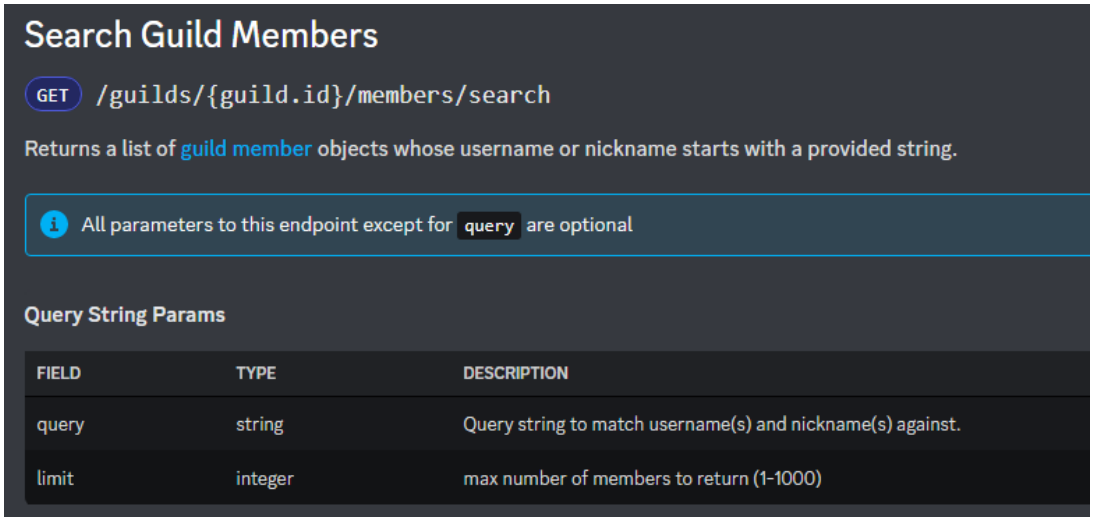
4.2.3.2 Introducció a les API

Discord, al ser una aplicació de missatgeria, necessita processar una quantitat ingent de dades a la vegada que oferir un servei d'alta disponibilitat. Els seus desenvolupadors van crear tota una infraestructura basada en dues API principals per

a suplir aquestes necessitats: l'API REST HTTP i l'API Gateway. Cada una té diferents casos d'ús i és important conèixer les seves peculiaritats:

4.2.3.2.1 API REST HTTP

És la forma estàndard de comunicació amb *Discord*. És simple però no ofereix una comunicació en temps real ni la capacitat de reaccionar a esdeveniments. A continuació a la Imatge 7 es pot veure un exemple del seu ús.



Imatge 20: Exemple d'una consulta a l'API REST

La Imatge 20 mostra una possible consulta mitjançant l'API REST especificada en la pàgina oficial per a desenvolupadors de Discord. L'operació GET indica que es vol realitzar una acció d'obtenció d'informació i que per obtenir-la s'ha d'enviar al link *"/guilds/{guild.id}/membres/search".* Aquesta operació a part requereix emplenar una sèrie de paràmetres.

Un exemple d'ús seria buscar en un servidor 20 persones que comencin amb la lletra "A" o "a". La consulta que s'enviaria en JSON quedaria de la següent forma:

```
{
    "query": "/^a",
    "límit": 10
}
```

4.2.3.2.2 API Gateway

El Gateway és una tecnologia desenvolupada per Discord a partir dels WebSockets que permet la comunicació persistent i bidireccional entre usuaris i el servidor (Vass, 2018). És una part essencial, ja que és el principal agent a l'hora d'enviar dades de text i altres esdeveniments com una notificació o un nou missatge en un canal. El client de Discord que els usuaris es descarreguen en l'ordinador o al mòbil utilitza la Gateway per actualitzar-ne els continguts i llençar avisos a l'usuari final.

4.2.3.2.2.1 Gateway Events

Els missatges o *payloads* que s'envien a través d'aquesta comunicació estan estandarditzats i els desenvolupadors han de respectar la seva estructura. Es poden enviar en format de text via JSON o codificats en binari. La seva dimensió mai pot superar els 4096 bytes o la connexió serà tancada de forma automàtica.

FIELD	TYPE	DESCRIPTION
op	integer	Gateway opcode, which indicates the payload type
d	?mixed (any JSON value)	Event data
s	?integer *	Sequence number of event used for resuming sessions and heartbeating
t	?string *	Event name

* `s` and `t` are `null` when `op` is not `0` (Gateway Dispatch opcode).

Imatge 21: Estructura d'un payload enviat per Gateway. Elaboració pròpia.

El camp "*op"* conté la categoria del missatge i la seva funció. Hi ha missatges amb l'únic objectiu de confirmar que el paquet s'ha rebut (Coneguts com *acknowledgments*) o missatges amb l'objectiu d'iniciar una reconnexió. El camp "*d*" és opcional i representa el cos de les dades del payload en qüestió.

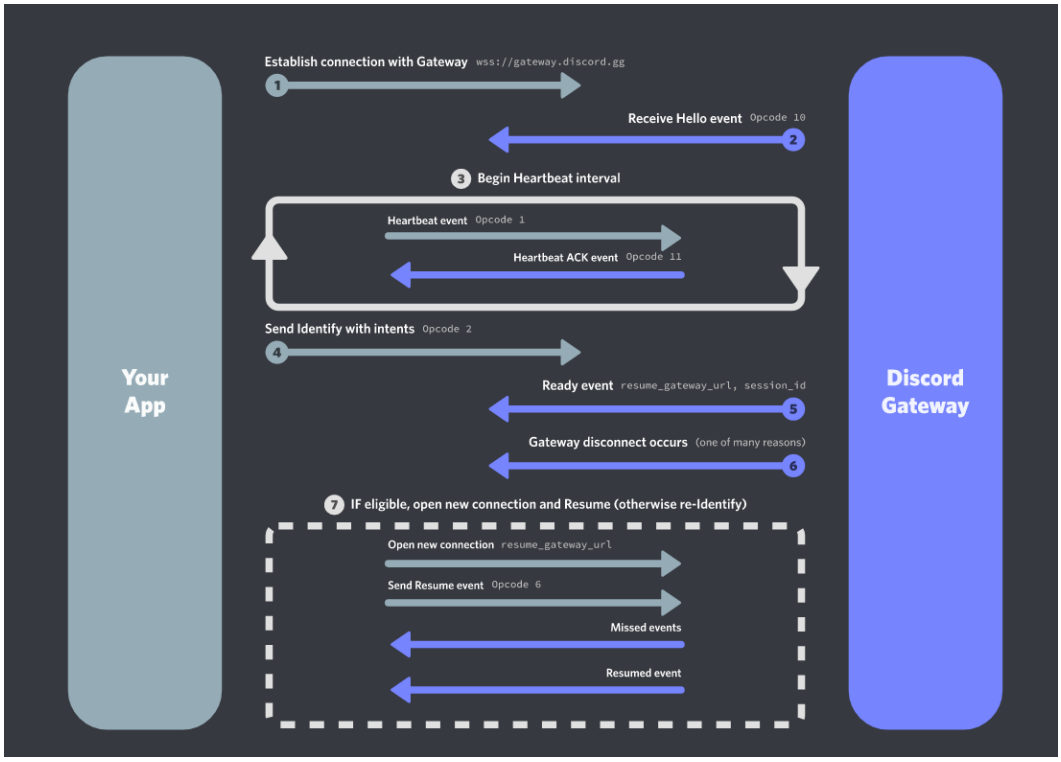
Un tipus d'esdeveniment especial és l'anomenat "*Dispatch*" identificat amb la categoria 0. Crear un missatge, entrar a un servidor o començar a escriure son exemples que entren dintre d'aquesta classificació. Per aquests casos també cal emplenar els valors de "s" i "t". "s" és un número de seqüència que identifica l'ordre d'expedició del missatge i "t" és el nom de l'esdeveniment en concret.

Amb tota aquesta informació es pot escriure un exemple de missatge que un client enviaria a la Gateway al detectar que l'usuari ha enviat un missatge en un canal específic:

```
{
    "t": "MESSAGE_CREATE",
    "s": 1234,
    "op": 0,
    "d":
    {
        "id": "1234567890123456678",
        "channel_id": "9876543210987654321",
        "author":
        {
            "id": "1234567890123456678",
            "username": "usuari_exemple",
            "avatar": "abcdef1234567890"
        }
    }
}
```

4.2.3.2.2.2 Establiment de connexió

Per a poder-se connectar a la *Gateway*, *Discord* ofereix un procediment de *handshake* estandarditzat. Per sort en la majoria dels casos no cal implementar-ho de forma manual donat que hi ha llibreries amb la solució ja desenvolupada, però és important conèixer el procés genèric.



Imatge 22: Procediment de connexió al servidor de la Gateway

Primer de tot cal obtenir el link del Gateway mitjançant una consulta a l'api HTTP REST. Aquest link va canviant i conté informació addicional sobre l'estat de l'API i la quantitat màxima de processos que es poden connectar a la vegada. Un cop fet el primer establiment (1), l'API contesta amb un esdeveniment de categoria *"Hello"* perquè comenci el procés de *heartbeat*. (2)

Example Hello Event

```
{
  "op": 10,
  "d": {
    "heartbeat_interval": 45000
  }
}
```

Imatge 23: Payload d'un esdeveniments de categoria Hello

El procés de *heartbeat* no és més que una sèrie de missatges de categoria "Heartbeat" que ajuden a saber si l'aplicació connectada segueix estan operativa. Si es mira la imatge 10, es pot veure el camp *"hearbeat_interval"*. Aquest camp indica amb quina

freqüència en mil·lisegons s'ha d'enviar un *ping* a la Gateway informant que es vol mantenir la connexió. Al rebre aquest missatge la Gateway confirma que l'ha rebut mitjançant un missatge de categoria "*Ack*" o "*Acknowledgement*" (3).

Un cop l'aplicació està mantenint els *heartbeats* amb la Gateway, toca el procés d'identificació (4). Aquest missatge conté la informació necessària per a que la teva aplicació sigui verificada i, per tant, pugui començar a interactuar amb els *endpoints*. Aquí també s'expressen els "intents", un identificador en el qual expresses quines accions o esdeveniments es volen fer servir i quins es volen ignorar. D'aquesta forma la comunicació és molt més eficient.

Una tècnica molt utilitzada junt amb els intents és el *sharding*. Aquesta tècnica es basa en especialitzar diferents processos de la mateixa aplicació a través dels intents. D'aquesta forma, pots tenir dues instàncies només dedicades a gestionar esdeveniments més comuns com els de missatges i una instància als esdeveniments més residuals.

Després que l'aplicació hagi estat identificada la connexió és acceptada mitjançant un missatge de categoria "*dispatch*" amb el nom de "*ready*" (5). Aquest esdeveniment conté a part un enllaç privat perquè l'aplicació es pugui connectar de forma immediata en cas que hi hagi una interrupció (6).

El missatge que cal llençar per a tornar-se a connectar és de categoria "*resume*" i permet no només realitzar l'establiment sense la necessitat d'identificar-se sinó que l'aplicació rebrà una llista d'esdeveniments que no s'han consumit des de la seva desconnexió (7).

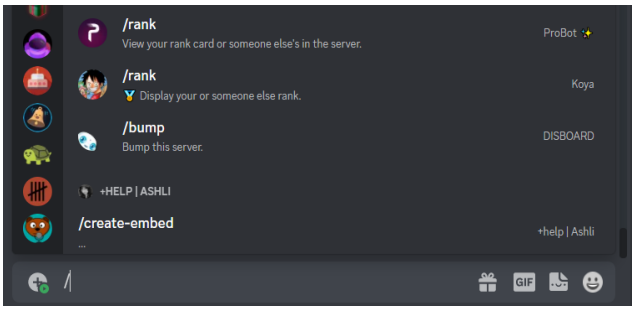
4.2.3.2.3 Interactions

En un primer moment la forma en que un usuari s'havia de comunicar amb un bot no estava estandarditzada. Això donava bastants problemes quan un servidor amb múltiples bots tenien la mateixa comanda. De forma temporal, es va optar per a fer servir prefixos per a diferenciar-los: El bot A utilitza "?play" i el bot B utilitza "!play". Aquesta forma d'interacció primitiva no només era incòmode pels usuaris sinó que obligava als bots ha d'estar constantment llegint cada missatge creat per si contenia el seu prefix.

El nou sistema d'interaccions de *Discord* és un intent per a normalitzar tota aquesta part tan important per a la comunitat. De fet, són l'exemple més important on es veuen involucrades les dues API (HTTP REST i Gateway). Per utilitzar aquesta nova funcionalitat cal indicar a l'aplicació que demani els permisos corresponents de creació i execució de comandes via Oauth2.

4.2.3.2.3.1 Application Commands

La millora més substancial de cara el sistema antic és la possibilitat d'actualitzar les possibles accions que pot realitzar el bot de forma nativa amb *Discord* a través de l'API HTTP REST. A més, la interfície gràfica s'adapta de forma automàtica a la informació de la comanda per a poder guiar als usuaris en el seu ús. Un cop registrats les comandes amb l'acció "PUT", el bot llegirà tots els esdeveniments vinculats a través de la Gateway. A continuació es pot veure una imatge d'aquesta integració:



Imatge 24: Interfície gràfica del nou sistema d'interaccions. Elaboració pròpia.

Discord defineix un total de tres tipus de comandes:

- ***Chat Input*:** Comandes més utilitzades. Són les que utilitzen un missatge via text per a definir una operació. Com es pot veure a la imatge 11, un usuari pot escriure *"/play <song>"* on song és el nom d'una cançó per indicar al bot que s'uneixi al canal de veu i comenci a reproduir-la.
- ***User*:** Aquestes comandes es poden invocar a un usuari en concret fent clic dret sobre el seu nom. Per exemple podries enviar una petició de fer una partida a un usuari sense la necessitat d'escriure res.

- ***Message*:** El mateix efecte que els *User Commands*, però l'acció s'aplica a un missatge. Un exemple seria demanar a un bot que et tradueixi el contingut d'un text al castellà.

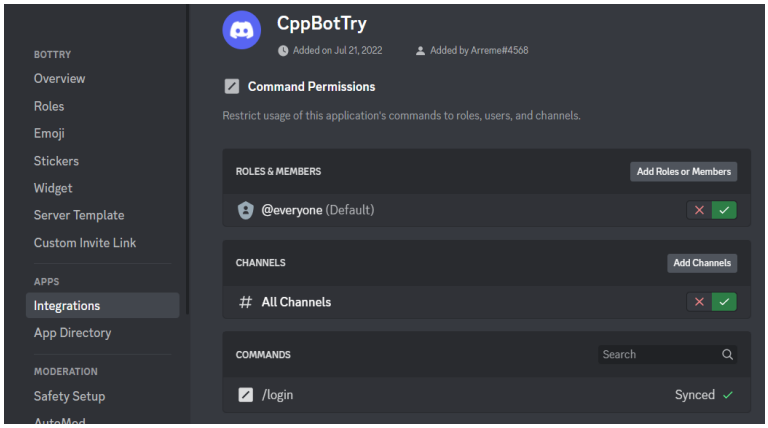
A part d'aquesta categorització per tipus d'invocació, *Discord* també permet categoritzar-los en funció de si són globals o "guild-specific". Les comandes globals estan disponibles a tots els servidors que el teu bot hi forma part automàticament. Les "*guild-specific*" s'activen només a un servidor en concret i això dona peu a possibles usos per a la monetització i a interaccions de pagament.

	Global	Guild
Chat Input	100	100
User	5	5
Message	5	5

Taula 2: Número màxim de comandes per aplicació. No pots repetir el nom d'una comanda en la mateixa categoria. Elaboració pròpia.

El formulari de creació d'una comanda requereix certes dades importants per a que els usuaris puguin utilitzar la interacció de la forma més còmode possible. Primer cal emplenar un nom que tingui relació amb l'acció i una descripció sobre aquesta. Es poden afegir textos de localització perquè el bot canviï de llenguatge a partir de la configuració del client. Finalment, una comanda pot tenir opcions ja definides. Per exemple, una interacció que només pugui mostrar fotos de gats o de gossos pot tenir les opcions prefixades en ella i així els usuaris saben el que poden fer o no.

Com a punt final, les comandes tenen rols i permisos. Un administrador pot configurar que una interacció requereixi un accés especial o limitar el seu ús per a usuaris que porten menys de 10 dies en el servidor.



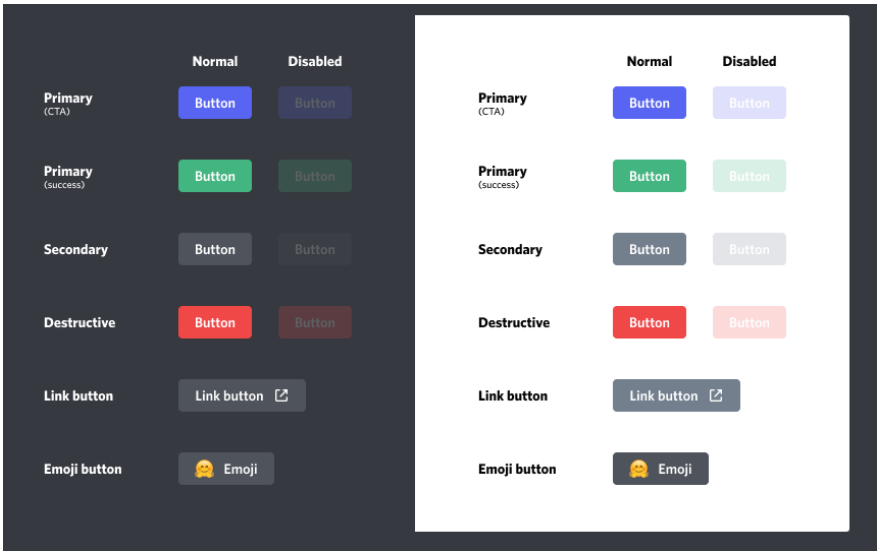
Imatge 25: Interfície de configuració de permisos. En aquest cas el bot està actiu per a tots els usuaris. Elaboració pròpia.

4.2.3.2.3.2 Message Components

A partir d'aquesta nova interfície per a la creació de comandes *Discord* també va aprofitar per afegir els *Components*, un *framework* per afegir elements interactius als missatges que el teu bot envia (Discord, Discord Developer Platform, n.d.). D'aquesta forma la teva aplicació aconsegueix ser més interactiva.

Hi ha diferents tipus de components i poden ser afegits de forma molt fàcil a partir d'un nou camp "components" els quals conté una llista *d'action row components* (Component per emmagatzemar components).

Els botons són els components més utilitzats, ja que permeten realitzar accions amb un simple clic. També n'hi ha per a obrir un menú desplegable amb opcions de text, rols o usuaris i inclús un component que obre un formulari de text privat.



Imatge 26: Disseny de botons que permet en aquest moment Discord. Elaboració pròpia.

4.3 L'aplicació

Fins ara s'ha revisat breument en quin entorn es realitza el treball i quin joc encaixa millor en aquest entorn. Per acabar el marc teòric cal entendre algunes eines que es faran servir per a la implementació de l'aplicació i quines necessitats satisfan.

4.3.1 *D++ (DPP)*

D++ és una llibreria minimalista construïda amb C++ per a la interacció amb Discord i les seves funcionalitats (D++, 2023). Ofereix una sèrie d'algorismes que s'ocupen d'implementar els protocols i especificacions de Discord i embolicar-los en funcions. A part, ofereix *thread safety* en la gran majoria de les seves operacions.

Per exemple, el codi per a voler enregistrar un nou *application command* quedaria de la següent forma:

```
bot.on_ready([&bot](const dpp::ready_t & event) {
    if (dpp::run_once<struct register_bot_commands>()) {

        /* Create a new global command on ready event */
        dpp::slashcommand newcommand("blep", "Send a random adorable animal photo", bot.me.id);
        newcommand.add_option(
                dpp::command_option(dpp::co_string, "animal", "The type of animal", true).
                    add_choice(dpp::command_option_choice("Dog", std::string("animal_dog"))).
                    add_choice(dpp::command_option_choice("Cat", std::string("animal_cat"))).
                    add_choice(dpp::command_option_choice("Penguin", std::string("animal_penguin")
                )
            )
        );

        /* Register the command */
        bot.global_command_create(newcommand);
    }
});
```

Imatge 27: Registrar un nou *command* a l'aplicació, elaboració pròpia.

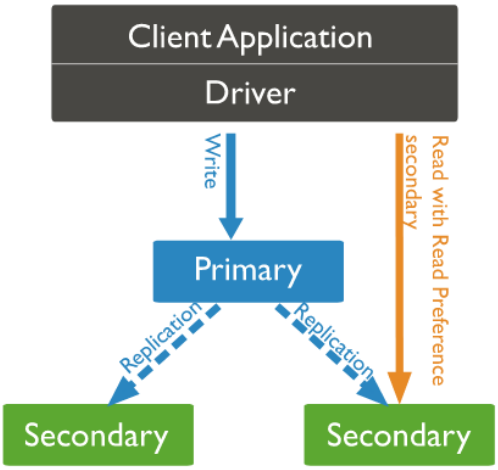
L'objecte *"bot"* és de tipus *clúster* i s'encarrega de crear instàncies de tipus *shard* de forma dinàmica. Aquestes últimes són les responsables de mantenir la connexió amb la Gateway i de donar servei a un nombre limitat de servidors. Si l'aplicació creix el desenvolupador és qui ha de crear més clústers fent ús de tècniques concurrents per així poder suplir la demanda.

Quan el clúster finalment detecta que els seus "*shards*" s'han inicialitzat, l'esdeveniment "on_ready" executa la funció lambda corresponent. En el cas de la , la funció defineix un nou "*slashcommand*" amb l'identificador "*blep*" i un seguit d'animals com a opcions per emplenar. Finalment, es registra la comanda dins del grup de comandes globals descrita en la Taula 2.

4.3.2 *MongoDB*

MongoDB és una base de dades documental que emmagatzema la informació en forma d'arxius JSON flexibles. Això significa que la seva estructura pot variar entre documents i que la seva informació és fàcilment traslladable als objectes del codi (MongoDB, 2023). L'ús d'una base de dades és indispensable per aquest tipus d'aplicacions, ja que si es vol realitzar un MUD la principal mecànica és que els usuaris puguin guardar el seu estat en el món virtual.

Per poder funcionar, MongoDB primer requereix una instància de la base de dades. Per a poder realitzar transaccions (Un número determinat d'accions de forma atòmica: o es fan totes o no es fa cap) cal activar un clúster especial anomenat Replica Set. A l'activar-ho, la base de dades "primària" crea instàncies "secundàries" amb les que constantment intentarà validar-se. D'aquesta forma la instància primària podrà fer un rollback amb una secundària en cas que la transacció sigui errònia.



Imatge 28: Esquema d'una instància de MongoDB amb Replica Set, per MongoDB.

Dins d'una instància hi ha un conjunt de col·leccions i aquestes a la vegada són un conjunt de documents. Cada col·lecció permet definir índexs: Identificadors que ordenen en forma d'arbre els documents per a poder-los buscar més fàcilment. També es poden implementar validacions i identificadors únics per evitar documents invàlids o repetits.

4.3.2.1 Accions

Per a modificar la informació present en la base de dades MongoDB defineix una sèrie d'accions. En la taula següent s'explica de forma breu les més importants:

Acció	Utilitat
Insert	Afegir un o variïs documents a la base de dades
Update	Actualitzar un o variïs documents només si coincideixen amb la consulta enviada
Find	Retorna un o variïs documents si coincideixen amb la consulta enviada
Delete	Eliminar tots aquells documents que coincideixen amb la consulta enviada
Aggregation	Consultes complexes que permeten retornar informació transformada o calculada
Operators	S'utilitzen per a transformar o filtrar documents.

Taula 3: Llistat d'accions comunes amb MongoDB. Elaboració pròpia.

4.3.2.2 Realització de consultes

Per a fer consultes amb MongoDB, s'han de seguir una sèrie de normes de la mateixa forma que amb una base de dades SQL tradicional. La primera norma és que l'estructura bàsica de tota operació segueix el format de dades JSON. A partir d'aquí, la forma més fàcil de realitzar una consulta o query és comparant valors en funció de diferents camps en el model de dades de la col·lecció. Per exemple, la consulta "{quantity: 20}" retorna tots aquells documents que el seu camp "quantitat" sigui igual a 20. Si per una casualitat es volguessin obtenir tots els documents amb un valor de quantitat més gran que 20 i amb un camp "tipus" amb el valor "casa" la consulta resultant seria del tipus: { tipus: "casa",quantity: {$gt: 20}}. En aquest cas s'ha introduït un "*operator*", un camp especial que comença amb el caràcter "$" i que permet realitzar modificacions o afegir lògica en la consulta.

4.3.2.3 *Mongocxx* i *bsoncxx*

MongoDB permet interactuar amb la base de dades de forma manual a partir d'una terminal anomenada *MongoShell*. Per a poder realitzar operacions des del codi de forma automatitzada cal l'ús d'un *driver*, específic per a cada llenguatge. En el cas de C++ el *driver* recomanat és "*mongocxx*". Aquesta llibreria és oficial i està construïda a partir de *mongoc*. Requereix l'ús d'un compilador de C++11 com a mínim (MongoDB, 2023).

Tot i així, encara hi ha un requisit extra a tenir en compte: Els objectes declarats dins de C++ no es poden guardar directament com a informació en la instància de la base de dades ja que MongoDB utilitza una forma especial per a codificar-la que requereix BSON (Binary JSON). Contràriament al JSON on les dades estan escrites en format de text, BSON les codifica en binari i permet una lectura més eficient (BSON Specification, sense data). "*Bsoncxx*" va inclosa dins de "*mongocxx*" i és qui ajudarà en aquest procés de transformació.

4.3.3 Protocol Buffers

Google (2023) defineix el protocol buffers com una forma de serialitzar dades sense tenir en compte el llenguatge o la plataforma. El desenvolupador defineix un esquema a partir d'un arxiu especial ".proto" i la llibreria l'exporta i el tradueix al llenguatge que es vulgui. A continuació es pot veure un exemple:

```
message Person {
  string name = 1;
  optional int32 id = 2;
  optional string email = 3;
}
```

En aquest cas s'està definint un objecte Persona que requereix un nom de tipus string i opcionalment un id i un email. També es poden definir enumeracions o inclús estructures més complexes com missatges dins d'altres missatges.

Quan s'exporta a C++, aquesta llibreria crea un fitxer .h i un fitxer .cc per cada .proto. Les implementacions que genera automàticament permeten:

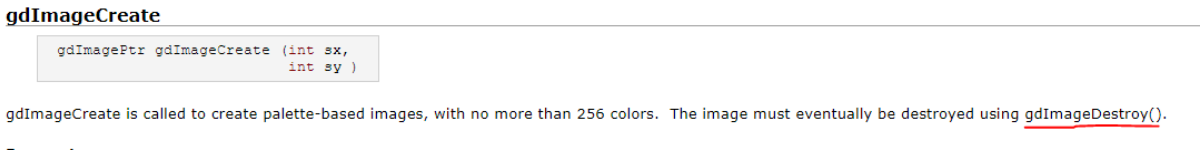
- Crear i destruir objectes
- Realitzar funcions de tipus set i get per a cada valor definit en el missatge

- Serialització i lectura automàtica d'arxius binaris o JSON
- Validació i comprovació de valors dins d'enumeracions

4.3.4 LibGD

La llibreria GD és una col·lecció de funcions i eines escrita en C que permet l'edició d'imatges o la generació de noves completament a través de codi. Es va optar per aquesta opció ja que a més de ser extremadament lleugera (Boutell & Joye, sense data) dona solució a totes les necessitats del projecte. Cal tenir en compte però que aquesta característica també comporta els seus sacrificis: Moltes funcions d'exportació, importació o generació de text necessiten ser enllaçades amb llibreries externes. Això significa que hi ha programada la forma d'exportar un arxiu PNG però que per a executar-la cal realitzar primer la compilació i l'enllaç de la llibreria oficial que implementa totes aquestes operacions, anomenada libPNG. Aquesta pràctica és molt comuna ja que permet a l'aplicació destí escollir quines característiques específiques necessita. Per exemple, en el cas de l'aplicació que es vol realitzar, es poden desactivar totes aquelles funcions d'exportació per a imatges de tipus JPEG o GIF.

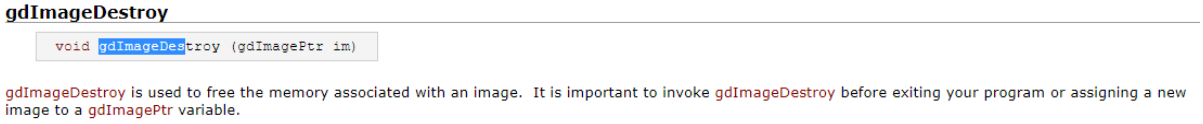
Inicialment es va plantejar un canvi donada la diferència de llenguatge, però després de realitzar les proves pertinents es va veure que no afegia gaire complexitat extra. El més important a tenir en compte però és la filosofia del llenguatge C pel que fa a gestió de memòria. En la documentació es pot observar que diferents inicialitzadors de les imatges produeixen punters a assignacions dinàmiques de memòria. Això implica que la persona encarregada d'alliberar-les és el programador. No fer-ho pot ocasionar un error anomenat "Memory Leak"



gdImageCreate

```
gdImagePtr gdImageCreate (int sx,
                          int sy )
```

gdImageCreate is called to create palette-based images, with no more than 256 colors. The image must eventually be destroyed using gdImageDestroy().

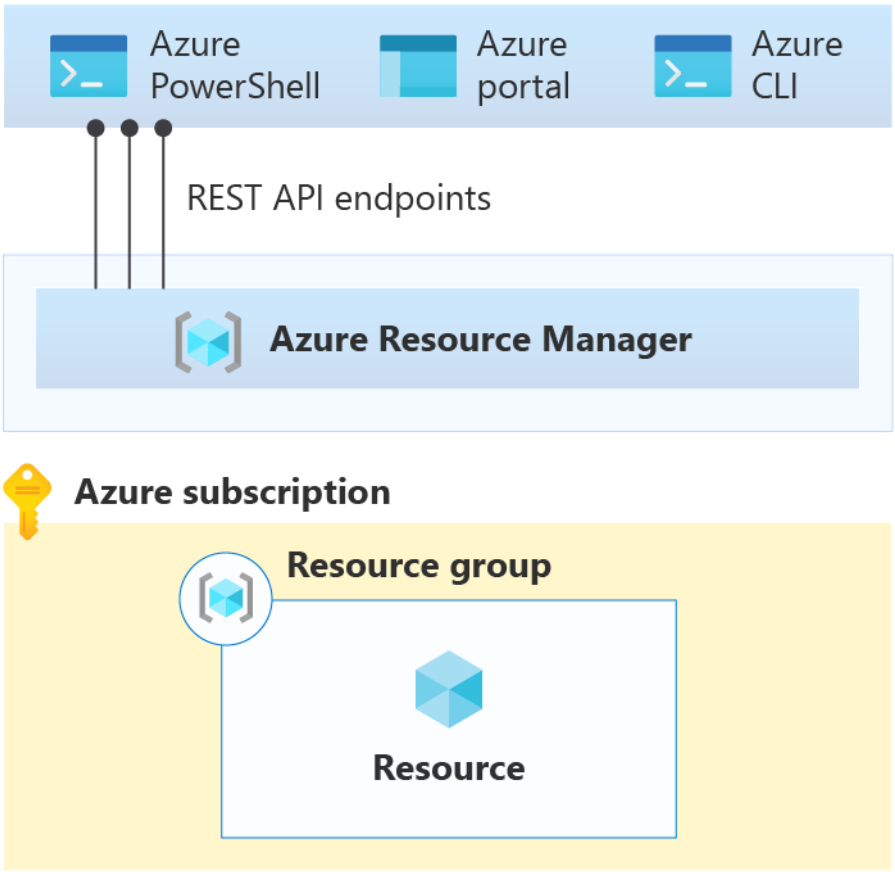


Imatge 29: (A dalt) gdImageCreate genera l'asignació dins la memòria dinàmica. gdImageDestroy s'encarrega d'alliberar-la. Elaboració pròpia.

4.3.5 Azure

Azure és una plataforma cloud de la companyia Microsoft que consisteix en una sèrie de productes i serveis que ajuden als desenvolupadors a gestionar les seves pròpies solucions (Microsoft, 2023).

El funcionament intern és molt similar a una estructura per capes on cada agent que participa en el procés gestiona un o més grups de la capa inferior. Per començar, el component fonamental amb què es basa tota la plataforma són els "resources". La documentació oficial (Resource access management in Azure, 2023) estipula que aquestes entitats són gestionades per Azure dins de servidors virtuals i poden ser qualsevol tipus de servei com una màquina virtual o un grup de funcions que executen esdeveniments (Function As A Serice). Per a gestionar aquests recursos, s'utilitzen uns objectes especials anomenats "Resource groups". D'aquesta forma es poden assignar propietats similars a totes les entitats del grup a la vegada. Anant cap a un nivell més exterior entren en joc les Subscripcions, que permeten gestionar el cost de cada "Resource Grup". Amb això els desenvolupadors poden tenir una visió clara de l'estat de les seves aplicacions i quan estan pagant al final de mes per elles. Per acabar, tot aquest sistema de recursos el controla el "Azure Resource Manger", un agent que funciona a partir d'una API REST i que serveix per a crear les diferents entitats a través del portal oficial o d'una aplicació externa.



Imatge 30: Esquema de l'aplicació cloud Azure, per Microsoft Foundation

5 Disseny metodològic i cronograma

La idea principal de l'aplicació que es vol desenvolupar és un entorn de joc virtual a l'estil dels MUDs on els jugadors puguin realitzar accions i millorar el seu personatge interactuant a través de Discord. El joc es basarà en la recol·lecció de recursos i manteniment d'una base localitzada en "pobles" (servidors). Els jugadors podran millorar les seves estructures privades per aconseguir encara més recursos. A part, també podran desbloquejar l'accés a noves zones per aconseguir més recursos i continuar explorant el mapa.

També es permetrà la lluita entre jugadors per mantenir el component online amb el que es basen els *Multi User Dungeons*.

5.1 Anàlisi de requisits

Amb això, alguns requisits funcionals que sorgeixen són:

- Desenvolupar una sèrie de comandes per a que els jugadors puguin interactuar amb el món virtual.
- Capacitat d'emmagatzemar la informació dels jugadors per a que puguin tornar més tard.
- Poder crear imatges de forma dinàmica.
- Sistema de lectura d'arxius JSON per llegir definicions i dades.
- Banc d'imatges per a poder realitzar la generació.
- Servidor web per a fer un deploy de l'aplicació i llençar-la a Discord.

Els requisits no funcionals de l'aplicació són:

- Temps de reacció i càlcul acceptable per a poder mantenir un bon ritme d'interaccions.
- API clara i entenedora.
- Sistema de testing BDD per a poder assegurar una bona qualitat del codi.
- Que sigui segura i utilitzi procediments que puguin ser utilitzats en producció. La seguretat és un factor clau i cal mantenir al mínim els possibles errors o distribucions d'informació delicada.
- Disseny de joc interessant per mantenir als jugadors motivats per jugar.

5.2 Metodologia

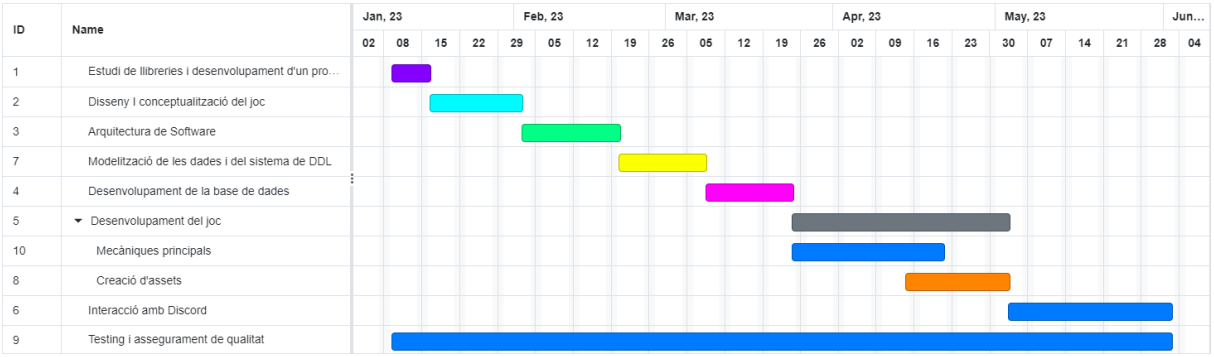
Donat el desconeixement tècnic de l'aplicació que es vol realitzar i els seus detalls, es veu adient realitzar el procés de desenvolupament amb la metodologia *Agile*. Aquesta metodologia principalment intenta perseguir un format d'entregues incrementals en comptes d'una entrega de cop ja que té en compte que els requisits van canviant a mesura que el projecte avança.

És per aquest motiu que es tindrà en compte durant la creació de l'aplicació obtenir formes per a realitzar controls de qualitat i la realització de *milestones* que permetin replantejar l'estructura del codi. Aquesta última part és important ja que es preveuen molts errors de concepte i d'arquitectura sobretot al principi.

5.3 Cronograma

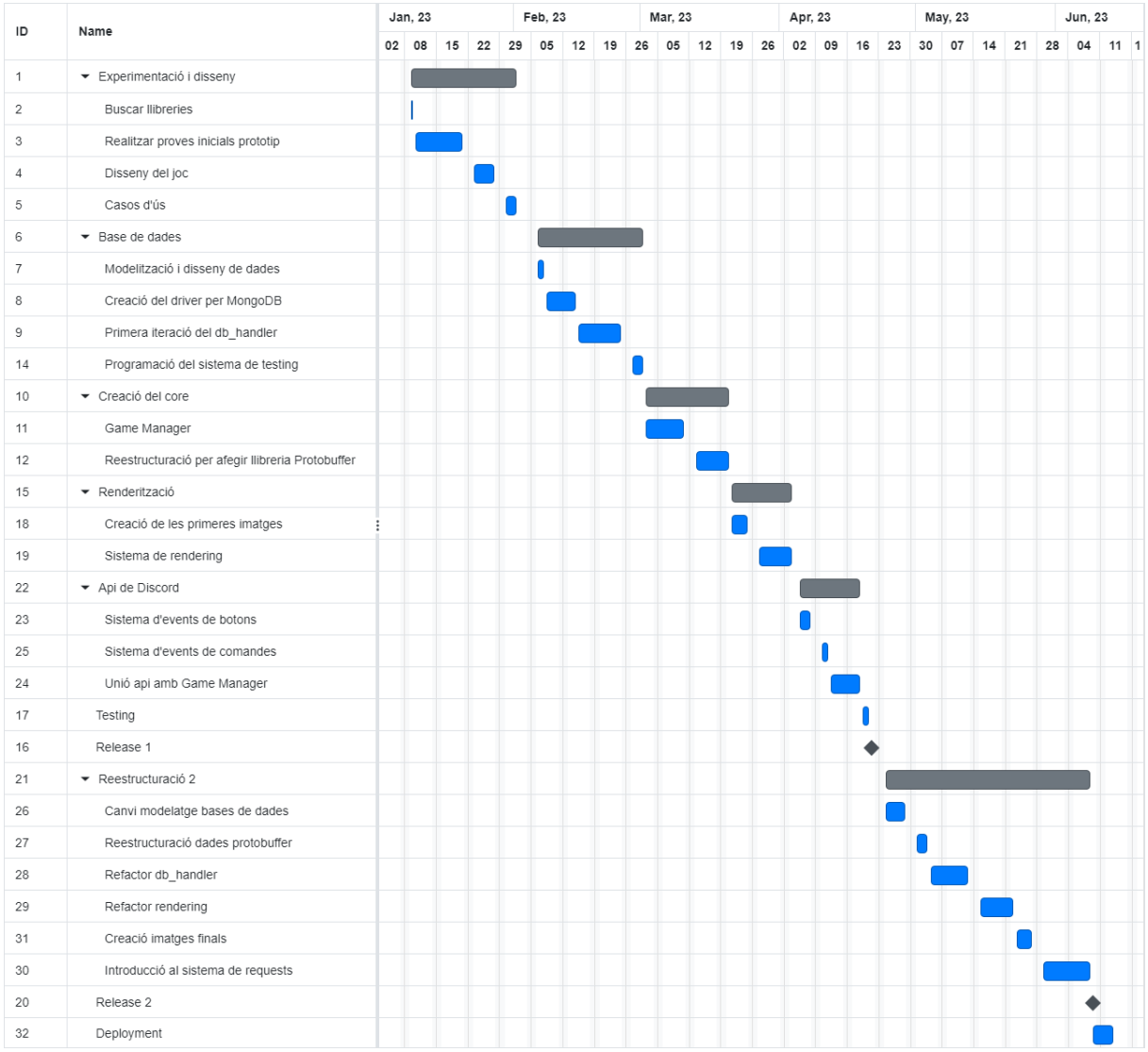
Juntament amb els requisits, és possible començar a dibuixar un primer esbós del cronograma del procés de desenvolupament. Aquest esquema està pensat a partir de l'inici del segon trimestre (9 de gener) fins a la setmana anterior de l'entrega del TFG (9 de juny). S'estima que es realitzarà de forma més o menys exacta unes 20 hores de treball a la setmana. El temps estimat de desenvolupament seria entre quatre mesos i cinc mesos contant possibles imprevistos i altres ocurrències.

El diagrama de *Gantt* inicial pren la següent forma:



Imatge 31: Gantt inicial del desenvolupament

Després de realitzar el procés de desenvolupament, s'ha efectuat un altre diagrama de Gantt amb els temps reals de desenvolupament.



Imatge 32: Diagrama de Gantt definitiu

6 Desenvolupament

L'aplicació final està conformada per un llistat de llibreries o "mòduls" que ajuden a fer que el codi estigui contingut i separat per responsabilitats. Donat que moltes tecnologies eren desconegudes al començar el procés de disseny, les decisions preses al principi han hagut de ser canviades a l'entendre més els sistemes i les seves implicacions. És per aquest motiu que l'explicació del desenvolupament es farà de forma cronològica i separada per *sprints*, donat que ha sigut el mètode utilitzat per distribuir la feina al llarg dels sis mesos de producció.

6.1 Inici i experimentació

La primera iteració del projecte tenia com a objectiu aconseguir definir les tecnologies claus per a realitzar el desenvolupament. També era essencial dissenyar el joc i així començar a entendre els possibles requisits tècnics de l'aplicació.

6.1.1 Definició del disseny de joc

La idea inicial del projecte era realitzar un simulador de granges a l'estil *Stardew Valley* però amb mecàniques similars al *Clash of Clans*. D'aquesta forma els jugadors anaven a guanyar recompenses i millorar la seva granja a partir de realitzar diferents plantacions i esperar la seva proliferació. Aquesta mecànica però comportava un problema a l'hora de com renderitzar la informació, ja que per cada planta s'havia de mostrar el seu temps de collita, el seu tipus i possibles accions a realitzar.

Com a alternativa es va decidir canviar a un sistema de generació automàtica de recursos. Els jugadors tenen al seu abast unes estructures anomenades "*posts*" que actuen com a generadors de materials. Interactuant amb elles dona com a resultat una sèrie de recompenses segons el nivell i experiència de l'usuari.

El sistema de progressió en total ha rebut poques modificacions i també està basat en gran part al Stardew Valley. Per una banda, cada jugador té una sèrie d'estadístiques i habilitats:

- Les habilitats s'obtenen a partir de la interacció amb els "*posts*" pel que donen una progressió continuada als jugadors. En total hi ha quatre habilitats: Foraging (Buscar), Mining (Minar), Combat (Combat) i Athletics (Atletisme). Si

un post necessita *foraging* per a obtenir recursos (Per exemple buscar branques en un arbust) significa que:

  - Obtenir un recurs dona experiència en *foraging*
  - El número de recursos obtinguts són directament proporcionals al teu nivell en *foraging*

- Les estadístiques s'obtenen a partir dels objectes equipats. A diferència de les habilitats la progressió aquí no és continuada ja que si un jugador es treu l'objecte equipat perd també les bonificacions corresponents. Cada habilitat també està relacionada amb una estadística, augmentant la seva puntuació de forma directament proporcional al seu nivell. En total hi ha 7 estadístiques:
  - Vida màxima
  - Força (+ Combat)
  - Defensa (+ Minar)
  - Precisió (+ Busca)
  - Velocitat (+ Atletisme)
  - Sort
  - Gold (La moneda virtual)

De la mateixa manera, cada post també té una sèrie de característiques que el jugador pot pagar per a millorar: La capacitat augmenta el recurs màxim que pot guardar a la vegada, la generació marca com de ràpid s'emplena el dipòsit i la fortuna dona una probabilitat juntament amb la sort de duplicar els recursos obtinguts.

A part de millorar els posts els jugadors també poden pagar per desbloquejar noves zones mitjançant accessos. D'aquesta forma l'exploració passa a ser un eix central en la motivació intrínseca del joc, permeten que els jugadors es sentin realitzats a l'hora de desbloquejar una part nova del mapa.

6.1.2 Definició de les tecnologies

El llenguatge de programació que es va escollir ha sigut C++ donat que s'esperava molta càrrega de computació a l'hora de generar imatges i calia ser eficient amb les respostes (Tal i com s'ha esmentat en els requisits). També el fet de poder compilar i generar un executable ha servit per a millorar encara més aquesta eficiència.

A continuació es va formalitzar un llistat de comportaments que requerien l'ús de llibreries:

- Base de Dades
- Generador d'imatges
- Descodificador de l'api de Discord

Cada un d'aquests comportaments podia ser resolt per diferents llibreries i calia contrastar possibles solucions per obtenir la més bona.

6.1.2.1 Base de dades

En un principi la base de dades que es pensava utilitzar era Cassandra ja que era open-source y permetia emmagatzemar dades en format NoSQL. No obstant, donada la seva complexitat i que no hi havia llibreries oficials en C++, es va donar el salt a MongoDB. Aquesta base de dades ofereix de primera mà drivers actualitzats en C++ (mongocxx) i permet una alta flexibilitat a l'hora de definir esquemes, característica que ha estat molt important a l'hora de desenvolupar sota el context d'una investigació.

6.1.2.2 Generador d'imatges

La llibreria GD va ser l'òptima des d'un principi ja que era molt simple d'utilitzar i tenia molt poc codi. Altres llibreries com OpenCV estan més estandarditzades però porten a sobre moltes funcionalitats que no feien falta.

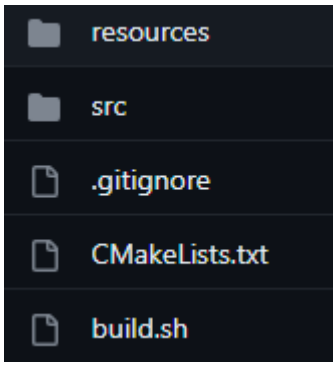
6.1.2.3 Descodificador de l'api de Discord

Finalment, la llibreria que es va decidir per interactuar i descodificar els protocols de Discord va ser D++. El seu manteniment estava al dia i dotava als seus usuaris d'una documentació excel·lent amb capacitat de respondre dubtes de forma instantània.

6.1.3 Disseny inicial

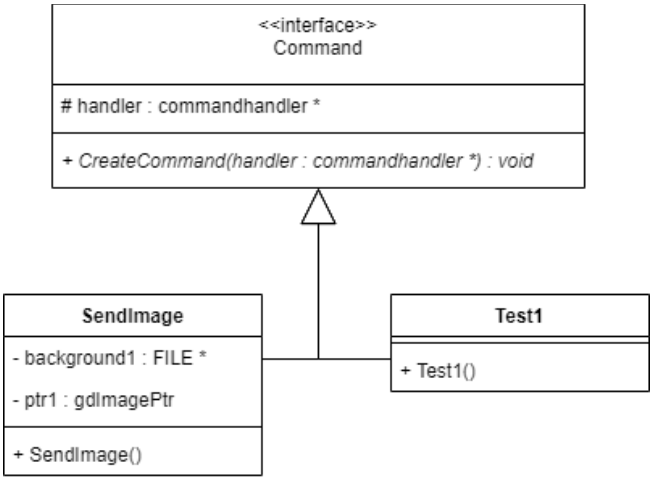
Un cop definides les llibreries que s'anaven a utilitzar, ara calia provar que realment poguessin donar l'abast a tots els requisits plantejats pel projecte.

La comprovació es va dur a terme amb la següent aplicació:



La carpeta de "*resources*" contenia les imatges que el programa utilitza per a la generació i "*src*" contenia el codi font. L'arxiu "*CMakeLists.txt*" és utilitzat per CMake per a generar els fitxers de compilació i així poder realitzar la build. "*build.sh*" és un script en llenguatge Shell que realitza de forma automàtica les comandes necessàries per realitzar la compilació.

El disseny de classes pensat per el primer prototip va ser el següent:



Imatge 33: Primer esquema UML de l'aplicació. SendImage enviava una imatge i test1 un petit text del tipus "Hello World". Tots dos implementaven la interfície Command que permetia el registre de la comanda a Discord. Elaboració pròpia.

D'aquesta forma SendImage permetia comprovar que la llibreria GD podia crear imatges de forma dinàmica i que la interfície Command funcionava bé a l'hora de garantir un número flexible d'implementadors, ja que s'esperava que hi hagués moltes comandes en acabar l'aplicació.

Dins del codi font, l'estructura d'una comanda tenia la següent forma:

```
SendImage::SendImage()
{
    background1 = fopen("resources/Background1.png","rb");
    ptr1 = gdImageCreateFromPng(background1);
}
```

El constructor de la classe s'encarregava d'obrir els punters als recursos que li feien falta. En aquest cas la classe *SendImage* necessitava obrir l'arxiu Background1.png localitzat en la carpeta de resources. *gdImageCreateFromPng* era una funció de la llibreria GD que gestiona les imatges de forma interna.

```
void SendImage::CreateCommand(dpp::commandhandler *handler) {
    this->handler = handler;
    handler->add_command
    (
        "image",
        {
            //{"name", dpp::param_info(dpp::pt_string, true, "Optional test parameter") }
        },
        [this](const std::string& command, const dpp::parameter_list_t& parameters, dpp::command_source src) {
            FILE *background2 = fopen("resources/Fruit1.png","rb");
            gdImagePtr ptr2 = gdImageCreateFromPng(background2);
            gdImagePtr baseImg = gdImageClone(ptr1);
            gdImageCopy(baseImg,ptr2,48,48,0,0,32,32);

            int size = -1;
            void *pngData = gdImagePngPtr(baseImg,&size);
            std::string s((char *) pngData,size);

            gdFree(ptr2);
            gdFree(baseImg);
            fclose(background2);

            dpp::message msg(src.channel_id, "");

            msg.add_file("img1out.png", s);
            gdFree(pngData);
            this->handler->reply(msg,src);
        },
        "Simple image"
    );
```

Imatge 34: Primera iteració de la funció que s'encarrega d'enviar una imatge a través de Discord. Elaboració pròpia.

La funció *CreateCommand* era obtinguda per herència i cada classe la implementava de forma concreta. L'objecte *commandhandler* formava part de la llibreria de D++ i gestionava totes les comandes de l'aplicació. La funció *add_command* agregava una comanda al bot definint el seu nom ("image"), components dins del missatge (en aquest cas buit), un procediment a fer quan rebi l'esdeveniment d'execució (el codi dins la funció lambda) i una descripció ("Simple Image").

El codi dins la funció lambda utilitzava la llibreria GD per a editar i superposar dues imatges:

Imatge 35: Explicació gràfica del funcionament de libGD. Elaboració pròpia.

Per a comprovar la base de dades es va realitzar un test de caràcter molt simple directament integrat en la funció *main*:

```
  mongocxx::instance inst{};
const auto uri = mongocxx::uri{"mongodb://localhost:27017"};
mongocxx::client conn{uri};
mongocxx::database db = conn["firstdb"];

auto builder = bsoncxx::builder::stream::document{};
bsoncxx::document::value doc_value = builder
    << "name" << "MongoDB3"
    << "type" << "database"
    << "count" << 1
    << "versions" << bsoncxx::builder::stream::open_array
        << "v3.2" << "v3.0" << "v2.6"
    << close_array
    << "info" << bsoncxx::builder::stream::open_document
        << "x" << 203
        << "y" << 102
    << bsoncxx::builder::stream::close_document
    << bsoncxx::builder::stream::finalize;
auto col = db["user"];
col.insert_one(doc_value.view());
```

Imatge 36: Primera iteració del procés de consulta a MongoDB. Elaboració pròpia.

En aquest exemple s'obria una connexió a la instància de MongoDB localitzada a l'ordinador local per accedir a la base de dades anomenada "firstdb". Finalment, s'afegia el document definit en la variable *doc_value* dins la col·lecció "user".

Al final del *sprint* es va poder comprovar que les llibreries satisfeien les necessitats bàsiques del projecte. Tanmateix, calia revisar l'estructura que es feia servir per interactuar amb Discord ja que la classe *commandhandler* estava documentada com

a *deprecada*. A part també era necessari començar a dissenyar l'aplicació amb un patró corresponent per garantir la separació de responsabilitats.

6.1.4 Casos d'ús

Per a determinar com el disseny i la definició de les mecàniques afectava els diferents fluxos de l'aplicació, es va decidir realitzar un estudi de casos d'ús.

Els casos s'han anat actualitzant a mesura que els requisits de l'aplicació han anat canviant. Per aquest motiu només estan escrits en aquest apartat els caos d'ús definitius corresponents a la release final del projecte.

Cas d'ús 1	Executar comanda "/start"	Actor	Jugador
Flux alternatiu 1	Si el jugador està registrat a l'aplicació, s'ha d'agafar la seva informació a la base de dades.		
Flux alternatiu 2	Si el jugador no està registrat a l'aplicació, s'ha de crear nova partida inserint la informació del jugador a la base de dades.		
Resultat	S'ha fet un print amb la imatge de la localització actual del jugador, les seves dades i les interaccions corresponents.		

Cas d'ús 2	Mostrar llista d'interaccions.	Actor	Jugador
Condicions prèvies	El jugador està registrat a l'aplicació.		
Flux normal	L'aplicació ha d'agafar la informació de l'usuari per aconseguir la localització actual. Després ha d'agafar la informació del joc per obtenir el llistat d'interaccions en aquella localització.		
Resultat	S'ha mostrat un llistat d'interaccions en la localització actual.		

Cas d'ús 3	Clicar en una interacció de la llista.	Actor	Jugador
Condicions prèvies	El jugador està registrat a l'aplicació.		
Flux normal	S'ha d'agafar la informació guardada de l'estat de la interacció dins de la base de dades. Després cal renderitzar el menú d'informació segons el tipus de la interacció.		
Resultat	S'ha imprès la informació de la interacció.		

Cas d'ús 4	Obtenir un recurs d'una interacció.	Actor	Jugador
Condicions prèvies	El jugador està registrat a l'aplicació.		

Condicions prèvies	La interacció ha de ser de tipus *post.*
Descripció flux normal	Aquest flux descriu el procediment de com es calculen les recompenses.
1	Agafar la informació del jugador a la base de dades.
2	Agafar la informació de l'estat de la interacció (el seu nivell) a la base de dades.
3	Determinar quina habilitat fa servir la interacció i quins objectes dona.
4	Calcular els segons que han passat des de l'últim cop que es va interactuar amb el post. Amb aquests segons determinar quants materials nous s'han generat. Limitar el màxim de materials a partir de la capacitat.
5	Utilitzar l'habilitat del post per determinar quants materials és capaç de recol·lectar l'usuari.
6	Calcular l'experiència obtinguda a partir dels materials recol·lectats.
7	Si hi ha sort (calculada amb la fortuna i la sort de l'usuari), duplicar els materials totals obtinguts.
8	Actualitzar les dades de l'usuari, de la interacció i de l'inventari a la base de dades.
9	Mostrar per pantalla els resultats obtinguts.
Resultat	S'ha imprès quants materials s'han obtingut.

Cas d'ús 5	Millorar una característica d'un *post.*	Actor	Jugador
Condicions prèvies	El jugador està registrat a l'aplicació.		
Flux normal	El jugador clica en la interacció i selecciona l'opció de millora. Després es tria la característica que es vol millorar i s'accepta. L'aplicació processa les dades i treu de l'inventari els objectes requerits per millorar la característica. Finalment, actualitza en la base de dades el nou nivell de la interacció.		
Flux alternatiu	Si la característica està millorada al màxim, el jugador no pot millorar-la.		
Resultat	El nou nivell de la interacció s'ha actualitzat en la base de dades.		
Error 1	El jugador no té els materials necessaris per efectuar la millora. Es cancel·la la transacció.		

Cas d'ús 6	Millorar una característica d'un *post.*	Actor	Jugador
Condicions prèvies	El jugador està registrat a l'aplicació.		
Condicions prèvies	La interacció ha de ser de tipus *post.*		
Flux normal	El jugador clica en la interacció i selecciona l'opció de millora. Després es tria la característica que es vol millorar i s'accepta. L'aplicació processa les dades i treu de l'inventari els objectes requerits per		

	millorar la característica. Finalment, actualitza en la base de dades el nou nivell de la interacció.
Flux alternatiu	Si la característica està millorada al màxim, el jugador no pot millorar-la.
Resultat	El nou nivell de la interacció s'ha actualitzat en la base de dades.
Error 1	El jugador no té els materials necessaris per efectuar la millora. Es cancel·la la transacció.

Cas d'ús 7	Desbloquejar una zona.	Actor	Jugador
Condicions prèvies	El jugador està registrat a l'aplicació.		
Condicions prèvies	La interacció és de tipus *ZoneAccess .*		
Flux normal	El jugador clica en la interacció i selecciona l'opció per desbloquejar-la. L'aplicació processa les dades i treu de l'inventari els objectes requerits per a efectuar la millora. Finalment, actualitza en la base de dades que s'ha realitzat l'operació.		
Flux alternatiu	Si la interacció ja està desbloquejada, no es pot tornar a desbloquejar.		
Resultat	S'ha desbloquejat l'accés a la nova zona.		
Error 1	El jugador no té els materials necessaris per efectuar la compra. Es cancel·la la transacció.		

Cas d'ús 8	Canviar de zona	Actor	Jugador
Condicions prèvies	El jugador està registrat a l'aplicació.		
Condicions prèvies	La interacció és de tipus *ZoneAccess.*		
Condicions prèvies	La zona està desbloquejada.		
Flux normal	L'aplicació actualitza a la base de dades la localització actual de l'usuari. Finalment mostra per pantalla la nova localització.		
Resultat	S'ha mostrat una imatge de la nova localització.		

Cas d'ús 9	Mostrar inventari	Actor	Jugador
Condicions prèvies	El jugador està registrat a l'aplicació.		
Flux normal	L'aplicació agafa les dades de l'inventari de l'usuari a la base de dades. Després mostra la informació per cada objecte en la casella corresponent. Si la longitud del vector és més gran que la capacitat màxima de la pàgina, es mostra l'opció per canviar de pàgina.		
Resultat	Es mostra per pantalla l'inventari de l'usuari.		

Cas d'ús 10	Iniciar combat amb un jugador	Actor	Jugador inicial
Condicions prèvies	El jugador inicial ha d'estar registrat a l'aplicació		
Condicions prèvies	El jugador inicial ha de tenir suficient or per a l'aposta.		
Flux normal	Es mostra un missatge amb el nom del jugador oponent		
Resultat	Es mostra pel xat una invitació a participar en el jugador oponent.		

Cas d'ús 11	Jugador oponent accepta invitació	Actor	Jugador oponent
Condicions prèvies	El jugador oponent ha d'estar registrat a l'aplicació.		
Condicions prèvies	El jugador oponent ha de tenir suficient or per acceptar l'aposta.		
Flux normal	Es crea una instància del combat a la base de dades amb la informació dels dos contrincants. Es compara l'estadística de velocitat per determinar qui comença el combat. Es realitza un parsing de tota la informació dels dos jugadors en un document llest per poder ser mostrat pel xat.		
Resultat	Es mostra en el xat l'estat del combat amb les opcions del jugador que té el torn.		

Taula 4: Casos d'ús de l'aplicació. Elaboració pròpia.

6.2 Creació de la Base de dades i testing

Aquesta fase va estar principalment dedicada al desenvolupament del pont entre la base de dades i l'aplicació del projecte. Tot i que el driver mongocxx s'ocupa de realitzar les accions principals com inserir, eliminar, actualitzar o buscar documents, és responsabilitat del programador el manteniment de la connexió i les crides a aquestes funcions.

A part era important començar a pensar una forma de comprovar la qualitat del codi i evitar canvis que puguin trencar altres funcionalitats.

6.2.1 Mòdul "database"

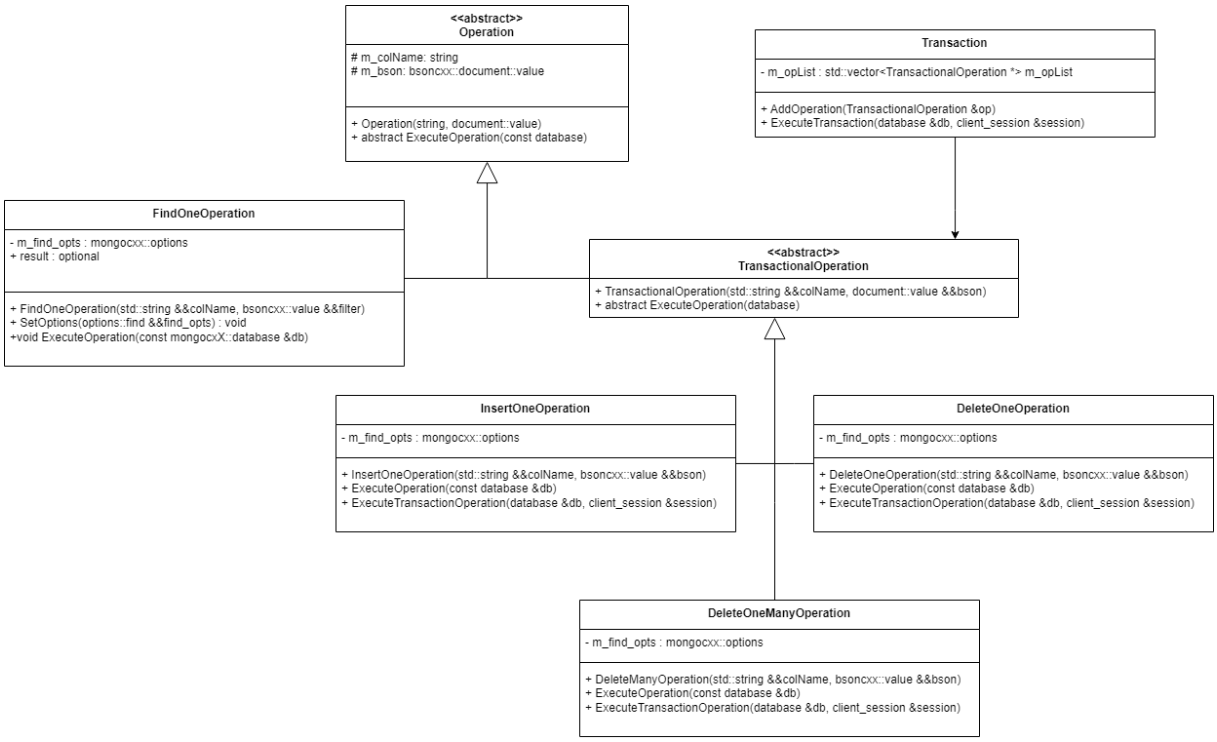
Per a poder començar la connexió amb MongoDB a partir del driver mongocxx primer de tot cal crear una instància. La forma directa de fer-ho és mitjançant el seu constructor, però en el cas de l'aplicació del treball aquesta manera està totalment prohibida ja que hi ha múltiples instàncies concurrents.

```
/* Create a pool object only once from MongoDB URI */
void createPool(std::string uri_)
{
    if (!m_client_pool)
    {
        m_client_pool = (std::unique_ptr<mongocxx::pool>)
        new mongocxx::pool { mongocxx::uri {uri_} };
    }
}

/* Acquire a client from the pool */
mongocxx::pool::entry getClientFromPool ()
{
    return m_client_pool->acquire();
}
```

Imatge 37: Sistema thread safe per aconseguir l'accés a la base de dades. Elaboració pròpia.

L'alternativa que assegura "*thread safety*" utilitza un sistema de pooling on es reserva un número fixe d'instàncies que els clients poden adquirir. Si no en queden lliures la funció *acquire* bloqueja el fil d'execució fins que s'alliberi una.

Un cop garantida la connexió calia crear una forma de poder abstreure la lògica dins el driver de mongocxx. Per fer-ho es va crear la següent estructura de classes:



Imatge 38: Disseny UML de la primera versió de la base de dades. Les accions que canvien les dades reben l'herència de TransactionalOperation. D'aquesta forma el controlador Transaction pot executar totes les operacions de forma atòmica. Elaboració pròpia.

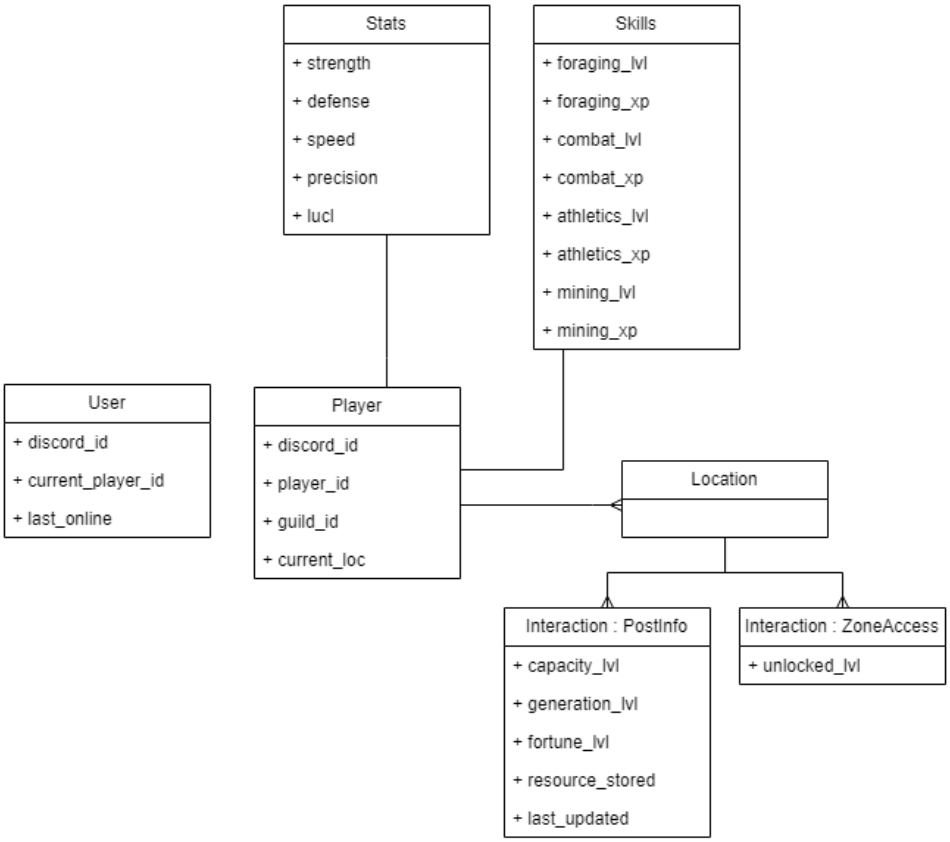
La base sota la qual va estar construïda el disseny del mòdul va ser la reducció de còpies innecessàries de variables. C++ permet ser molt específic a l'hora de gestionar la memòria i el programador pot triar entre realitzar assignacions per còpia, per moviment o per referència. Amb l'ús de (&&) al cantó d'un paràmetre s'està indicant que aquell objecte és un rvalue, activant així els constructors per moviment i no els de còpia. Un objecte que ha estat passat per moviment a un altre objecte deixa de ser vàlid.

El motiu pel qual es pretenia evitar les còpies innecessàries és degut a mongocxx. Per parlar-se amb MongoDB aquest utilitza objectes pesants de tipus BSON per a realitzar les operacions a la base de dades. En comptes d'un insert tradicional amb SQL, els

objectes o el DAO (Database Access Object) han de ser els responsables de passar tota informació a documents BSON.

6.2.2 Esquema relacional de les dades

La primera iteració de l'esquema de la base de dades intentava utilitzar al màxim l'estructura en JSON de MongoDB. Segons la documentació oficial, és important no tenir tant en compte implementacions amb Foreign Keys similars a SQL i apostar més per un disseny de documents integrats. D'aquesta forma no només s'alleugera la computació del servidor sinó que es facilita la cerca de documents.



Imatge 39: Esquema inicial de la base de dades. Elaboració pròpia.

La part important a destacar de la Imatge 39 és com es guardava l'estat actual del joc per a cada jugador. Ja que les dades de MongoDB estan inserides en format BSON i permeten ser llegides com simples objectes JSON, els vectors i objectes són també un tipus de dada amb el que es pot guardar informació. D'aquesta forma es va idealitzar un esquema on totes les localitzacions podien ser actualitzades a partir d'un índex. Amb això es complicava la forma de generar les quèries però a la vegada es

guanyava en eficiència ja que no feia falta realitzar tècniques de *joins* o cerques en col·leccions separades.

6.2.3 Mòdul *db_handler*

Per unir tota la lògica del joc amb el mòdul de base de dades, es va crear el concepte de *db_handler*. Aquest mòdul funciona similar als *DAOs* tradicionals ja que prepara i encapsula les accions que el joc necessita amb la comunicació adient a la base de dades.

```
std::unique_ptr<User> db::FindUserById(uint64_t id)
{
    auto dbClient = MongoDBInstance::GetInstance()->getClientFromPool();
    auto access = MongoDBAccess(*dbClient,DATABASE_NAME);

    auto doc = bsoncxx::builder::basic::make_document(bsoncxx::builder::basic::kvp("discord_id",0));
    FindOneOperation op = FindOneOperation("users",doc);
    access.ExecuteOperation(op);
    if (op.result)
    {
        return std::make_unique<User>(User(op.result->view()));
    }
    return nullptr;
}
```

Imatge 40: Fragment de codi responsable de buscar un usuari a la base de dades amb el mateix *id.* Elaboració pròpia.

En el codi de la Imatge 40 es pot observar com s'aplicava tot el que s'ha explicat en aquest apartat. *dbClient* s'obtenia a partir d'adquirir una instància en el servidor de la base de dades. La classe MongoDBAccess inicialitzava aquesta instància dins la base de dades corresponent. D'aquesta forma es podien separar les dades de producció amb les dades de *testing*:

```
#ifdef TEST_BUILD
    #define DATABASE_NAME "testdb"
#else
    #define DATABASE_NAME "gamedb"
#endif
```

Imatge 41: TEST_BUILD era una constant definida en el procés de compilació i permetia canviar la base de dades destí i així separar els tests de les dades de producció. Elaboració pròpia.

La variable *doc* en la Imatge 40 de tipus *document::value* és l'objecte BSON que s'utilitza per interactuar amb MongoDB.

Finalment, si la consulta tenia un valor, el db_handler s'encarregava de transformar l'objecte BSON a una entitat de l'aplicació (Una referència a un punter únic de tipus User en aquest cas).

6.2.4 Consultes

Tot i que MongoDB realitza les operacions mitjançant objectes JSON, el driver de mongocxx t'obliga a seguir les seves directrius pròpies. Per tal d'explicar com és aquest procés es mostra el següent exemple:

```
bool db_handler::GoToLocation(Player &player, g_enums::GameLocations new_location)
{
    db::UpdateOneOperation update_op{"players",
        make_document(
            kvp("discord_id",b_int64{static_cast<int64_t>(player.GetId())}),
            kvp("player_id",b_int32{player.GetPlayerId()})
        ),
        make_document(kvp("$set",make_document(
            kvp("current_loc",b_int32{static_cast<int>(new_location)})
        )))
    };

    update_op.ExecuteOperation();
    return update_op.GetState() == db::OperationState::SUCCESS;
}
```

Imatge 42: Consulta del db_handler. Elaboració pròpia.

En aquest cas s'estava actualitzant el valor de la localització actual de l'usuari per la nova localització on es volia viatjar. Les sentències update de MongoDB tenen dues parts: Quins documents es volen actualitzar i què es vol actualitzar dels documents seleccionats. El primer *make_document* contenia a dins dos camps anomenats *discord_id* i *player_id* que representaven les variables del document destí que es volien actualitzar. La funció *kvp* permet crear claus valors amb variables que venen per codi. El segon *make_document* utilitzava l'operador *"$set"* per assignar el nou valor *"new_location"* al camp *"current_loc".*

6.2.5 Entitats

Les entitats van ser la forma inicial de programar els *DAOs* per a interactuar amb la base de dades. Aquestes classes s'anaven a encarregar exclusivament de realitzar la conversió d'objectes a BSON i a la inversa.

```
Skills::Skills(bsoncxx::document::element element) :
m_forage_lvl(static_cast<uint32_t>(element["forage_lvl"].get_int32())),
m_forage_xp(static_cast<uint32_t>(element["forage_xp"].get_int32())),
m_mining_lvl(static_cast<uint32_t>(element["mining_lvl"].get_int32())),
m_mining_xp(static_cast<uint32_t>(element["mining_xp"].get_int32())),
m_combat_lvl(static_cast<uint32_t>(element["combat_lvl"].get_int32())),
m_combat_xp(static_cast<uint32_t>(element["combat_xp"].get_int32())),
m_athletics_lvl(static_cast<uint32_t>(element["athletics_lvl"].get_int32())),
m_athletics_xp(static_cast<uint32_t>(element["athletics_xp"].get_int32()))
{}
```

```
bsoncxx::document::value Skills::ToJson() const noexcept
{
    auto skills = bsoncxx::builder::basic::document{};

    skills.append(kvp("forage_lvl",b_int32{static_cast<int32_t>(m_forage_lvl)}));
    skills.append(kvp("forage_xp",b_int32{static_cast<int32_t>(m_forage_xp)}));
    skills.append(kvp("mining_lvl",b_int32{static_cast<int32_t>(m_mining_lvl)}));
    skills.append(kvp("mining_xp",b_int32{static_cast<int32_t>(m_mining_xp)}));
    skills.append(kvp("combat_lvl",b_int32{static_cast<int32_t>(m_combat_lvl)}));
    skills.append(kvp("combat_xp",b_int32{static_cast<int32_t>(m_combat_xp)}));
    skills.append(kvp("athletics_lvl",b_int32{static_cast<int32_t>(m_athletics_lvl)}));
    skills.append(kvp("athletics_xp",b_int32{static_cast<int32_t>(m_athletics_xp)}));

    return skills.extract();
}
```

Imatge 43: Exemple del DAO de "Skill". (A dalt) Conversió de BSON a Objecte. (A baix) Conversió d'Objecte a BSON. Elaboració pròpia.

Com es pot veure, la forma d'interactuar amb els objectes de tipus BSON és molt similar a la d'un mapa. A part, permet definir de quin tipus són les dades, fent que la conversió sigui molt més precisa (Sobretot quan s'està tractant amb *timestamps*).

6.2.6 Testing

Per assegurar la qualitat del codi es va veure òptima la realització d'un apartat de testing. La llibreria més famosa de C++ en aquest àmbit és Catch2. No només està ben mantinguda sinó que dota d'un control i una facilitat excel·lent per a gestionar els tests necessaris.

```
SCENARIO("Testing write to database","[db]")
{
    auto dbClient = MongoDBInstance::GetInstance()->getClientFromPool();
    auto access = MongoDBAccess(*dbClient,DATABASE_NAME);
    GIVEN("A new empty user collection database")
    {
        auto doc = bsoncxx::builder::basic::make_document();
        DeleteManyOperation op = DeleteManyOperation("users", doc);
        REQUIRE(access.ExecuteOperation(op));

        WHEN("FindingUser with an empty database")
        {
            auto doc = bsoncxx::builder::basic::make_document(bsoncxx::builder::basic::kvp("discord_id",0));
            FindOneOperation op = FindOneOperation("users",doc);
            access.ExecuteOperation(op);
            THEN("Result should be null")
            {
                REQUIRE_FALSE(op.result);
            }
        }
```

Imatge 44: Fragment de codi que realitzava el *testing* en el cas que un usuari no es trobava a la base de dades. Elaboració pròpia.

La funció *require* és similar a un *assert* on, si el resultat obtingut es fals, Catch2 atura l'execució de l'escenari i mostra en pantalla la causa de l'error.

6.3 Modularització i automatització del projecte

A mesura que el codi va evolucionant és necessari revisar com està estructurat el projecte de forma interna i qui s'encarrega de definir les dades més internes per tal d'executar tota la lògica del joc.

6.3.1 CMake

CMake permet generar sistemes de compilació automàtics i especificar *toolchains* per a cada tipologia d'executable (Com en el cas de test i producció). Amb la creació i inici del desenvolupament dels mòduls de *db* i *db_handler* el següent pas necessari era poder millorar l'estructuració del projecte ja que al compilar tants arxius a la vegada i sense un ordre definit sovint feia falta afegir o eliminar codi en el CMakeList.txt general del projecte.

Després de tot aquest treball es va passar d'un executable amb un sol procés de compilació a un conjunt de llibreries internes que s'afegien de forma dinàmica al compilar el *main.cpp*:

```
option(DEBUG_MODE "Activate debug build" OFF)
option(BUILD_TESTS "Activate test project" OFF)

if(BUILD_TESTS)
    project(dcl_test VERSION 1.0.0 DESCRIPTION "Main Bot For DiscLand")
    add_executable(${PROJECT_NAME}
        test/setup.cpp
        test/db_test.cpp
        test/db_handler_test.cpp
    )
    add_definitions(-DTEST_DATABASE)
    find_package(Catch2 3 REQUIRED)
    add_subdirectory(dcl/db)
    add_subdirectory(dcl/db_handler)
    target_link_libraries(
        ${PROJECT_NAME}
        PRIVATE Catch2::Catch2WithMain
        dcl_db
        dcl_db_handler
        pthread
    )
```

Imatge 45: CMakeList.txt general del projecte. En aquest cas es pot observar la part que gestiona la *build* de *testing.* Elaboració pròpia.

Una part important a destacar de la Imatge 45 és la funció *add_subdirectory.* Aquesta funció permet afegir mòduls de CMake i executar-los separant així la responsabilitat de cada mòdul a compilar-se per ell mateix:

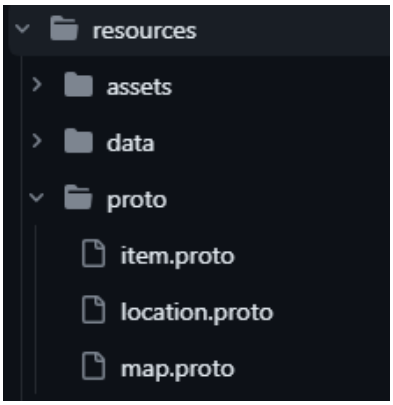
```
project(dcl_db VERSION 0.0.1 DESCRIPTION "Database Handler library for DiscLand")
include_directories("include/")
add_library(${PROJECT_NAME} SHARED "src/db_write.cpp" "src/db_query_operations.cpp")

target_include_directories(
    ${PROJECT_NAME}
    PUBLIC "/usr/local/include/mongocxx/v_noabi"
    PUBLIC "/usr/local/include/bsoncxx/v_noabi"
    PUBLIC "/include"
)

target_link_libraries(
    ${PROJECT_NAME}
    PUBLIC mongocxx
    PUBLIC bsoncxx
)
```

Imatge 46: CMakeList.txt del mòdul *db*. Elaboració pròpia.

En la Imatge 46 es pot veure com el mòdul amb el nom *dcl_db* és definit com una llibreria i s'encarrega de generar el *linking* corresponent amb el *driver* de *mongocxx*. D'aquesta forma el mòdul *db* és l'únic que està en contacte directe amb el *driver* de la base de dades.

6.3.2 DDL

Per a realitzar l'objectiu de definir tot el joc en arxius de dades JSON es va escollir la llibreria desenvolupada per Google anomenada *Protobuf*. La utilització ha sigut la següent:



Dins de recursos es va afegir una carpeta amb el nom de *data* i *proto*.

Els arxius que pertanyen a *proto* són definicions de missatges i classes que utilitzen el llenguatge estipulat per protobuf. Cada .proto genera un arxiu .cc i .h amb tots els getters i setters dels paràmetres definits.

```
message PBInteraction
{
    PBInteractionType type                  = 1;
    string interactionName                  = 2;
    PBItems resource                        = 3;
    int32 databaseId                        = 4;
    int32 posX                              = 5;
    int32 posY                              = 6;
    repeated string imagePaths              = 7;

    PBLocationID nextLoc                    = 8;
    repeated PBUpgrades upgradeInfo         = 9;
}
```

Imatge 47: Definició del missatge *PBInteraction* dins de *map.proto.* Elaboració pròpia.

PBInteraction, mostrada a la Imatge 47, és un dels missatges més importants i que es veu adient analitzar alguns dels seus camps: **type** defineix el tipus de la interacció. En el moment de creació només es podia escollir entre post (generació de recursos) o *ZoneAccess* (canviar de zona). La **posX** i la **poxY** defineixen la posició de la interacció dins de la imatge generada amb la icona referenciada per **imagePaths**. Finalment, les dades **nextLoc** i **upgradeInfo** són valors opcionals. La primera defineix, en cas de ser de tipus *ZoneAccess*, cap quina localització porta la interacció. La segona defineix tots els requisits necessaris per a millorar de nivell una estadística i el seu valor actual. Per exemple, al llegir els continguts de **upgradeInfo** per la generació de nivell u, es pot obtenir que es crea una unitat nova cada deu segons i que per ser millorada requereix deu pals i cinc pedres.

Imatge 48: Dades en format json de la interacció. (Esquerra) Primera iteració sense la llibreria Protobuf (Dreta) Segona iteració utilitzant el sistema de la llibreria Protobuf. Elaboració pròpia.

Amb tot aquest sistema en marxa només calia generar el codi que pogués llegir i construir el món de joc a partir d'aquestes dades.

6.4 *Core*

El mòdul *core* conté tota la lògica interna de l'aplicació més els *protobuffers* generats de forma automàtica. D'aquesta forma el mòdul actua com a capa de negoci unint les interaccions que genera la capa d'API amb la base de dades de MongoDB.

6.4.1 DCLMap

La classe DCLMap és un Singleton que s'encarrega de descodificar i crear tot el joc a partir dels arxius JSON i la llibreria protobuffer. La informació dins d'aquesta classe és constant i només permet la lectura externa. Aquesta filosofia ajuda a no haver de preocupar-se de possibles problemes de concurrència.

```
    DCLMap()
    {
        for (const auto & entry : std::filesystem::directory_iterator(map_data_location))
        {
            std::ifstream t(entry.path());
            std::stringstream buffer;
            buffer << t.rdbuf();
            auto loc_data = PBLocation{};
            google::protobuf::util::JsonStringToMessage(buffer.str(),&loc_data);
            PBLocationID loc_id = loc_data.locid();
            std::cout << "LocID: " + std::to_string(loc_id) << std::endl;
            std::cout << entry.path() << std::endl;
            m_locations.emplace(loc_id,DCLLocation{std::move(loc_data)});
        }
    }

public:
    static DCLMap& getInstance()
    {
        static DCLMap instance;
        return instance;
```

Imatge 49: Creació del *Singleton DCLMap* i lectura de les dades. Elaboració pròpia.

A l'inicialitzar-se DCLMap, per cada arxiu json dins la carpeta de *data* genera una classe fent servir el missatge PBLocation. Aquest objecte és després col·locat dins d'un mapa on la clau és la id de la localització i el valor l'objecte en si.

L'objecte PBLocation és embolcallat per una classe interna del projecte i així poder realitzar comprovacions i altres lògiques de l'aplicació. També d'aquesta forma s'assegura que el codi que hagi de treballar sobre aquestes dades sigui incapaç de realitzar cap mena de canvi:

```
const std::string &GetInteractionsImage(int id, int level) const
{
    const PBInteraction &target_int = m_loc_data.interactions(id);
    int max_lvl = target_int.imagepaths_size();
    if (level >= max_lvl) return target_int.imagepaths(max_lvl - 1);
    return target_int.imagepaths(level);
};
```

Imatge 50: Funció constant que retorna el path a una imatge de la interacció definit en el JSON. Elaboració pròpia.

```
std::optional<bool> GetZoneAccessUnlocked(int id,int build_level = 0) const
{
    auto interaction = m_loc_data.interactions(id);
    if (interaction.type() != PBInteractionType::ZONE_ACCESS) {return std::nullopt;}
    return interaction.upgradeinfo(0).info(build_level).currentstat() == 1;
}
```

Imatge 51: Funció que valida si l'accés a la zona està o no desbloquejat a partir del *build_level.* Elaboració pròpia.

6.4.2 Game Manager

Finalment, dins del mòdul de *core* també estava localitzat el game manager, el controlador de totes les accions del joc. Aquesta classe utilitzava tant el DCLMap com les funcions dins de db_handler per executar i realitzar totes les comprovacions pertinents de l'aplicació.

```
gm::Errors gm::GoToZone(uint64_t discord_id, int32_t interaction)
{
    auto user = db_handler::FindUserById(discord_id);
    if (!user) return Errors::USER_NOT_FOUND;

    int result = db_handler::CurrentPlayerLocation(discord_id,user->GetCurrentPlayer());
    if (!PBLocationID_IsValid(result)) return Errors::GENERAL_ERROR;
    auto location = GameMap::DCLMap::getInstance().GetLocation(static_cast<PBLocationID>(result));

    auto interaction_type = location->GetInteractionType(interaction);
    if (!interaction_type.has_value()) return Errors::INTERACTION_NOT_FOUND;
    if (interaction_type.value() != PBInteractionType::ZONE_ACCESS) return Errors::ILLEGAL_ACTION;
    auto databaseID = location->GetInteractionDatabaseID(interaction);
    if (databaseID == -1)
    {
        PBLocationID next_loc = location->GetZoneAccessNextLoc(interaction).value();
        auto next_loc_info = GameMap::DCLMap::getInstance().GetLocation(static_cast<PBLocationID>(next_loc));
        if (!next_loc_info) return Errors::ILLEGAL_ACTION;
        if (next_loc_info->GetDatabaseID() != -1 && db_handler::PlayerFirstTimeToLocation(discord_id,user->GetCurrentPlayer(),next_loc))
```

Imatge 52: Fragment dins del Game Manager que canvia la zona de l'usuari si la interacció és de tipus ZoneAccess i està desbloquejada. Elaboració pròpia.

El codi de la Imatge 52 forma part de la capa més externa dins de la pròpia capa de negoci. Les funcions que es troben en aquest arxiu seran executades per la llibreria que s'encarregui de gestionar els esdeveniments llençats per Discord. El paràmetre *interaction* és l'id de la interacció dins d'una localització específica. Cada localització té una llista d'interaccions amb les que l'usuari pot interactuar. El codi verifica que la interacció seleccionada efectivament és de tipus *ZoneAcces* per a poder comprovar el seu estat en la base de dades.

Si la id de base de dades era -1 indicava que la interacció no calia ser guardada. En el cas dels ZoneAccess això implica que aquella zona estava oberta per defecte i no calia realitzar cap més comprovació.

En el cas que la id no fos -1, era necessari primer buscar l'estat actual guardat a MongoDB per a realitzar la comprovació de si la zona estava desbloquejada o no.

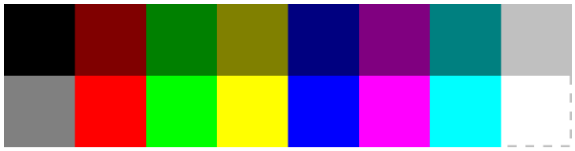
Finalment, si la zona estava desbloquejada i el jugador podia passar a la següent localització, calia realitzar una sèrie de comprovacions extres per assegurar-se que la zona estava creada de forma correcta a la base de dades. En cas que la localització fos nova, l'aplicació havia de generar un esquema buit de dades a inserir en la posició correcta del vector. Per exemple, si la localització dos tenia tres interaccions de les quals només una estava marcada amb un id de -1, les altres dos havien de guardar-se a la base de dades en la posició dos del vector de localitzacions.

6.5 Renderització

El mòdul de renderització ha de complir amb un dels objectius principals del treball: La generació d'imatges de forma dinàmica a partir de dades variables. És d'esperar que no es podia començar a construir el sistema sense tenir prèviament muntat tota la lectura de dades tant del joc com dels usuaris registrats a l'aplicació.

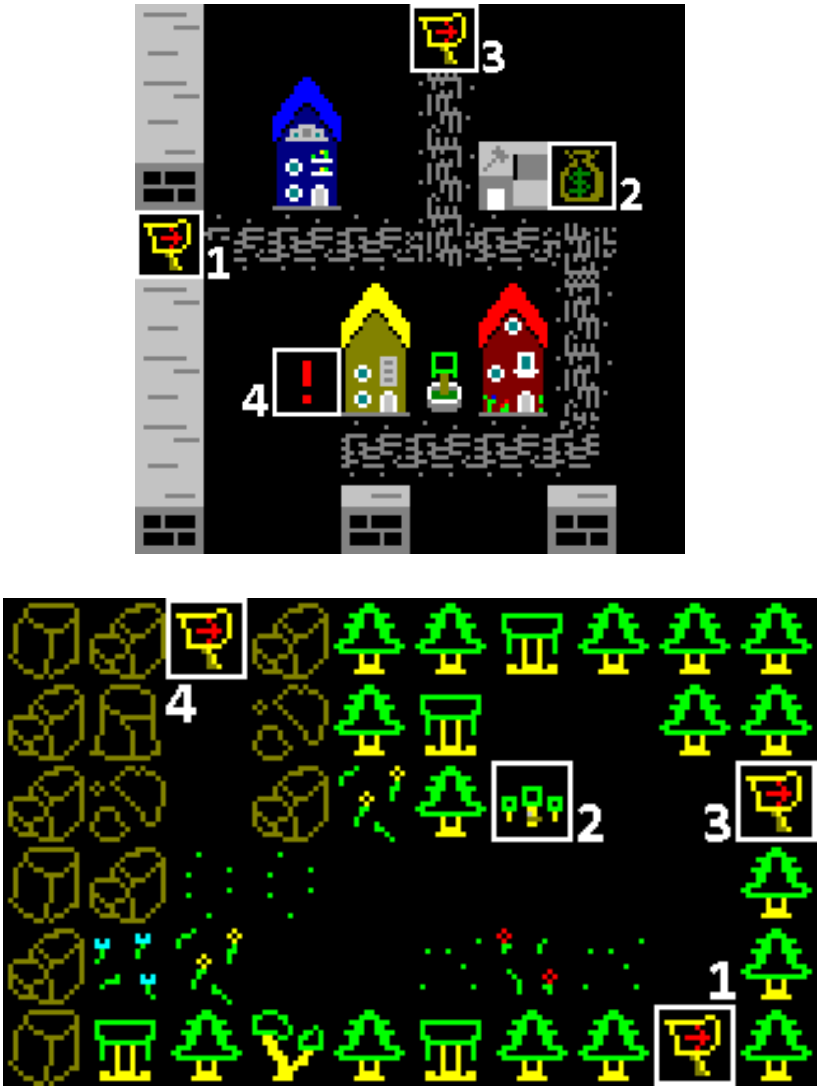
6.5.1 Primeres imatges

Seguint la temàtica i els gràfics dels primers MUDs, es va decidir recrear un estil de píxel art amb la paleta de colors de Windows16. A part, no es podien crear imatges amb molta resolució ja que la seva grandària anava a afectar de forma proporcional les necessitats de computació.



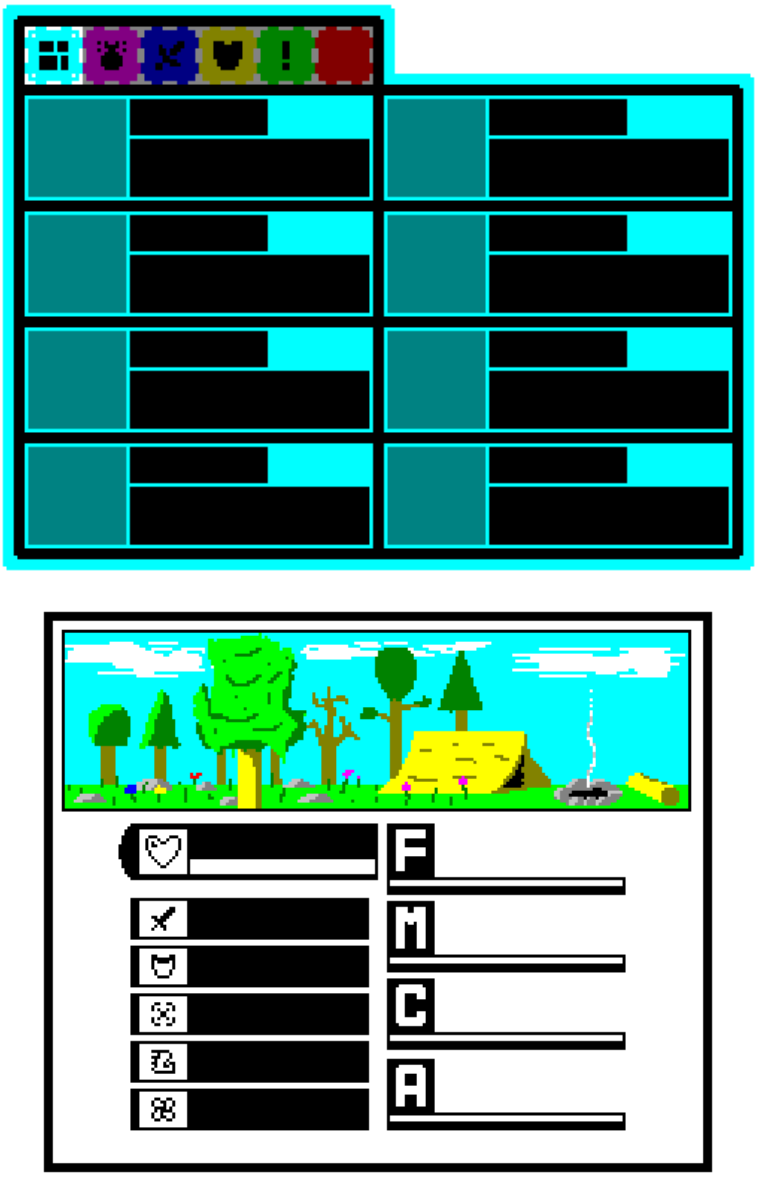
Imatge 53: Paleta de colors de Windows16. Imatge de *Wikimedia Commons.*

A part de l'estil també era important determinar quina informació essencial s'havia de mostrar en les imatges de les localitzacions. D'alguna forma s'havia d'indicar a l'usuari quines eren les estructures amb les quals podia interaccionar més alguna mena d'identificador per a poder seleccionar alguna d'específica. El resultat va ser el següent:



Imatge 54: Primera iteració de la localització del poble (a dalt) i de l'entrada a la mina (a baix). Elaboració pròpia.

A part de les localitzacions també calia dissenyar les interfícies on representar la informació de l'usuari. Les primeres propostes van ser:



Imatge 55: (A dalt) Primer inventari de l'aplicació. (A baix) Visualitzador d'estadístiques. Elaboració pròpia.

6.5.2 Aplicació de la llibreria GD

GD està programat amb el llenguatge C, pel que moltes funcions de l'API demanen que s'utilitzin tipus de dades del llenguatge com FILE *. Tot i que no és un problema com a tal, treballar a baix nivell i sortir de la seguretat amb la que treballen els

programes amb C++ no és del tot desitjat (Per exemple, en el cas dels FILE *, C++ treballa amb objectes de tipus fstreams que permeten encapsular tota la lògica).

Per sort, la pròpia llibreria GD té al seu abast una sèrie d'arxius que embolcallen tota la lògica en funcions i objectes de C++ amb el nom de GDPP. Tot i que aquests arxius s'han de compilar de forma manual dins de l'aplicació, això permet modificar algunes de les funcions en cas que es vegi oportú.

```
void AddImageText(GD::TrueColor &color, GD::Point &p, int size, std::string &text, bool bold) {
    std::string font;
    if (bold)
    {
        font = "resources/bold.ttf";
    } else {
        font = "resources/regulat.ttf";
    }
    StringFT(nullptr,color.Int(),(char*)font.c_str(),size,0,p,text);
}
```

Imatge 56: Funció creada dins de la llibreria GDPP per afegir texts a les imatges mitjançant freetype2. Elaboració pròpia.

Amb tot això configurat, es poden començar a generar imatges de la següent forma:

```
Renderer::LocationRender::LocationRender(const std::string &locationPath)
{
    std::ifstream in{locationPath, std::ios::binary};
    image = GD::Image{in};
    std::cout << image.good() << std::endl;
}

void Renderer::LocationRender::AddInteraction(int posX, int posY, const std::string &interaction_image)
{
    std::ifstream in{interaction_image, std::ios::binary};
    GD::Image interaction{in};
    std::cout << interaction.good() << std::endl;
    GD::Size s;
    interaction.GetSize(s);
    std::cout << s.H() << std::endl;
    std::cout << s.W() << std::endl;
    GD::Point p{posX,posY};

    image.Copy(interaction,p,GD::Point{0,0},s);
}
```

Imatge 57: Codi per a visualitzar una localització. Elaboració pròpia.

La llibreria treballa molt amb objectes de tipus *GD::Image*. Aquests objectes demanen guardar memòria al *heap* i obren un punter a l'espai reservat de forma automàtica. Un cop inicialitzat de forma correcta, es pot modificar el contingut de la imatge fent servir el conjunt de funcions de l'objecte més altres contenidors auxiliars de tipus GD::Point o GD::Size.

Finalment, cal centrar-se en un problema que pot afectar seriosament l'estabilitat de l'aplicació. Si no s'haguessin aplicat mesures extres al codi de la Imatge 57 s'estaria produint un *memory leak* ja que no s'estava alliberant en cap moment la memòria reservada de la imatge. Per solucionar aquest error es va fer servir una característica del mateix C++11: Els punters amb destructors personalitzables.

```
std::unique_ptr<char, void(*)(char*)> Renderer::LocationRender::RenderLocation(int *size)
{
    return std::unique_ptr<char, void(*)(char*)>{(char *) image.Png(size), Renderer::FreePointer};
}
```

```
void FreePointer(char * data)
{
    std::cout << "I'm freeing your pointer" << std::endl;
    gdFree((void *)data);
}
```

Imatge 58: (A dalt) Funció que assigna un punter únic amb destructor personalitzat anomenat Renderer::FreePointer. (A baix) la funció *Renderer::FreePointer.* Elaboració pròpia.

Els *unique_ptr* serveixen per crear una forma de punters intel·ligents. A diferència dels tradicionals, aquests punters alliberen la memòria de forma automàtica quan detecten que no hi ha ningú que els estigui referenciant. A més, si de segon paràmetre afegeixes un punter a una funció, aquest actuarà com a destructor personalitzat. La definició *std::unique_ptr<char, void(*)(char*)>* està indicant que el punter intel·ligent apunta a una variable de tipus *char* i que té un destructor en forma de punter a una funció amb un paràmetre *char**** que no retorna res (*void*). Com es pot veure, la funció *FreePointer* compleix amb aquesta descripció.

6.6 Mòdul d'API

Amb tota la lògica finalment desenvolupada el punt final per a la primera release era realitzar la comunicació amb Discord. Per fer-ho es va decidir en una primera instància utilitzar un sistema basat en comandes. El procediment va ser el següent:

```
bot.on_ready([&bot,&bootstrap,delete_commands](const dpp::ready_t& event) {
    if (delete_commands)
    {
        auto commands = bot.global_commands_get_sync();
        for (auto & command : commands)
        {
            bot.global_command_delete_sync(command.first);
        }

        bootstrap.RegisterCommands();
    }
```

Imatge 59: Registre de les comandes a Discord. Elaboració pròpia.

Primer de tot cal registrar quines seran les comandes i quina serà la seva estructura. Per fer-ho Discord té una funció lambda que és llençada quan el bot està finalment operatiu. Tot aquest procediment d'enregistrament de comandes (i esborrat) utilitza l'API construïda sota el protocol HTTP, pel que és important no abusar de les peticions.

La classe de bootstrap va ser creada amb l'objectiu de reemplaçar el *commandhandler* que estava marcat com a funcionalitat deprecada:

```
CommandBootstrap::CommandBootstrap(dpp::cluster *bot)
{
    this->bot = bot;
    command_list.insert({"start",new StartCommand(bot->me.id)});
    command_list.insert({"photo",new PhotoCommand(bot->me.id)});
    command_list.insert({"inventory",new InventoryCommand(bot->me.id)});
    command_list.insert({"unlockzone",new UnlockZoneCommand(bot->me.id)});
    command_list.insert({"goto",new GotoCommand(bot->me.id)});
    command_list.insert({"collect",new CollectCommand(bot->me.id)});
    command_list.insert({"improve",new ImproveCommand(bot->me.id)});
    command_list.insert({"stats",new StatsCommand(bot->me.id)});
    command_list.insert({"postinfo",new PostInfoCommand(bot->me.id)});
}
```

Imatge 60: Constructor del bootstrap, inicialitza totes les classes. Elaboració pròpia.

Principalment *CommandBootstrap* s'encarregava de mantenir un mapa desordenat on les claus eren identificadors de la comanda i els valors punters a les diverses implementacions de la interfície *Command*. Cal recordar que C++ no permet declarar a la pila variables que són interfícies o classes abstractes, pel que és obligat en aquest cas utilitzar la paraula reservada *new*.

```
class InventoryCommand : public Command
{
public:
    const std::string COMMAND_NAME = "inventory";
    InventoryCommand(dpp::snowflake appid)
    {
        isGlobal = true;
        command = dpp::slashcommand(COMMAND_NAME,"Print inventory",appid);
        command.add_option(
            dpp::command_option(dpp::co_string, "inventory_page", "The page of the inventory you want to look", true).
                add_choice(dpp::command_option_choice("Resources", std::string("resources"))).
                add_choice(dpp::command_option_choice("Armor", std::string("armor"))).
                add_choice(dpp::command_option_choice("Quest Items", std::string("quest_items"))).
                add_choice(dpp::command_option_choice("Consumables", std::string("utilities")))
        );

        command.add_option(
            dpp::command_option(dpp::co_integer, "page_number", "The number of the page you want to look at", false)
        );
    }
```

Imatge 61: Definició de la comanda per obrir l'inventari. Elaboració pròpia.

Les classes que implementaven la interfície *Command* després definien en el seu constructor com volien ser executades i quina era la informació necessària per a funcionar. En el cas concret de la Imatge 61, s'està definint com accedir a les diferents pàgines de l'inventari a través del mètode *add_option*.

Amb totes aquestes classes creades dins del mapa de *CommandBootsrap,* només feia falta cridar el mètode RegisterCommands que, per cada entrada, creava una petició a Discord per afegir la comanda a l'aplicació online.

```
void CommandBootstrap::RegisterCommands()
{
    std::cout << "Registering commands!" << std::endl;
    for(std::map<std::string, Command *>::iterator it = command_list.begin(); it != command_list.end(); ++it)
    {
        Command* current = it->second;
        if (current->isGlobal)
        {
            bot->global_command_create(current->command);
        }
    }
}
```

Imatge 62: Registre de totes les comandes a l'aplicació dins de Discord. Elaboració pròpia.

```
bot.on_slashcommand([&bot,&bootstrap](const dpp::slashcommand_t & event) {
    Command *command = bootstrap.Find(event.command.get_command_name());
    command->HandleCommand(&event);
});
```

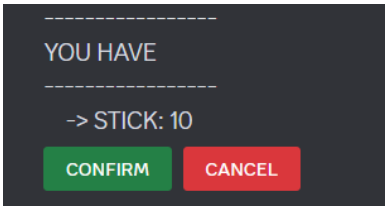
Imatge 63: Funció lambda que és executada quan un usuari utilitza una comanda. Elaboració pròpia.

```
void HandleCommand(const dpp::slashcommand_t *event)
{
    int size = -1;
    dpp::user target = event->command.get_issuing_user();
    auto image_data = gm::PhotoCurrentLocation(target.id, &size);
    dpp::message m("");
    std::string s(image_data.get(),size);
    m.add_file("img1.png",s);
    m.set_flags(dpp::m_ephemeral);
    event->reply(m);
}
```

Imatge 64: Implementació de la funció HandleCommand que defineix què passa quan s'executa la comanda. Elaboració pròpia.

Per a poder executar les comandes, el funcionament era bastant similar. Al rebre una comanda d'algun usuari, l'aplicació executava la funció lambda *on_slashcommand*. Aquesta funció trobava en el mapa la comanda adient i executava el seu mètode concret *HandleCommand*. En el cas de Imatge 64, la funció cridava al *Game Manager* per obtenir la imatge corresponent de la localització i imprimir-la en pantalla.

El flux d'execució dels botons era una mica diferent però utilitzava una funcionalitat molt potent anomenada identificadors personalitzables. Cada botó permetia assignar una cadena de caràcters com identificador que podia ser llegit per l'aplicació un cop interceptat l'esdeveniment de *on_button_click*. Ja que l'identificador permetia una longitud màxima de 100 caràcters, es podia utilitzar l'espai restant per guardar altra informació important com per exemple el "propietari" del botó. Al conèixer aquesta tècnica, es va idear un protocol intern per a poder separar les dades i llegir variables de la següent forma: *"<identificador del botó>::<variable_1>::<..>::<variable_n>"*.



Imatge 65: Exemple d'uns botons amb identificadors personalitzats a l'hora de millorar un *post*. Elaboració pròpia.

Tot i ser invisible pels jugadors, els botons de la Imatge 65 contenen informació de qui és l'usuari que està realitzant l'acció i per quina millora ho està fent. Per exemple, el contingut de l'identificador pel botó de confirmació queda de la següent manera: "confirm_button::@user_id::@upgrade_id".

```
bot.on_button_click([&bot,&button_bootstrap](const dpp::button_click_t & event) {
    std::vector<std::string> button_commands;
    size_t pos = 0;
    std::string token;
    std::string custom_id_mutable = event.custom_id;
    std::string delimiter = "::";
    while ((pos = custom_id_mutable.find(delimiter)) != std::string::npos) {
        token = custom_id_mutable.substr(0, pos);
        button_commands.push_back(token);
        custom_id_mutable.erase(0, pos + delimiter.length());
    }
    button_commands.push_back(custom_id_mutable);
    auto button = button_bootstrap.Find(button_commands.at(0));
    button->HandleButton(event, button_commands);
});
```

Imatge 66: Parsing del identificador personalitzable (custom id) un cop ha sigut clicat. Elaboració pròpia.

6.7 Primera release de Discland

La primera release va ser el resultat de tota la investigació realitzada i tenia l'objectiu de veure tots els mòduls en funcionament per analitzar possibles millores.

6.7.1 Canvis importants

Tot i haver explicat cada un dels mòduls del projecte en punts anteriors (db, db_handler, core, api i rendering), aquesta release comportava canvis significatius originats pel creixement orgànic de l'aplicació. És per això que abans de passar als primers resultats es veu pertinent anomenar algunes millores destacables que van rebre aquests mòduls.

6.7.1.1Items amb protobuffers

Per a definir els noms dels diferents objectes es va implementar un sistema similar al parsing de localitzacions. Aquesta funcionalitat extra també es va afegir en el constructor estàtic de DCLMap fent que tot el joc es carregués a la vegada.

Imatge 67: Estructura de les dades del protobuffer d'ítems. Elaboració pròpia.

6.7.1.2 *Refactor* mòdul *db*

La necessitat de mantenir els requisits de la base de dades en un sol lloc va propiciar aquest refactor. Amb el canvi, la responsabilitat d'adquirir els *locks* va a passar a formar part de les pròpies funcions d'accés a la MongoDB. D'aquesta forma es va poder esborrar de forma definitiva la classe DBAccess. Finalment, es va afegir un *try*

catch per donar una capa de seguretat addicional a l'operació. Amb això el propi objecte *Operation* és l'encarregat de definir el seu estat a través de l'enumeració *OperationState*.

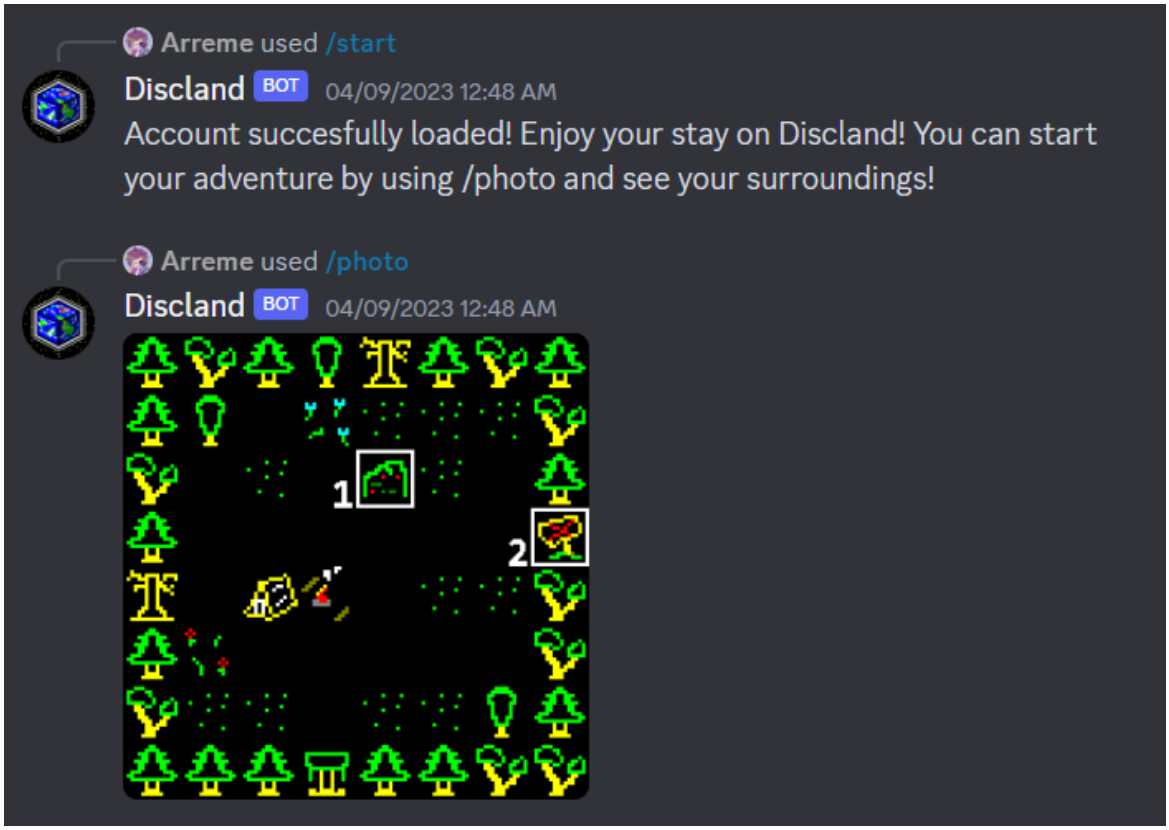
```
void FindManyOperation::ExecuteOperation() noexcept
{
    try
    {
        auto client = MongoDBInstance::GetInstance()->getClientFromPool();
        auto db = (*client)[DATABASE_NAME];

        if (m_has_options)
        {
            m_result = db[m_colName].find(m_bson.view(), m_find_opts);
        } else
        {
            m_result = db[m_colName].find(m_bson.view());
        }
        m_db_state = OperationState::SUCCESS;
    }
    catch(std::exception e)
    {
        m_db_state = OperationState::GENERAL_ERROR;
    }
}
```

Imatge 68: Nou mètode per accedir a la base de dades. Elaboració pròpia.

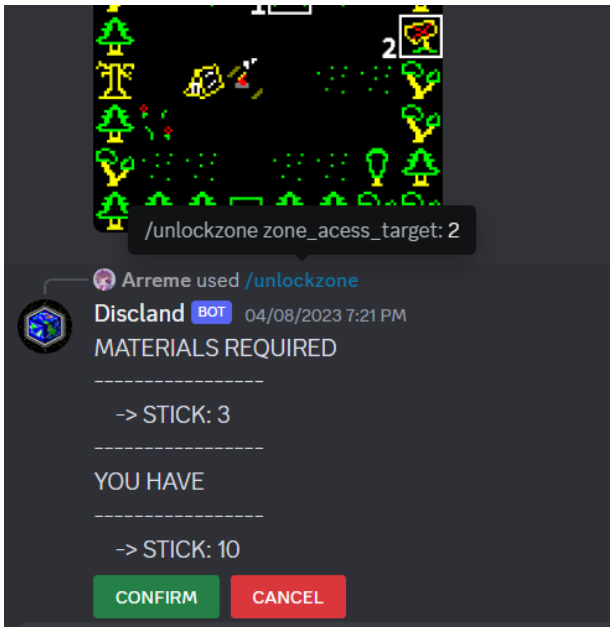
6.7.2 Imatges de l'aplicació

- Capacitat de desbloquejar i d'explorar diverses zones del mapa.
- Capacitat per recollir recursos dels posts.
- Comunicació basada principalment en comandes.
- Renderització de l'inventari amb els diferents objectes de l'inventari.
- Mostra d'estadístiques del jugador.
- Sistema online que permetia la connexió de múltiples partides a la vegada.
- Sistema de compra de millores i desbloquejables.

A continuació es mostren algunes captures de com estava l'aplicació en el moment de llençar la primera release. En els annexos es pot comprovar també el codi amb el que es va acabar de fer l'entrega.



Imatge 69: Registre i mostra de la localització actual. Elaboració pròpia.



Imatge 70: Desbloquejar una zona. Es pot observar com el cartell en la interacció "2" mostra que està bloquejat. Elaboració pròpia.

Imatge 71: Sistema de inventari amb imatges i número d'objectes guardats. Elaboració pròpia.

6.7.3 Problemes

L'origen de la motivació per a realitzar una segona release va néixer al fer-se aparents diversos problemes d'arquitectura i de disseny que calien ser millorats. A continuació s'exposen les més importants a destacar:

6.7.3.1 Disseny de la base de dades

Un problema que es va fer evident al començar a programar les diverses funcions del Game Manager va ser l'estructura de la base de dades inicial. Al posar els usuaris i jugadors en col·leccions separades era pràcticament impossible no haver de realitzar mínim dues consultes per a cada interacció: Una per saber el jugador actual de l'usuari i una segona per obtenir la informació del jugador en si. També la decisió de realitzar consultes petites però específiques agreujava encara més aquest problema:

```
auto user = db_handler::FindUserById(discord_id);
if (!user) return Errors::USER_NOT_FOUND;

int result = db_handler::CurrentPlayerLocation(discord_id,user->GetCurrentPlayer());
if (!PBLocationID_IsValid(result)) return Errors::GENERAL_ERROR;
auto location = GameMap::DCLMap::getInstance().GetLocation(static_cast<PBLocationID>(result));

auto interaction_type = location->GetInteractionType(interaction);
if (!interaction_type.has_value()) return Errors::INTERACTION_NOT_FOUND;
if (interaction_type.value() != PBInteractionType::ZONE_ACCESS) return Errors::ILLEGAL_ACTION;

auto interaction_db = location->GetInteractionDatabaseID(interaction);
if (!interaction_db.has_value()) return Errors::INTERACTION_ALREADY_UNLOCKED;
auto interaction_info = db_handler::FindPlayerCurrentInteraction(discord_id,user.value().GetCurrentPlayer(
if (!interaction_info.has_value()) return Errors::DATABASE_CONNECTION_ERROR;
auto zone_access = static_cast<ZoneAccessInfo * const>(interaction_info->second.get());
```

Imatge 72: Per a obtenir la lògica del joc s'han de realitzar tres consultes consecutives a la base de dades. Elaboració pròpia.

6.7.3.2 Doble definició d'entitats

Possiblement un dels problemes més importants. La separació en la definició d'entitats feia que moltes de les classes que utilitzava el db_handler no es poguessin comunicar de forma directa amb les classes generades per protobuffers.

```
Item::Item(bsoncxx::document::view bson)
{
    m_item_id = bson["item_id"].get_int32();
    m_quantity = bson["quantity"].get_int32();
}
```

Imatge 73: Entitat item definida al db_handler. Elaboració pròpia.

```
auto get_requirements = location->GetZoneAccessLevelRequirements(interaction,zone_access->GetUnlockedLvl());
std::vector<Item> item_requirements;
item_requirements.reserve(get_requirements.size());
for (auto &req : get_requirements)
{
    item_requirements.emplace_back(req.itemid(),req.quantity());
}

auto items = db_handler::GetItems(discord_id,user.value().GetCurrentPlayer(),Item::RESOURCE_TYPE , item_requirements);
```

Imatge 74: Necessitat de convertir els PBItems a Items del db_handler en un for per a que es pugui realitzar la crida a la base de dades. Elaboració pròpia.

6.7.3.3 Mala implementació del Game Manager

La falta de capacitat de previsió a l'hora de realitzar una de les parts més importants va comportar una problemàtica en el disseny de l'arquitectura. Es va començar definint mètodes globals ja que funcionaven bé per a realitzar tasques de testing i eren fàcils d'implementar. No es va començar a veure les possibles mancances de disseny fins a les etapes finals de la release, ja que les funcions sovint tenien molta lògica repetida ignorant per complet el principi DRY (Don't Repeat Yourself). A part, el sistema de retorn d'errors afegia molta complexitat al codi, ja que les funcions de l'API havien de crear diferents casos d'ús per a tots els possibles valors de l'enumeració.

```
Errors UnlockZone(uint64_t discord_id, int32_t interaction);

Errors CollectPost(uint64_t discord_id, int32_t interaction, std::string &output);

Errors CanImprove(uint64_t discord_id, int32_t interaction, std::string upgrade_name, std::string &out);

Errors ImprovePost(uint64_t discord_id, int32_t interaction, std::string upgrade_name);



std::unique_ptr<char, void(*)(char*)> PhotoCurrentLocation(uint64_t discord_id, int *size);
```

Imatge 75: Captura del fitxer game_manager.hpp. Elaboració pròpia.

Una altra problemàtica d'aquest Game Manager ve influenciada pel punt 6.7.3.4 de la release. Ja que la comunicació amb els usuaris es feia principalment mitjançant comandes, els errors potencials que podien cometre eren molt elevats. És per aquest motiu que en la Imatge 72 es poden observar tantes clàusules condicionals.

6.7.3.4 Interfície i experiència d'usuari

La primera release va apostar per una comunicació principalment via comandes. Això comportava molts problemes de cara l'experiència d'usuari: Primer de tot, els jugadors havien de recordar un gran número d'accions diferents i la seva sintaxi corresponent. Tot i que Discord ofereix una ajuda per a poder escriure les comandes de forma visual, la quantitat d'informació a recordar no ho feia un sistema viable. Seguidament, el fet d'haver de relacionar un número del mapa amb la interacció de la comanda a realitzar era un sistema molt poc intuïtiu. A més a més, s'havia de dedicar un espai bastant

important en el mapa a escriure els números, pel que també era un limitador a l'hora de dissenyar el món de joc. Finalment i lligat al punt anterior, no hi havia cap comprovació externa per si l'usuari utilitzava una comanda incompatible amb una interacció: Utilitzar un /collect a un *ZoneAccess*, per exemple. Ja que l'API no podia realitzar aquestes comprovacions, tota la responsabilitat havia de ser passada al Game Manager dins del mòdul de *core*.

6.8 Segona release de Discland (Finalització)

La release final ha sigut el resultat de totes les millores aplicades a la primera release. No només s'ha realitzat una anàlisi profunda dels problemes sinó que també s'ha reformat pràcticament tot el codi des de zero. En aquesta part és on s'ha notat sobretot l'esforç dedicat a l'estructuració del projecte, ja que es va poder separar tot el procés per mòduls.

6.8.1 Canvis importants:

Per a explicar correctament el codi i les seves millores, s'ha separat l'explicació pels diferents mòduls. L'única part que es va decidir mantenir per complet va ser el paquet de *db* ja que es va comprovar la seva robustesa en la primera release.

6.8.1.1 Db_handler

S'ha eliminat per complet les entitats dins del db_handler. Ara, tot està definit en els protobuffers. En aquesta release les classes dins d'aquest mòdul actuen més com un DAO ja que embolcallen l'objecte guardant una referència per punter i actualitzen el seu contingut amb les diverses crides a la base de dades. D'aquesta forma el manteniment de l'estat i la seva *ownership* està centralitzada al responsable de cridar les funcions de db_handler.

```
class DBLocationHandler
{
private:
    PBLocation *m_location;
    static const bsoncxx::document::value s_start_post;
    static const bsoncxx::document::value s_start_zone_access;
    static const bsoncxx::document::value s_start_dialog;
    static const std::unordered_map<std::string,void (db_handler::DBLocationHandler::*)(PBInteraction *,bsoncxx::docume

    bsoncxx::document::value PostToBson(const PBPostInteraction &post_info) const;

    void BsonToPost(PBInteraction *interaction, bsoncxx::document::view doc);

    bsoncxx::document::value ZoneAccessToBson(const PBZoneAccessInteraction &zone_access_info) const;
```

Imatge 76: Fragment de la classe DBLocationHandler. Elaboració pròpia.

A part, s'ha reestructurat l'esquema de la base de dades per a que anés lligat a la definició de missatges establert pels protobuffers. Amb aquest nou disseny els

sistemes es poden comunicar amb completa harmonia i no fa falta realitzar cap mena de conversió.

6.8.1.2 Core

El principal factor que ha propiciat tots els canvis de la segona release han estat els protobuffers. Amb el nou disseny, les estructures han passat a ser més genèriques i ara tenen l'habilitat de tenir diferents tipus d'interacció a la vegada. També s'han preparat les dades per introduir-ne de nous com és el cas del "Dialog Interaction", que permet explicar amb més detall les estructures o parlar amb NPCs.

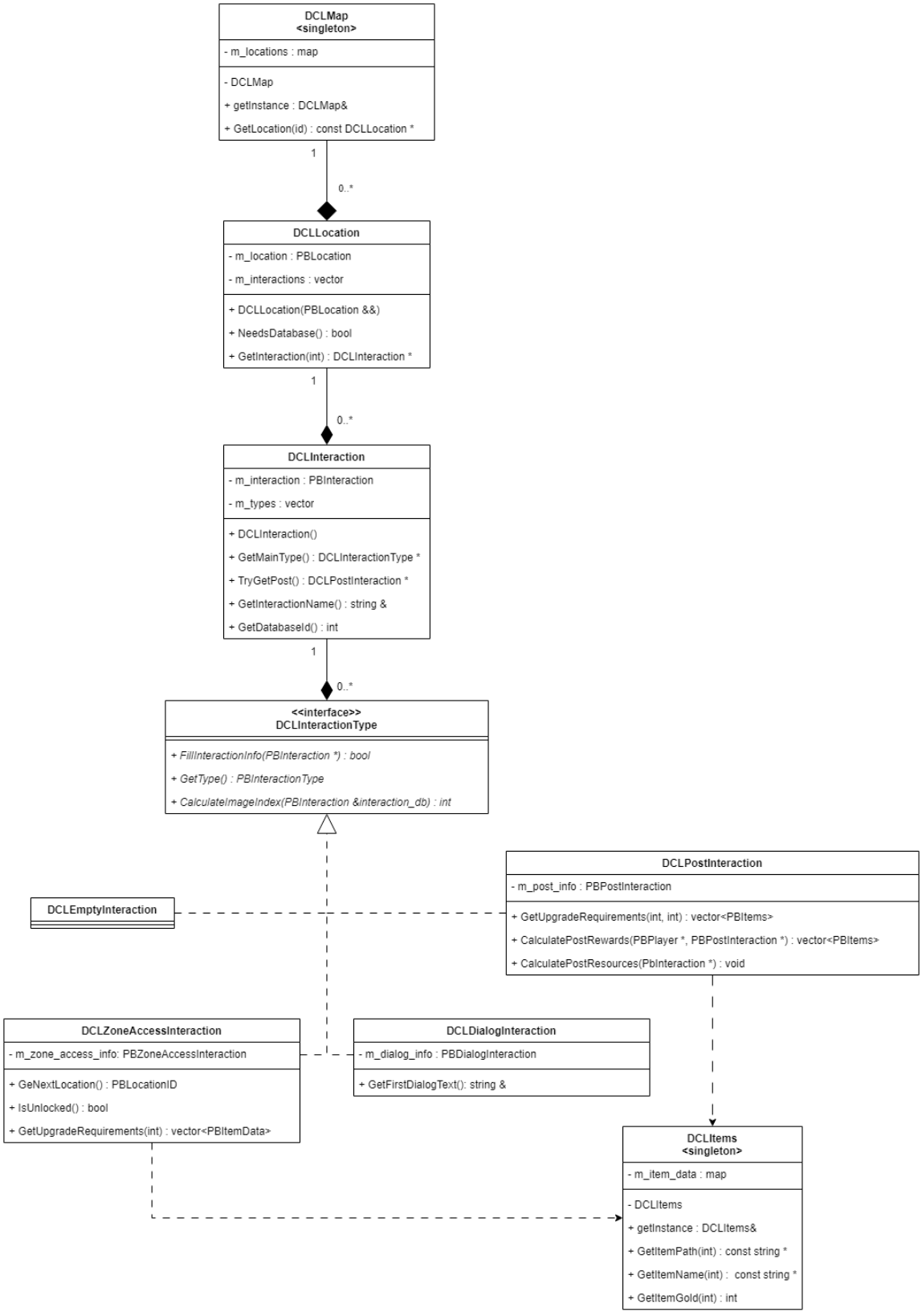
El nou sistema funciona de la següent forma: En comptes de generar missatges diferents per a cada tipus d'interacció, les estructures tenen sempre un missatge genèric que permet l'addició de "mòduls". L'agent principal que defineix de quin tipus és la interacció és el camp "*types*", que ara ha passat a ser una llista. A més, també ha nascut el concepte de "*main type*" que defineix com la interacció demana ser visualitzada, corresponent a l'índex zero de l'array. D'aquesta forma, definint al camp *types* la llista de tipus que es vol assignar a una interacció el codi ja pot configurar tota la informació pertinent.

```
message PBInteraction
{
    repeated PBInteractionType types              = 1;
    int32 database_id                             = 2;
    string interaction_list_name                  = 3;
    string interaction_name                       = 4;
    string description                            = 5;
    int32 pos_x                                   = 6;
    int32 pos_y                                   = 7;
    repeated string map_icon_paths                = 8;

    PBPostInteraction post_info                   = 9;
    PBZoneAccessInteraction zone_access_info      = 10;
```

```
message PBZoneAccessInteraction
{
    PBLocationID next_loc                         = 1;
    bool needs_database                           = 2;
    repeated PBUpgradeInfo unlock_info            = 3;

    //database
    int32 unlock_level                            = 4;
}
```

Imatge 77: Nous esquemes dels protobuffers. Elaboració pròpia.

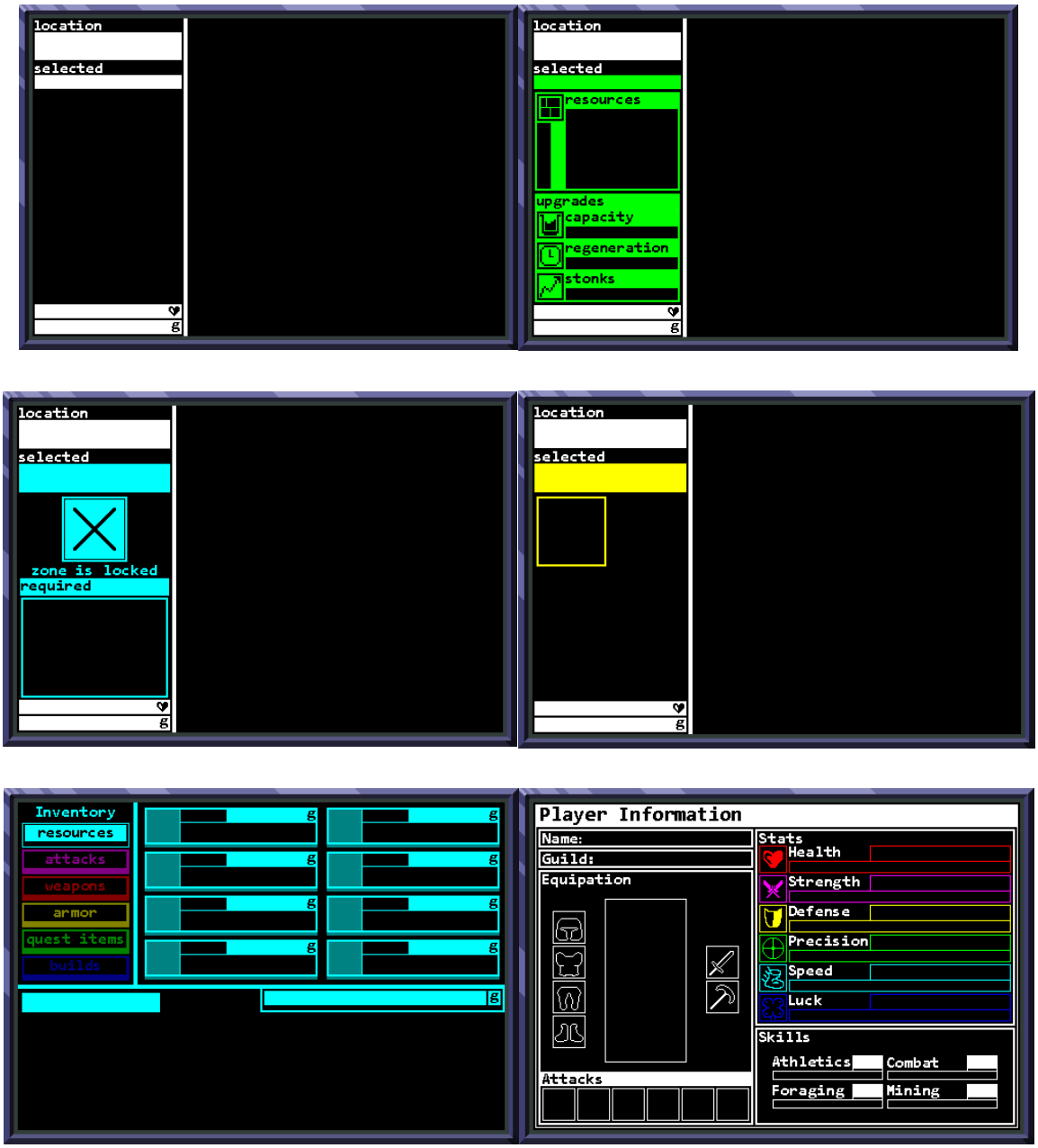
També s'ha canviat de forma dràstica el funcionament dels contenidors d'informació com DCLMap. S'ha separat d'una forma més consistent les responsabilitats de tal manera que cada classe (DCLMap, DCLLocation, DCLItem, DCLInteraction, ...) sap com gestionar de forma interna l'objecte el qual està embolcallant. A continuació es mostra el nou esquema UML:



Imatge 78: Nou UML del core de Discland. Elaboració pròpia.

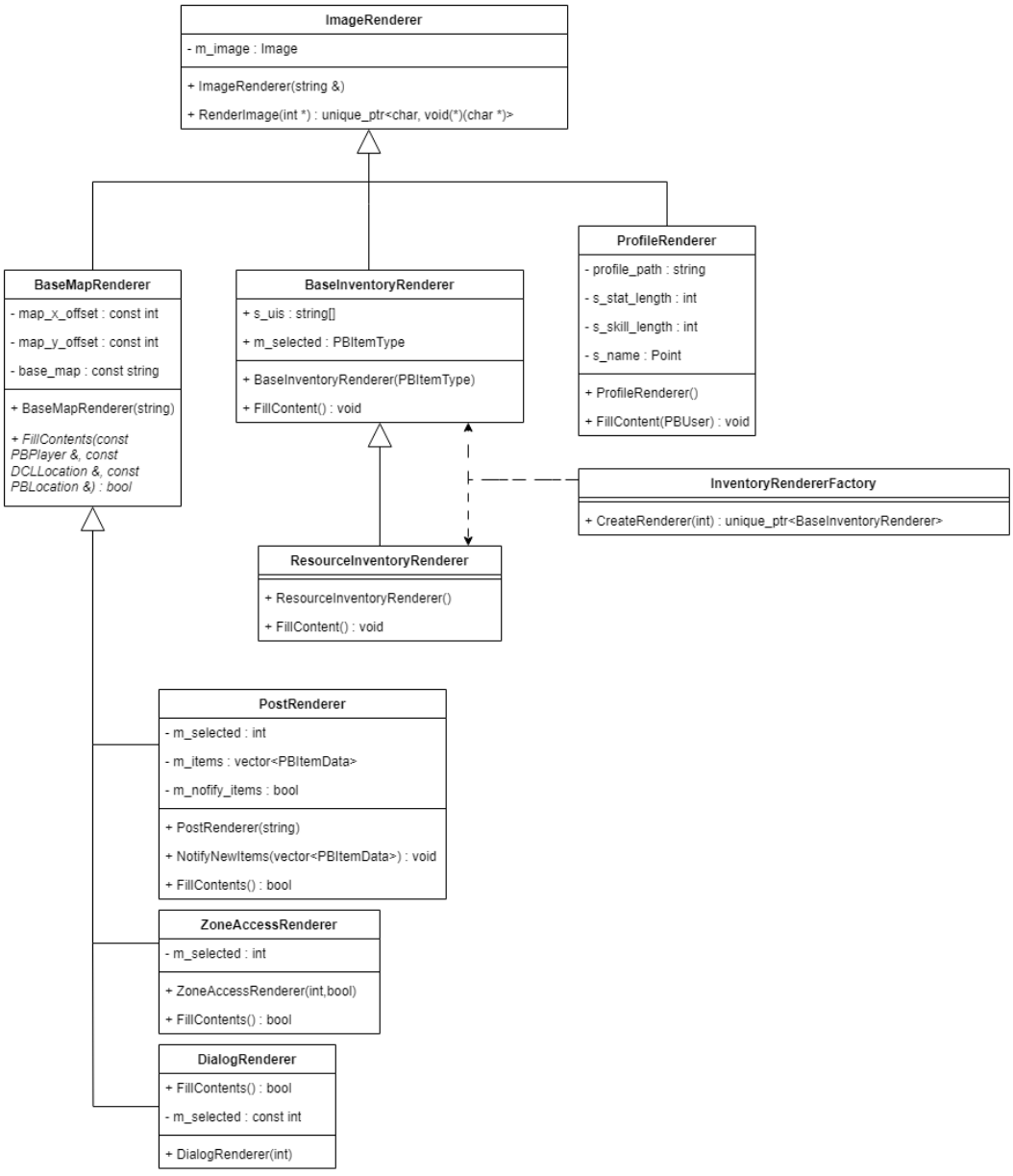
6.8.1.3 Img

Amb els canvis del mòdul core també s'ha vist necessari un replantejament dels visuals i el sistema de renderització. Les imatges, en comptes de centrar-se únicament a ensenyar el mapa o l'inventari, ara intenten mantenir una interfície homogènia tan en forma com en concepte. Així s'aconsegueix una millora en la immersió del jugador. També s'ha incrementat la grandària de les imatges en píxels permetent així poder dibuixar més detalls a part de tenir més marge per a dibuixar lletres i icones.



Imatge 79: Interfícies que el bot utilitza per emplenar les dades. Elaboració pròpia.

El sistema de renderització també ha rebut molts canvis per tal de simplificar encara més la visualització. S'han agrupat les variables fixes i de classe per tal d'abstreure al màxim la posició especifica dels texts o de les imatges. D'aquesta forma al crear un renderer tota aquesta informació és comuna entre les classes i no depèn dins de cap funció (així s'elimina el concepte de "*magic numbers*"). A més a més, s'han especialitzat els diferents agents encarregats de generar les imatges per tal d'aconseguir una API més extensible i clara:



Imatge 80: UML del mòdul de rendering. Elaboració pròpia.

6.8.1.4 API

Amb la necessitat de millorar l'experiència de l'usuari, moltes comandes instaurades en la primera release han sigut esborrades o canviades per botons i llistes. D'aquesta forma les accions que abans s'havien de comprovar a causa de possibles incorreccions de l'usuari ara estan protegides, ja que si una acció no es pot realitzar només cal amagar el botó corresponent. Un exemple bastant clar surt amb el tema d'interactuar amb una estructura. Abans hi havia la possibilitat que un usuari intentés, per exemple, recollir una recompensa d'una estructura marcada de tipus *ZoneAccess*. Ara, si una estructura no té el tipus *Post* el botó "*COLLECT*" no es renderitza.

També s'han implementat les llistes que reemplacen els números identificadors d'estructures per un simple desplegable amb totes les opcions a interactuar de la zona.

Finalment, la classe game manager que residia en el mòdul de core s'ha esborrat per complet i s'ha apostat per un sistema de Requests implementat dins del mòdul d'Api. El motiu del canvi és que aquest mòdul és l'únic que està enllaçat amb la llibreria de D++, pel que només ell és capaç d'editar els missatges enviats a l'usuari. Amb la implementació de les requests, la pròpia classe és capaç de modelar aquests missatges en funció dels errors, pel que ja no cal retornar res per avisar sobre aquests esdeveniments.

```
gm::Errors gm::CanUnlock(uint64_t discord_id, int32_t interaction, std::string &output)
{
    auto user = db_handler::FindUserById(discord_id);
    if (!user) return Errors::USER_NOT_FOUND;

    int result = db_handler::CurrentPlayerLocation(discord_id,user->GetCurrentPlayer());
    if (!PBLocationID_IsValid(result)) return Errors::GENERAL_ERROR;
    auto location = GameMap::DCLMap::getInstance().GetLocation(static_cast<PBLocationID>(result));
```

Imatge 81: Mètode que permet desbloquejar una zona. Elaboració pròpia.

```
bool PrintMapRequest::FillRequest(dpp::message &m)
{
    if (!m_data.m_user_created) return false;
    m.set_flags(dpp::m_ephemeral);

    const DCLData::DCLLocation *location_data = DCLData::DCLMap::getInstan
    if (!location_data)
    {
        m.set_content("Error! Please use command /repair to fix");
        return false;
    }
```

Imatge 82: Comparativa entre el sistema d'errors de la primera release (A dalt) amb el de la release final (a baix). Elaboració pròpia.

6.8.1.5 Combat

El combat permet la interacció de dos usuaris a l'estil de jocs JPRG. S'ha implementat un sistema molt senzill on els jugadors a partir de les seves estadístiques poden elegir fins a un màxim de tres accions al començar el seu torn:

- Atacar: Baixar la vida al contrincant a partir del teu valor a "*strength*"
- Bloquejar: Bloquejar el mal rebut del contrincant a partir del teu valor de "*defense*"
- Esquivar: Permet esquivar tot el mal del contrincant a partir d'un sistema de probabilitats lligat a l'estadística de velocitat. Si és la mateixa pels dos jugadors, la probabilitat és del 50%

A l'acabar, el jugador que guanya el combat rep els diners que s'han apostat del seu contrincant.

```
int damage = 1 + your_turn->stats().strength() + your_turn->skills().combat_lvl();
std::cout << damage << std::endl;
switch (not_your_turn_action)
{
case PBCombatActions::CA_DODGE:
    {
        int speed_dif = (not_your_turn->stats().speed() + not_your_turn->skills().foraging_lvl()) - (your_turn->stats().speed() + your_turn->skills().foraging_lvl());
        speed_dif = std::min(std::max(speed_dif,-10),10) + 10;
        std::random_device rd;
        std::mt19937 gen(rd());
        std::uniform_int_distribution<> distr(0, 20);
        if (speed_dif <= distr(gen))
            not_your_turn->mutable_stats()->set_current_health(not_your_turn->stats().current_health() - damage);
    }
    break;
case PBCombatActions::CA_BLOCK:
    damage = std::max(0,damage - (not_your_turn->stats().defense() + not_your_turn->skills().mining_lvl()));
    not_your_turn->mutable_stats()->set_current_health(not_your_turn->stats().current_health() - damage);
    break;
default:
    not_your_turn->mutable_stats()->set_current_health(not_your_turn->stats().current_health() - damage);
    break;
}
```

Imatge 83: Lògica que s'encarrega d'implementar els requisits del combat. Elaboració pròpia.

6.8.2 Deployment

Per a realitzar el deployment cal primer de tot obtenir un servidor que pugui estar encès durant tot el dia i que permeti connexió a Internet. Ja que no es disposa de cap dispositiu local s'ha de buscar una alternativa al núvol. Per això, s'han comparat diferents opcions com Heroku, Gitlab, Clouding.io i Azure.

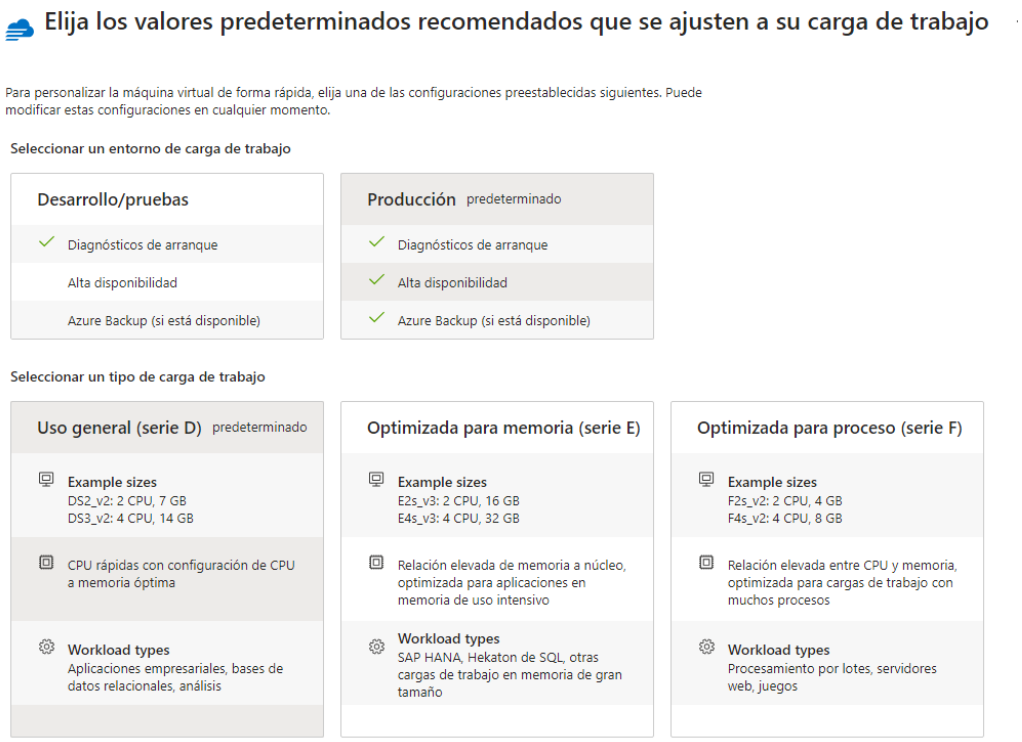
De les quatre opcions, Azure és l'opció que s'ha escollit ja que permet tenir obert un servidor durant més temps de forma gratuïta que els altres i amb unes característiques més potents.

6.8.2.1 Azure

S'ha creat un grup de recursos anomenat "tfg" amb el que es relacionaran totes les entitats que s'hagin de necessitar de cara a la realització de deployment. De tots els serveis que ofereix Microsoft en el seu portal, només es farà servir un recurs de tipus "Màquina Virtual" amb un sistema operatiu Linux que pugui mantenir-se encès de forma consistent.

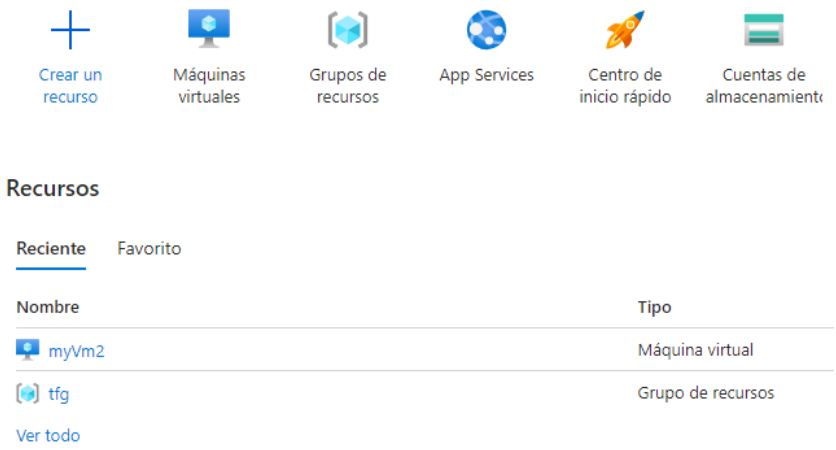
El procés de creació pot ser una mica avançat ja que és important escollir bé les característiques de l'ordinador. De fet, s'han tingut problemes amb aquest tema a causa d'un primer intent amb una màquina massa poc potent. Gràcies a l'ajuda del

sistema de recomanació de Microsoft s'ha pogut elegir una màquina adequada pel treball:



Imatge 84: Sistema de creació d'una màquina virtual per recomanació. Per realitzar el treball s'ha elegit una màquina de desenvolupament d'ús general (Sèrie D). Elaboració pròpia.

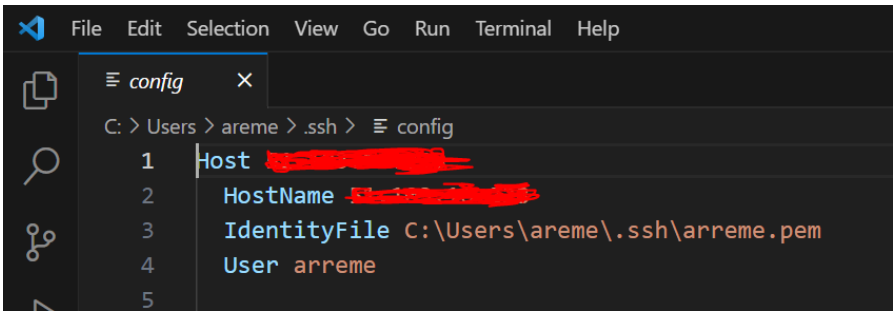
Després de realitzar l'elecció bàsica i de configurar l'accés via SSH a la màquina virtual, comença el procés de creació dels recursos dins d'Azure. Amb això el portal on es llisten totes les entitats queda de la següent forma:



Imatge 85: Captura del portal Azure. Elaboració pròpia.

6.8.2.2 Connexió i preparació de l'entorn

Un cop activada la màquina virtual es pot realitzar la connexió mitjançant el protocol SSH amb la ip pública del servidor ubuntu. Aquest protocol s'assegura que només l'usuari amb l'arxiu d'identitat correcte pugui registrar-se a l'ordinador en remot. Per raons de seguretat s'ha censurat aquesta ip en les fotografies d'exemple.



Imatge 86: Configuració del SSH. Elaboració pròpia.

Després de tenir accés directe a la màquina virtual, ja es poden començar a realitzar els diferents processos de preparació de l'entorn.

6.8.2.2.1 Llibreries bàsiques

A partir de la comanda sudo apt-get install s'han de descarregar les següents llibreries:

- build-essential: Compiladors i altres eines importants per a realitzar el desenvolupament.
- zlib1g: Compressor d'arxius.
- libpng-dev: Eines per editar imatges png.

- libfreetype-dev: Generació per codi de text a imatges.
- libss-dev: Implementació del protocol OpenSSL i TLS per Ubuntu.
- protobuf-compiler: Per poder compilar executables amb protobufers.
- cmake: Gestor de toolchains per C++.
- mongodb: Servei de base de dades.

6.8.2.2.2 Instal·lació MongoDB:

Primer de tot cal crear el replica set adient per així tenir accés a les transaccions. Per fer-ho s'ha seguit la guia oficial de MongoDB:

- Anar a l'arxiu mongod.service i editar el procés d'execució per a que rebi els següents arguments: "--replSet rs0" i "--bind_ip localhost".
- Activar el servei de mongodb i obrir una connexió via mongoshell.
- Executar "rs.initiate()".
- Fer restart de la instància amb "sudo sysemctl start mongod.service".

6.8.2.2.3 Compilació de la llibreria Mongoc

La llibreria mongoc és la base per a la llibreria de mongocxx així que s'ha de construir abans de passar a la següent fase.

- Clonar via github el repositori adient de mongoc
- Crear una carpeta de build dins del repositori
- Executar la comanda cmake DENABLE_AUTOMATIC_INIT_AND_CLEANUP = OFF ..
- Realitzar la build: cmake --build .
- Fer la instal·lació dels arxius de la build: sudo cmake --build . --target install

6.8.2.2.4 Compilació de la llibreria mongocxx a partir de la mongoc

- "cmake .. -DCMAKE_BUILD_TYPE=Release DCMAKE_INSTALL_PREFIX= /usr/local -DCMAKE_CXX_STANDARD=20" Permet configurar la build de forma que sigui apta per a la release i que utilitzi el estàndard C++20. A part configurar que la instal·lació es faci a la carpeta /usr/local
- Realitzar la build amb "cmake --build ."
- Instal·lació de la build i dels arxius: "sudo cmake --build . --target install"

6.8.2.2.5 Instal·lació libgd

Cal configurar en el CMake les llibreries externes que es voldran utilitzar. En el cas del treball només caldrà marcar amb un "1" les opcions "ENABLE_PNG", "ENABLE_GD_FORMATS" i "ENABLE_CPP".

```
OPTION(ENABLE_GD_FORMATS "Enable GD image formats" 1)
OPTION(ENABLE_PNG "Enable PNG support" 1)
OPTION(ENABLE_LIQ "Enable libimagequant support" 0)
OPTION(ENABLE_JPEG "Enable JPEG support" 0)
OPTION(ENABLE_TIFF "Enable TIFF support" 0)
OPTION(ENABLE_ICONV "Enable ICONV support" 0)
OPTION(ENABLE_XPM "Enable XPM support" 0)
OPTION(ENABLE_FREETYPE "Enable Freetype2 support" 0)
OPTION(ENABLE_FONTCONFIG "Enable FontConfig support" 0)
OPTION(ENABLE_WEBP "Enable WebP support" 0)
OPTION(ENABLE_HEIF "Enable HEIF support" 0)
OPTION(ENABLE_AVIF "Enable AVIF support" 0)
OPTION(ENABLE_RAQM "Enable RAQM support" 0)
OPTION(ENABLE_ASAN "Enable (gcc) ASAN support" 0)
OPTION(ENABLE_CPP "Enable CPP GD API" 1)
OPTION(VERBOSE_MAKEFILE "Enable CMAKE_VERBOSE_MAKEFILE" 0)
```

Imatge 87: Cmake principal de la llibreria libGD. Elaboració pròpia.

Amb el principal CMake actualitzat a les necessitats del projecte, es pot realitzar el procés de configuració amb: "cmake -DBUILD_TEST=0 ..". Finalment, la compilació es pot executar amb "make" i la instal·lació amb "make install".

6.8.2.2.6 Retocs finals:

Actualitzar els *paths* del compilador per a que agafi els nous arxius .so necessaris per enllaçar la build del joc amb la comanda "sudo ldconfig". Finalment, donar permisos d'execució a l'arxiu "build.sh" dins de la carpeta del projecte amb "sudo chmod 777 build.sh".

Amb tots aquests retocs al Linux d'Azure i executant l'arxiu compilat del projecte, el deployment està finalment complet.

6.8.3 Resultats

Per determinar l'èxit del treball realitzat, a continuació hi ha una explicació punt per punt de com s'ha completat cada objectiu del projecte:

Principals:

- **Creació d'un videojoc fent servir l'API que proporciona Discord:** El videojoc ha estat creat i permet una interacció directa amb l'API de Discord tant per xat com per botons.
- **Generar imatges de forma dinàmica:** Es realitzen imatges diferents per a cada usuari com és el cas de l'inventari o del mapa.
- **Creació d'una base de dades:** S'ha creat un procés d'integració amb la base de dades MongoDB que s'utilitza per guardar tota aquella informació dels jugadors via codi.
- **Lectura i modelització d'un sistema DDL (Data Definition Language):** S'utilitzen els Protobufers per carregar les dades del joc de forma dinàmica de forma que el dissenyador pot efectuar els canvis que vulgui sense tocar el codi.
- **Disseny del joc:** S'han creat tota una sèrie de sistemes que donen vida al joc (Sistema de recompensa, estadístiques, exploració, ...).
- **Art funcional:** El joc està ple d'imatges que s'utilitzen de forma freqüent com a base per a generar entorns personalitzats per a cada usuari.
- **Arquitectura que garanteix flexibilitat:** S'ha realitzat tota una fase de replantejament d'arquitectura que ha donat fruït a una estructura molt flexible i que permet crear noves funcionalitats sense gaire esforç.

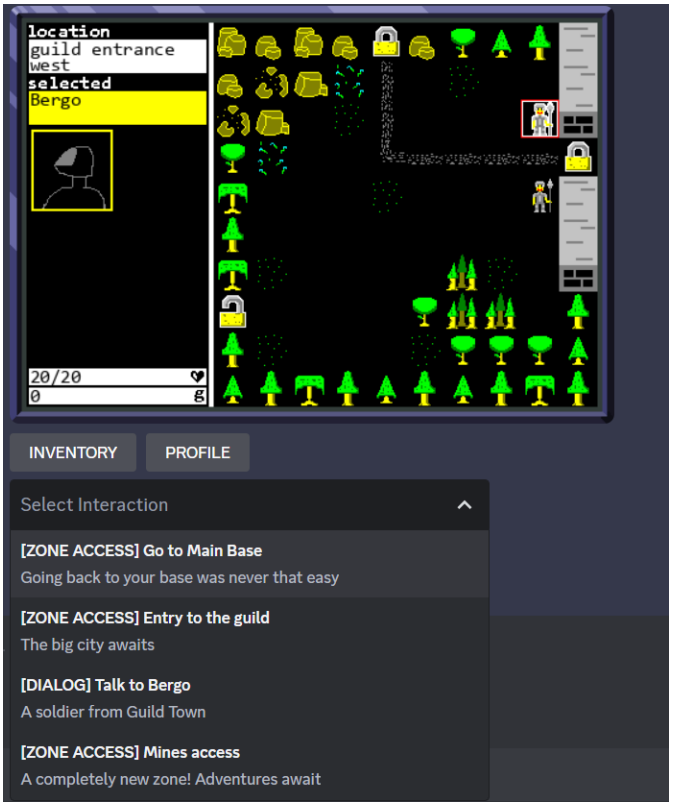
Secundaris:

- **Sistemes de combat:** En la release final s'ha implementat un sistema de combat a l'estil JPRG que permet a dos usuaris tenir un combat online.
- **Deployment:** El joc s'ha aconseguit replicar en una màquina virtual d'Azure pel que podria estar engegat durant tot el dia.

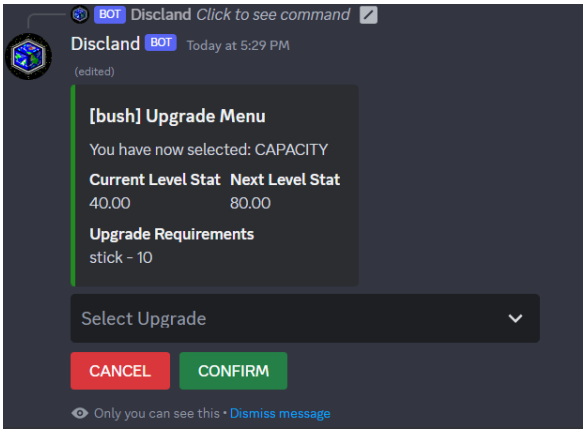
Finalment, al llarg de les següents pàgines es mostren diferents captures finals del joc per així poder contextualitzar el que s'ha descrit durant tot aquest apartat:

Imatge 88: Selecció d'un post (esquerra) i col·lecta de recompenses (dreta). Elaboració pròpia.

Imatge 89: ZoneAcces bloquejada (esquerra). ZoneAccess desbloquejada (dreta). Elaboració pròpia.



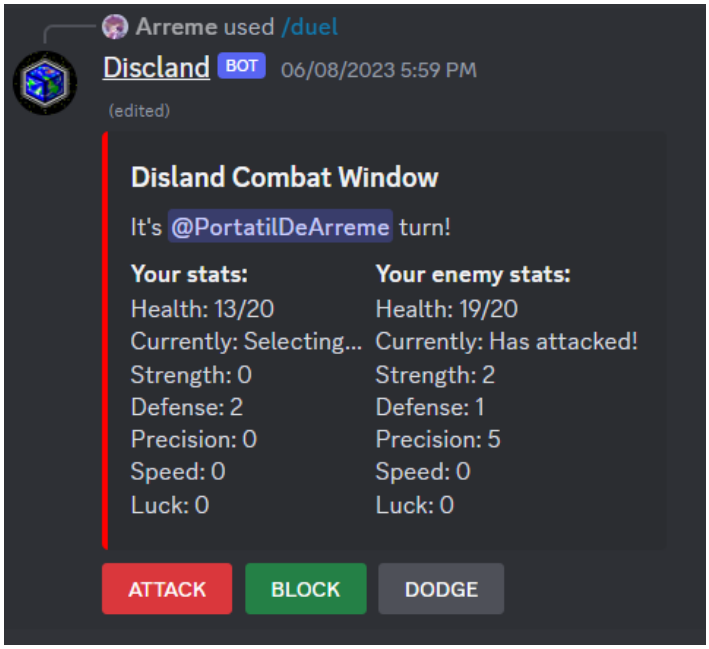
Imatge 90: Parlar amb un NPC. També es pot observar la llista d'interaccions disponibles. Elaboració pròpia.



Imatge 91: Comprar una millora, en aquest cas de capacitat. Elaboració pròpia.

Imatge 92: Inventari. També permet que hi hagi paginació. Elaboració pròpia.

Imatge 93: Estadístiques del jugador. Elaboració pròpia.



Imatge 94: Interfície del combat. Elaboració pròpia.

7 Conclusions

S'han pogut completar tots els objectius que es van plantejar des d'un principi a través de la realització d'una aplicació que mostra signes d'innovació en la comunitat de bots de Discord. A més, tot el procés de creació que va des de la conceptualització de la idea, el procés d'experimentació, la creació de la base de dades... Fins arribar al resultat final ha estat documentat en aquest treball per a que futurs desenvolupadors puguin tenir un esquema clar de cara a crear aplicacions similars.

Posant en vista pel que fa l'aplicació en si, un dels senyals més evidents sobre l'èxit del seu desenvolupament és que el seu potencial avui dia és monumental:

- Es poden afegir noves zones en qüestió de minuts.
- Plantejar interaccions diferents com: Botigues per a comprar objectes, NPCs capaços de plantejar missions als jugadors o construcció d'estructures que aporten millores al gameplay.
- Desenvolupar sistemes de combat més complexos amb habilitats pot ser una addició interessant al joc. A més, es podria acompanyar juntament amb tot el mòdul de renderització per poder visualitzar aquestes interaccions de la mateixa manera que es mostra el mapa.
- Creacions de masmorres generades de forma procedimental. Senzillament adaptant la base de dades a que guardi la informació del mapa generada a partir d'algorismes com la tècnica *wave function collapse*.

Aquesta llista d'idees és perfectament factible d'implementar a l'aplicació actual gràcies a tot l'esforç realitzat en la part d'arquitectura, complint així amb un dels objectius principals que buscava aquesta flexibilitat en el codi. De fet, un dels grans aprenentatges i demostrats de forma empírica amb aquests resultats ha sigut la rellevància de garantir una bona arquitectura. En un camp com en el dels videojocs on els canvis són constants i els requisits estan subjectes al disseny aquesta mentalitat pot suposar l'èxit o el fracàs de l'aplicació. És la feina dels programadors evitar que aquest col·lapse sigui donat per la rigidesa de la lògica programada.

Un altre dels aprenentatges que mereix ser mencionat en les conclusions és la tria meticulosa de llibreries. Desenvolupar aplicacions en un entorn tan divers i amb tantes tecnologies implica acabar lligat a codi extern que resol de forma compacte problemes comuns. Realitzar una bona decisió al principi sobre les llibreries pot ser un factor determinant no només de cara a l'eficiència o a característiques més tècniques sinó al procés qualitatiu del desenvolupament. Un exemple aplicat al treball va ser la decisió d'incloure els Protobufers com a eix central de l'esquema de dades. La millora qualitativa de codi gràcies a les facilitats que ha implicat treballar amb els arxius .proto s'ha notat de forma substancial.

Tot i així cap desenvolupament és perfecte. A dia d'avui algunes mancances importants encara continuen formant part de l'aplicació:

- El sistema de request codificat dins del mòdul de l'API encara té alguns problemes de concepte. La millora respecte el game manager és evident a través del seu nou sistema de classes que permet reutilitzar codi, però la funcionalitat no està prou ben definida i es creu que en un futur podria ser complicat de mantenir.
- El mòdul de db_handler segueix tenint traces del sistema antic i que no s'han pogut migrar per complet a causa del marge de temps estret per a realitzar tot el refactor.
- Algunes crides a la base de dades continuen amagant lògica interna del programa.
- Un sistema de cache per a guardar dades utilitzades amb freqüència pot acabar de reduir la dependència tan forta amb MongoDB

Una altra mancança que no s'ha treballat ja que es desviava del desenvolupament de l'aplicació principal ha sigut tota la part que fa referència al màrqueting i a la gestió de vendes. Tot i tenir un potencial a nivell mecànic, un bot incapaç de generar una llista d'usuaris actius que ajudin a crear una comunitat no té un futur massa brillant de cara a llarg termini. El manteniment d'un servei de *hosting* no és car però igualment suposa un cost rutinari que, en el pitjor dels casos, hauria de ser recuperat. Per això, idear un sistema de monetització és una tasca clau si es vol continuar amb el desenvolupament de l'aplicació, implementant per exemple contingut de pagament en forma d'aventures noves o de cosmètics.

Finalment, el treball també posa en context la complexitat de realitzar aquest tipus d'aplicacions tot i tenir a l'abast milers i milers d'arxius, pàgines web i vídeos explicatius de forma instantània a través d'Internet. La creació dels primers Multi User Dungeons es va dur a terme en una situació molt més complicada que ara ja que molts dels problemes que a dia d'avui poden ser trivials segurament van haver de ser pensats des de zero. Aquest fet però no va evitar que els seus creadors fessin història, tant dins dels seus mons virtuals com a fora de les pantalles i al llarg del temps.

8 Bibliografia

Bartle, R. (2010). *From MUDs to MMORPGs The History of Virtual Worlds.* Essex.

Bartle, R. A. (1996). Hearts, Clubs, Diamonds, Spades: Players Who Suit MUDs.

Bartle, R. A. (2003). *Designing Virtual Worlds.*

Bouchard, R. P. (29 / Juny / 2017). *How I Managed to Design the Most Successful Educational Computer Game of All Time.* Recollit de Medium: https://medium.com/the-philipendium/how-i-managed-to-design-the-most-successful-educational-computer-game-of-all-time-4626ea09e184

Boutell, T., & Joye, P. (sense data). *What is the GD library*. Consultat el 19 / Abril / 2023, a gdLibrary: https://libgd.github.io/pages/about.html

BSON Specification. (sense data). Consultat el 18 / Abril / 2023, a https://bsonspec.org/

Cox, A., & Campbell, M. (1994). Multi-User Dungeons. A A. Cox, & M. Campbell, *Interactive Fantasy 2* (p. 15-20). Hogshead Publishing. Recollit de https://mud.co.uk/richard/ifan294.htm

Curry, D. (9 / January / 2023). *BusinessOfApps*. Recollit de Discord Revenue and Usage Statistics (2023): https://www.businessofapps.com/data/discord-statistics/

D++. (2023). *What is D++?* Recollit de D++ (DPP): https://dpp.dev/

Discord. (2023). *What is discord?* Recollit de Discord.com: https://discord.com/safety/360044149331-what-is-discord

Discord. (n.d.). *Discord Developer Platform*. Recollit de Discord.com: https://discord.com/developers/docs/intro

Douglas, J. (2007). *Command Lines.* California.

Epic RPG. (4 / Abril / 2022). *Epic RPG Wiki*. Consultat el 21 / Abril / 2023, a Fandom: https://epic-rpg.fandom.com/wiki/EPIC_RPG_Wiki

Fette, I., & Melnikov, A. (Desembre / 2011). *The WebSocket Protocol*. Recollit de rfc: https://www.rfc-editor.org/rfc/rfc6455

Google. (2023). *Protocol Buffers*. Consultat el 4 / Abril / 2023, a Protocol Buffers Documentation: https://protobuf.dev/

Google. (n.d.). *Real-time communication for the web*. Recollit de Webrtc.org: https://webrtc.org/?hl=es-419

Gupta, L. (7 / Abril / 2022). *What is REST*. Recollit de REST API Tutorial: https://restfulapi.net/

Hardt, D. (10 / 2012). *The OAuth 2.0 Authorization Framework*. Recollit de RFC editor: https://www.rfc-editor.org/rfc/rfc6749

King, G., & Krzywinska, T. (2002). *ScreenPlay: cinema/videogames/interfaces.* London: Wallflower Press.

Microsoft. (3 / Març / 2023). *Resource access management in Azure*. Recollit de Microsoft Learn: https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/get-started/how-azure-resource-manager-works

Microsoft. (2023). *What is Azure?* Recollit de Azure: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-azure/?cdn=disable

Miriam, G. (2019). *A Study of Community Engagement on Discord for Game Marketing*. Recollit de Theseus: https://www.theseus.fi/bitstream/handle/10024/266217/Thesis_Greenberg_Miriam.pdf

Monaco, N., & Wolley, S. (2022). *Bots.* John Wiley & Sons.

Monfort, N. (2011). Toward a Theory of Interactive Fiction. A E. Short, D. Jerz, & K. Jackson-Mead, *IF Theory Reader.* Boston.

MongoDB. (2023). *What is MongoDB*. Recollit de MongoDB: https://www.mongodb.com/en/what-is-mongodb

Montfort, N. (2013). *Riddle Machines: The History and Nature of Interactive Fiction*. Consultat el 10 / 2 / 2023, a http://onlinelibrary.wiley.com/doi/10.1002/9781405177504.ch14/summary

Norman, D. A. (2013). *The Design Of Everyday Things.* MIT Press.

Oauth. (n.d.). *Oauth.net*. Recollit de https://oauth.net/2/

Sorolla, R. S. (1996). Crimes Against Mimesis. A E. Short, D. Jerz, & K. Jackson-Mead, *IF Theory Reader* (p. 7-30). Boston.

Strategic Simulations. (1984). *Questronn A Fantasy Adventure Game.* Consultat el 21 / Abril / 2023, a Questron Reference Card: https://www.mocagh.org/ssi/questron-refcard.pdf

Vass, J. (10 / September / 2018). *How discord handles two and half million concurrent voice users using webrtc.* Recollit de Discord: https://discord.com/blog/how-discord-handles-two-and-half-million-concurrent-voice-users-using-webrtc

Wolf, M. J. (2000). Genre and the Video Game. *The Medium of the Video Game*.

Wolf, M. J. (2012). *Encyclopedia Of Videogames: M-Z* (Vol. Vol. 2). ABC-CLIO.

World Wide Web Consortium. (26 / January / 2021). *WebRTC 1.0: Real-Time Communication Between Browsers*. Recollit de World Wide Web Consortium: https://www.w3.org/TR/webrtc/

TecnoCampus

Centres universitaris adscrits a la

upf. Universitat Pompeu Fabra Barcelona

Doble titulació en Enginyeria Informàtica i Disseny i Producció de Videojocs

Creació i desenvolupament d'una aplicació interactiva en línia a través de l'API de Discord

ANNEXOS

Arnau Reig Méndez
Tutor: Rafael González Fernández
2022-2023

TecnoCampus Mataró-Maresme

Codi Font:

1. Primera Release Discland: “DisclandRelease1.zip”
2. Segona Release Discland: “DisclandRelease2.zip”

Diagrames UML:

1. Mòdul Core: “CoreUML.png”
2. Mòdul DB: “dbUML.png”
3. Mòdul Img: “Rendering.png”

Vídeo

1. Vídeo de l’aplicació: “DisclandVideo.mp4”

Document Tècnic:

1. Document per a la configuració: “DocumentTecnicDiscland.pdf”