

- INDEX -

Introducció	Pag. 2
Eines utilitzades	Pag. 3
Disseny i modelat	Pag. 3
Motor gràfic	Pag. 4
Llenguatge utilitzat	Pag. 4
Llibreria gràfica 3D	Pag. 6
Multimedia Help Library	Pag. 8
Só	Pag. 9
Especificacions tècniques	Pag. 10
Disseny i modelat	Pag. 10
Entorn	Pag. 10
Items	Pag. 12
Personatges	Pag. 14
Armes	Pag. 18
Mapes	Pag. 18
Textures	Pag. 21
Sons	Pag. 23
Interfície gràfica	Pag. 23
Motor gràfic	Pag. 25
Algoritmes de renderitzat	Pag. 25
Algoritmes de renderitzat d'interiors	Pag. 25
Forma de pintar	Pag. 29
Classes	Pag. 29
Funcionament general	Pag. 33
Organització del codi	Pag. 35
Xarxa	Pag. 38
Tecnologies emprades	Pag. 41
Fitxers de configuració	Pag. 51
Col·lisions	Pag. 54
Rol	Pag. 56
Conclusions	Pag. 60
Problemes coneguts o coses pendents	Pag. 62
Annex.....	Pag
63	
Connexió a base de dades.....	Pag
64	
Col·lisions.....	Pag 69
Pressupost.....	Pag
70	

Introducció

El projecte sobre el que llegireu i sobre el que hem treballat ha consistit i consisteix en la creació, des de zero, d'un videojoc 3D d'acció en primera persona. Al estil de videojocs històrics com Doom o Quake.

En aquest projecte hem volgut elaborar un joc des de zero, saber quins passos segueix la creació d'un joc, en quines parts es divideix un videojoc d'aquest estil, en quins camps s'ha de treballar per assolir un resultat complet i, naturalment, divertir-nos en el procés i sentir la satisfacció de contemplar un resultat final digne de les nostres expectatives.

El primer que vam fer va ser estudiar els detalls del videojoc. Quins objectius volíem assolir i quines característiques volíem que tingués el resultat final.

Les nostres intencions eren fer un joc d'acció 3D en primera persona, tipus de joc també anomenat *shoot-em-up*. Aquests són els jocs més populars en el món de la informàtica actualment.

També vam decidir que el nostre videojoc tenia que estar orientat al joc en xarxa, ja que pensem que en un futur cada cop més proper els jocs s'aniran movent més i més en aquest camp. De fet, actualment ja hi ha una representació generosa en el mercat de videojocs exclusivament orientats al joc en xarxa. Actualment queda poca gent amb ordinador i sense internet, i aquest grup de individus cada cop s'anirà reduint més fins a ser nul.

Així, el nostre projecte gira totalment al voltant de la informàtica, però hem tingut que treballar en camps molt diversos dins d'aquesta. Així es compren com, en la indústria del videojoc i tenen cabuda tants tipus diferents de professionals informàtics, i com, en la creació d'un videojoc, intervenen tants d'aquest professionals en camps diferents.

Per tant, hem volgut emular els projectes professionals i dedicar-nos, cada un dels components del grup, exclusivament a una part de la creació del videojoc. Dividint-nos la feina en, la creació del motor del joc, la creació de la interacció per xarxa i la creació del entorn 3D i tota la interfície gràfica i multimèdia.

Eines utilitzades

DISSENY I MODELAT

En la part de la creació multimèdia hem empleat els programes més estesos i amb els que nosaltres estàvem familiaritzats.

Així, per a la creació de tot l'entorn 3D hem utilitzat l'eina de *Discreet, 3D Studio Max*. La versió empleada d'aquest programa ha estat la 4, la última existent al començar el projecte, i posteriorment la 5, llençada al mercat per *Discreet* a meitat de projecte. Ens vam actualitzar de versió degut a que ens vam adonar de que la nova versió ens oferiria petites avantatges a l'hora de crear els objectes 3D complexos com personatges.

Aquesta eina es suficientment potent per assolir els objectius del nostre projecte, a més a més de que ja teníem alguns coneixements sobre aquest i de que aquest ens oferia la, per nosaltres, necessària eina d'exportació en ASCII que el format d'exportació *.ASE ofereix.

Altres motius per els que hem utilitzat el *3D Studio Max*, es la eina de *Character Studio* que utilitza per a la animació de bípedes, ja que d'altra manera la animació d'un esquelet es fa molt més complicada, encara que possible, ja que s'hauria de crear ós a ós l'esquelet i posteriorment animar-lo ós a ós també. I encara que el *3D Studio Max* permet també aquesta última possibilitat més complicada, finalment hem escollit l'eina del *Character Studio* per la seva potencia i per facilitar la tasca d'animació.

Finalment dir que una altre de les raons per la que hem escollit el *3D Studio Max* és la, no menys important, ajuda que podíem rebre del professorat del centre en cas de que ens trobéssim amb entrebancs que no sabéssim com superar i degut a que es el software amb el que s'imparteixen les classes al centre.

Pel que fa a la creació i edició de les textures el programa utilitzat ha estat el *Adobe Photoshop* en la versió 7. Aquest editor d'imatges és ideal per retocar d'imatges ja existents per tal d'adaptar-les a la textura perquè encaixi amb el model 3D.

A més a més, es tracta de l'eina de retoc d'imatges més estesa i, per tant, de la que podíem extreure més suport en cas de necessitat. I novament podíem trobar en el professorat una font d'ajuda si ens sorgia algun contratemps que no sabéssim com solucionar. Ja que en el centre es disposa de professors amb experiència en la utilització d'aquesta eina.

Altres opcions vàlides haguessin estat *Paint Shop Pro* o *GIMP*, però tenint en compte les raons anteriors i que ja gaudíem d'experiència en aquesta eina, ens vam decidir per aquesta i no una altra.

MOTOR GRÀFIC

LLENGUATGE UTILITZAT

Hi ha molts llenguatges diferents per a escollir i cada un té els seus avantatges i inconvenients. A continuació descriurem les avantatges i inconvenients dels principals que hem plantejat utilitzar.

Troblem *Visual Basic* de *Microsoft* que és molt famós per la seva facilitat d'ús. En un moment pots tenir una aplicació gràfica en *Windows* i vincular-la a una base de dades. És clar, tot això està molt bé però cal recordar que estem parlant d'un videojoc i, per tant, no ens serveix. *Visual Basic* està enfocat a aplicacions comercials per a petits i mitjans negocis però no per a servir a les nostres necessitats.

Un altre llenguatge a tenir en compte és *Java*. Aquest ja és un llenguatge més enfocat a tot tipus d'aplicacions i ens podria haver donat uns millors resultats que l'analitzat anteriorment. Entre les seves principals avantatges trobem la seva gran consistència, seguretat i orientació a objectes. *Java* gestiona automàticament la memòria i, per tant, permet que ens oblidem en gran part de

punters, passar variables per valor o referència, etc. Té un sistema de excepcions que ajuda a no accedir a zones de memòria no assignades i assabentar-nos de quan, a on i de quin tipus a succeït un error d'execució. Fins ara tot semblen avantatges però realment no ho son del tot. El control automàtic de memòria de *Java* dificulta la optimització de codi i la plena llibertat i facilitat per a tractar les dades d'una manera ràpida i directa. Un altre problema que té es que es un llenguatge força lent i en un joc necessites molta velocitat per a aconseguir uns bons resultats.

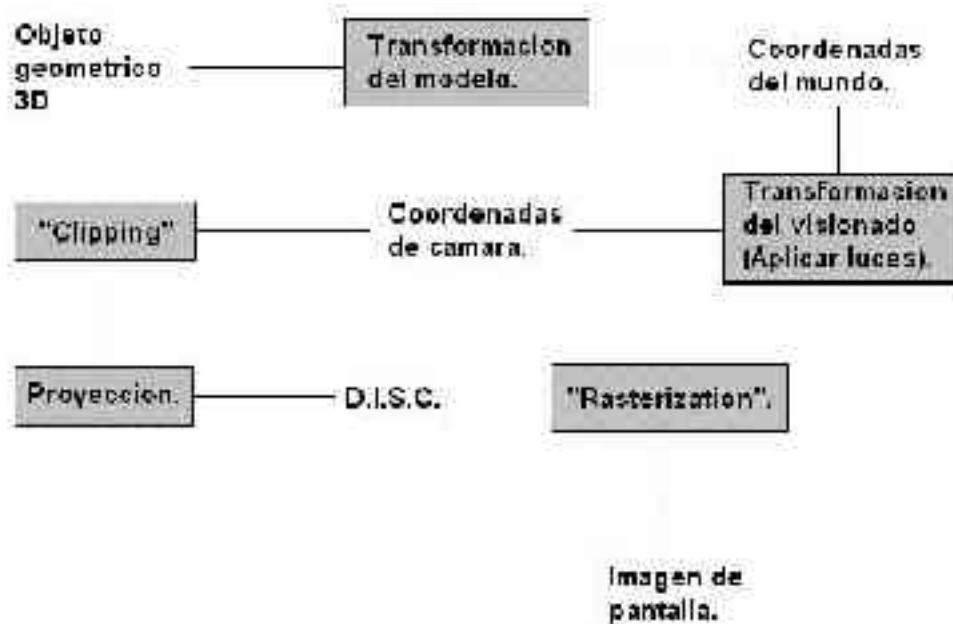
La tercera possibilitat podia ser realitzar el joc en *C*. Aquest llenguatge de força antiguitat (la seva creació començar el 1972) ha sigut, probablement, un dels llenguatges més utilitzats de tots els temps. La major part dels jocs de fa uns quants anys eren programats en aquest llenguatge. La seva facilitat d'ús i la seva potencia permeten que sigui un llenguatge ideal per a gent que acaba de començar a obrir-se camí en el món de la programació i gent que és tota una experta. De la mateixa manera, també permet fer petites i grans aplicacions amb una gran portabilitat i qualitat. Algunes de les seves majors avantatges es que és un llenguatge molt ràpid i portable. Si només contéssim aquests punts, hagués sigut el llenguatge ideal. La raó per la qual no l'hem escollit ha sigut la seva orientació a funcions. *C*, al no estar orientat a objectes, hagués dificultat força el control i la organització d'aquest. Aquest ha sigut el factor que ens ha fet inclinar cap a l'últim llenguatge analitzat.

Aquest es el llenguatge que hem acabat escollint. Les característiques que el fan superior als altres es la seva orientació a objectes, la gran llibertat que et dona, la seva portabilitat i la seva facilitat d'utilització. Avui en dia es el principal llenguatge de programació de videojocs i, encara que no és tan ràpid com *c*, es molt més fàcil d'utilitzar.

LLIBRERIA GRÀFICA 3D

Les llibreries 3D son les encarregades de controlar els diferents passos que trobem entre que creem l'objecte 3D fins que apareix per pantalla.

Aquest seria un exemple de les funcions que realitzen.



Objecte geomètric: És la unió de totes les primitives (línies, punts, polígons, etc.) que formen el món.

Transformació del model: S'encarrega de col·locar, rotar i escalar els objectes.

Coordenades del món: En aquest punt les coordenades del món no són relatives a la càmera (el punt on ens trobem) sinó a uns eixos de coordenades preestablerts. Ens situem en el món.

Transformació del visionat: Il·luminem els objectes i agafem la seva posició respecte la càmera.

Coordenades de càmera: En aquest punt ja sabem com veiem els altres objectes des de on estem.

Clipping: Retallem tot allò que no veu la càmera (tot el que està fora del nostre camp de visió).

Projecció: Passem les coordenades 3D a 2D.

D. I. S. C. (*Device Independent Screen Coordinates*): Associem la imatge 2D que obtenim amb els píxels de la pantalla.

Rasterització: S'il·luminen els fòsfors de la pantalla per a formar la imatge.

Imatge en la pantalla: Obtenim la imatge desitjada.

Un cop vista breument una de les funcions principals d'una llibreria 3D passarem a explicar les avantatges i desavantatges de les 3 més utilitzades i la raó per la qual hem escollit *OpenGL* davant de les altres.

La primera que analitzarem serà *Java3D*. Aquesta extensió del *JDK2* de *Java* proporciona un gran conjunt de llibreries i constructors per a manipular objectes 3D. Permet crear un univers virtual del tamany que és vulgui, renderitzar en paral·lel gràcies a la utilització de fils en *Java*, visualització web via *Applet*, etc. La raó per la qual hem descartat aquesta llibreria es que no ens permetia fer servir C++ com a llenguatge de programació i no és tan òptima per a la creació d'un videojoc (Encara que hi ha molts videojocs fets en *Java3D*, està més enfocada a la creació d'aplicacions d'altres tipus).

La segona gran API que podem trobar es *DirectX* (concretament *Direct3D*) de *Microsoft*. Aquesta API és de les més utilitzades en programació de videojocs ja que aporta un gran nombre de solucions diferents per a la creació d'aplicacions. Esta pensada per a treure-li el màxim de partit a la tarja gràfica (sobretot des de la aparició de *DirectX 9*). Entre els principals problemes que té, trobem la necessitat de tenir una tarja gràfica força actual per a treure-li el màxim profit (encara que això passa de la mateixa manera amb *OpenGL*), la seva manca de portabilitat (només funciona correctament en S.O. *Windows*) i la seva dificultat d'utilització (requereix un alt coneixement de la API de *Windows*). Per aquestes raons l'hem deixa't de banda en el nostre projecte.

La tercera API que vam tenir en ment (i amb la que ens vam quedar) va ser *OpenGL*. Originàriament anomenada *GL (Graphics Library)* i creada per *Silicon Graphics* durant un projecte intern va acabar ser oberta de codi per al seu lliure ús. Entre les seves principals avantatges trobem la seva independència del sistema operatiu utilitzat i la gran facilitat d'ús que té. També té suport per a multitud de llenguatges de programació (*C, C++, Java, Delphi*, etc). Tot això ha sigut el que ens ha impulsat a fer-la servir. Actualment va per la versió 1.6 però pròximament apareixerà la 2.0 que promet desbancar a *DirectX 9* (encara que, com sempre, haurem de tenir una bona tarja per treure-li el màxim profit).

MULTIMEDIA HELP LIBRARY

Un altre problema que va sorgir durant la resolució del projecte va ser com crear les finestres, controlar els events de teclat, de mouse, etc. Podríem utilitzar directament la API de *Windows*, i fer el mateix per a *Linux*, però vam considerar que no era una solució molt bona existint llibreries que ens poden facilitar la feina extremadament.

Glut (OpenGL Utility Toolkit) va ser la primera llibreria de suport que vam fer servir. És molt fàcil d'utilitzar i ofereix bastantes possibilitats. Al principi ens va anar bé però poc a poc es va anar quedant curta (sobretot en el que es refereix a events de teclat) i vam decidir prescindir d'ella.

DirectX també té suport per al control d'events de teclat (*Direct Input*) però per la seva poca portabilitat no la vam utilitzar.

Fltk és una llibreria portable molt fàcil d'utilitzar. És programada en *C++* i, per tant, està orientada a objectes. Això comporta una gran comoditat a la hora de programar. Té suport complet per a *OpenGL* i és compatible amb algunes funcions de *Glut*. Tot i així, no la vam agafar ja que n'hi havia una altra d'encara més potent.

La llibreria multimèdia que vam escollir al final va ser *SDL* (*Simple DirectMedia Layer*). Aquesta llibreria (programada en C) està pensada especialment per a fer videojocs, és portable a multitud de SO i utilitzable en molts llenguatges de programació. Incorpora un ple suport per a *OpenGL* i funcions per al control de vídeo, events de teclat i mouse, fils, temporitzadors, só (encara que no l'utilitzem) i *CD-Rom* (que tampoc utilitzem)

SO

El só és un apartat molt important en un videojoc. Aquest ajuda notablement a augmentar la sensació de realisme que sentim al jugar i fa el joc més atractiu.

DirectX torna a ser una de les solucions possibles. Aquesta vegada amb el seu *Direct Sound*, que permet utilitzar só 3D. Per raons de portabilitat va quedar descartat.

SDL també permet só 3D. Aquesta llibreria (que també utilitzem per a controlar les finestres, events de teclat, etc.) hagués sigut una bona solució. La principal raó per la qual no l'hem escollit és que és un pel limitada i no creiem que estigués a l'altura de les nostres necessitats.

OpenAL són unes llibreries per àudio molt potents. Durant un temps van estar una mica estancades i no van ser molt utilitzades, però ara mateix estan ressorgint. Uns clars exemples de jocs que utilitzen aquestes llibreries són el *Unreal Tournament 2003* i el *Unreal 2*. Hem escollit aquesta enlloc de les altres per la seva potència, la seva gran semblança a *OpenGL* en qüestions de programació i la seva enorme portabilitat a altres sistemes operatius.

Especificacions tècniques

DISSENY I MODELAT

En la creació d'un videojoc la part gràfica i multimèdia d'aquest assoleix un paper molt important en el resultat final del projecte, ja que es aquesta part del joc la que permetrà a l'usuari final visualitzar la interacció que es du a terme entre joc i usuari.

La part multimèdia del projecte engloba tot component del joc que fa d'interfície entre el joc i l'usuari amb la que l'usuari es relaciona i que ofereix informació a aquest sobre les conseqüències de la seva interacció. En definitiva, permet que la comunicació entre el programa i el jugador s'assoleixi entretinguda, visual i de fàcil comprensió per a qualsevol que faci us d'aquest.

Es poden englobar en dos grans blocs les parts del joc que integren el conjunt multimèdia del programa.

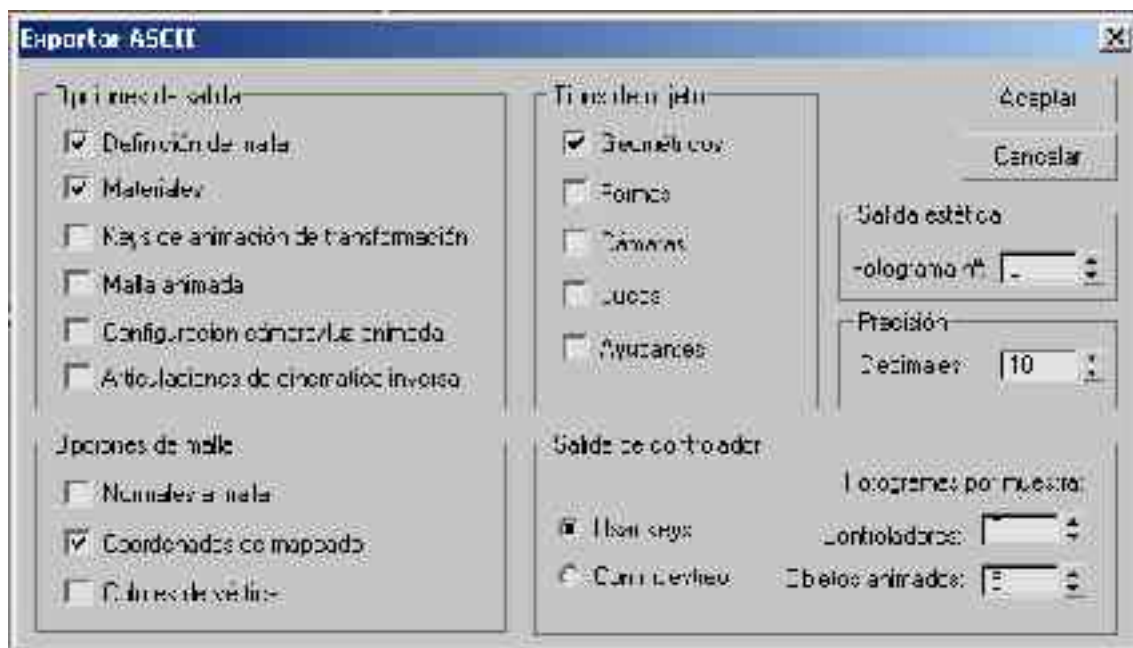
Per una banda tenim l'Entorn, on l'usuari interacciona amb el joc, com son tots els elements 3D que donen forma al mon virtual en el que es succeeix el joc, o els sons que ajuden a una millor ambientació i a una major immersió de l'usuari en el joc.

L'altre bloc es refereix a la Interfície Gràfica, en la que el joc mostra tota la informació necessària per tal de que l'usuari estigui al corrent del seu estat d'una manera còmoda i amb un simple cop d'ull, sense que tingui que desviar l'atenció del joc per rebre aquestes dades.

ENTORN

El modelat de tot el joc ha estat fet íntegrament amb l'eina *3D Studio Max* en la versió 4 i posteriorment en la versió 5, la més actual. Per al modelat s'ha procedit a treballar de manera totalment lliure només tenint en compte certes restriccions que permeten a posteriori la correcte conversió, lliure d'errors, als formats propis del joc. Per a poder convertir un disseny elaborat amb *3D Studio Max* a un dels formats comprensibles per al motor del joc, un cop

finalitzat el procés de modelat, havent tingut en compte les restriccions necessàries per a una correcta interpretació del conversor, cal exportar el resultat final obtingut a un format concret que ofereix el *3D Studio Max*, el format *ASE*, quedant finalment l'arxiu amb el nom "arxiu.ase". El format *ASE* permet varies opcions per tal de guardar un model 3D en un format de tipus text (*ASCCl*) que permetrà una major facilitat de lectura i interpretació pels conversors del nostre joc que passaran aquest *ASE* al format concret que utilitza el joc. De les diferents opcions que permet l'exportació amb *ASE*, les que ens interessin son les que tenen relació amb la posició de cada vèrtex (que exporta les coordenades de cada eix per cada vèrtex del model) i les coordenades de la textura (pell) del model. A més a més, l'exportació en *ASE* permet especificar el número de decimals amb que expressar les coordenades, sent convenient indicar-li que porti a terme la exportació amb el major numero de decimals possibles per una major exactitud i una reducció al màxim de petits errors.



D'aquest format, gràcies als conversors propis del joc, es passa a cada un dels formats diferents que el joc reconeix pels mapes, models i ítems, fent la lectura del fitxer *ASE* amb tota la informació necessària per representar el model i creant un nou fitxer amb les dades redistribuïdes.

Un cop passat pels nostres conversors, el model esta preparat per ser carregat i representat pel motor del joc.

Un dels factors més importants a tenir en compte al modelar per un videojoc és el número de cares amb el que es dotarà cada objecte, pensant globalment en les cares que el joc representarà amb tot el conjunt d'objectes que apareguin a escena a l'hora.

Això és tant important perquè el número total de cares que cada ordenador és capaç de visualitzar per pantalla a l'hora és proporcional a la potència d'aquest, i sobre tot a la potencia de la seva targeta gràfica. Per tant, el més convenient és no superar un número sensat de cares per objecte, de manera que es porti un cert control de les cares que el joc tindrà que moure a la vegada i d'aquesta manera assegurar que el videojoc funcionarà fluidament en el màxim número d'ordenadors possibles.

Finalment també cal controlar els tamanys i proporcions de cada element del entorn 3D del videojoc. A l'hora de crear un element es convenient crear la resta agafant com a mostra de les proporcions el primer dut a terme. D'aquesta manera s'assegura conserva les mateixes proporcions tant per a mapa, com per a models, armes, ítems o qualsevol altre element. De totes formes, prevenint això, els mateixos conversors tenen opcions de reescalat dels vèrtexs per múltiples. Així s'evita tenir que modificar de nou les dimensions de l'objecte des del *3D Studio Max* i tornar a fer els passos necessaris per exportar-lo. D'aquesta manera, si es comet un error en les proporcions del objecte es pot solucionar a traves del conversor.

Com hem dit, cada element del entorn 3D ha de complir certs requisits per tal de que la conversió a formats intel·ligibles per al motor del videojoc es faci correctament. Per a cada element aquests requisits varien, així que a continuació passarem a descriure la forma correcta de crear aquests elements:

Ítems

En aquesta categoria s'engloben tant els ítems estàtics, com ítems amb animació, com les armes dels personatges vista en 3a persona.

Els ítems estàtics són aquells objectes 3D del entorn que són interactius amb el jugador, es a dir que aquest els pot agafar i aquests li ofereixen millores al seu personatge, són objectes com vida, armadura i munició.

Els ítems amb animació són aquells objectes estàtics que estan a la espera d'un event dut a terme pel jugador per tal d'iniciar una animació i tornar al seu estat inicial un cop acabada, són objectes com portes que s'obren i tanquen.

Les armes vistes en 3a persona es refereix a les armes que deixen caure els jugadors al morir, que són l'arma que estava fent servir en aquell moment. També es refereix a l'arma que veiem en les mans dels jugadors contrincants, que són el mateix objecte que el de les armes dels jugadors morts, però aquestes són enganxades a les mans del model del personatge mitjançant codi.

Per tal de fer el modelat dels Ítems, simplement cal modelar l'objecte amb *3D Studio Max*, amb la tècnica que es prefereixi, tenint un compte un límit de cares sobre les 100 – 120. Un cop el objecte esta totalment modelat s'ha de convertir aquest en “*Malla Editable*” per tal de que l'ASE exporti correctament les coordenades dels vèrtex.

Per a texturitzar l'objecte cal seguir una tècnica comú en tots els objectes del joc i aquesta s'explica en el seu propi apartat una mica més endavant.

Posteriorment, un cop ja texturitzat, ja només caldrà posar el pivot en la posició que vulguem, ja que depenent de la posició en la que el posem, aquest es comportarà de forma diferent quan li apliquem la funció que fa que els objectes rotin i es belluguin amunt i avall. A més a més la posició del pivot determinarà la posició de l'objecte respecte la superfície del mapa.

En el cas dels objectes estàtics però amb animació, com poden ser portes o qualsevol altre objecte que es vulgui, simplement cal modelar i texturitzar com en el cas dels ítems estàtics i posteriorment fer la animació amb la tècnica q es vulgui. A continuació, s'haurà d'exportar cada frame de la animació un per un fins a completar el conjunt de l'animació al format ASE, anomenant a cada

arxiu amb una correlació lògica de números (*nom000.ase,nom001.ase...*). A l'hora que s'apunta de quin a quin fitxer va cada animació en un fitxer anomenat "*frames.txt*" en el que s'apunta en cada línia una animació i, en cada línia, primer el frame inicial, a continuació el frame final i per últim s'apunta si es una animació cíclica que s'hagi d'anar repetint infinitament (amb un 1) o si per el contrari la animació només s'ha de reproduir un sol cop (amb un 0). Després mitjançant codi ja crida la animació corresponent representant de manera seguida i continuada cada un dels fitxers que representen cada frame de l'animació.

Finalment, les armes que es veuen en 3a persona també estan englobades en aquesta categoria d'objectes, ja que a la pràctica no són més que objectes estàtics i inanimats que es pot representar sobre les mans dels models dels jugadors, Això es fa mitjançant el pivot de l'arma que s'ha de col·locar en el punt per el que volem que el personatge empunyí l'arma. Al model del personatge també se li ha de col·locar un indicatiu que mostri al motor del joc per on ha d'unir l'arma i el personatge, això s'explica a continuació.

La extensió feta servir per als ítems al joc es *.mdh.

Personatges

El tractament dels models dels personatges és una mica més complicat que la resta d'elements 3D del joc.

Per tal de adaptar el model al funcionament intern del joc s'ha decidit dividir el personatge en 3 parts diferents que s'han de modelar de forma independent o be modelant-ho tot junt, com un sol objecte, i posteriorment dividint-ho en les tres parts. Aquestes 3 parts en las que s'ha de dividir el personatge són: el cap, el tors amb els braços (de cintura cap amunt menys el cap) i les cames (de cintura cap avall).

Les raons de dividir el personatge en aquestes 3 parts són varies. La primera es la de facilitar d'una manera estàndard la detecció de danys segons la zona del cos, així evitem el problema sorgit perquè el joc detectes les diferents parts

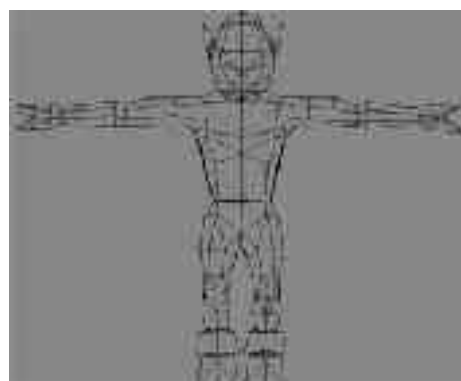
del cos amb un únic model 3D en el que segons el personatge les dimensions i proporcions podrien variar. La segona es la de facilitar les animacions del cos per tal de que aquestes quedin ordenades d'una manera més modular i no tenir que fer així un sol arxiu amb tot el pes de les diferents animacions.

D'aquesta manera el cap es mou independentment segons cap on miri el jugador. El tors s'anima de diferent forma segons l'arma que estigui fent servir el jugador en aquell moment (ja que no s'agafa d'igual forma una destrat, una pistola o un fusell). I les cames s'animaran de diferent manera segons camini, corri, salti o s'ajupi i en la direcció en que ho faci.

Es molt més fàcil fer aquestes animacions en 3 objectes diferenciats que en un de sol, ja que si es fes en un sol objecte es tindrien que contemplar les diferents combinacions d'animacions de les 3 parts del cos dutes a terme en un sol objecte, amb lo qual el nombre d'animacions total creix considerablement al mateix temps que creix el tamany dels fitxers i així l'agilitat del videojoc en segons quins equips informàtics.

Per tant, tindrem que el personatge estarà compost en la seva totalitat per 3 objectes diferents, cada un amb les seves animacions.

La forma de modelar les diferents parts del cos serà exactament igual que la feta servir pels ítems, però amb un afegit que és la necessitat de modelar el cos amb la famosa postura de *Da Vinci* en la que el cos te les extremitats ben separades del tronc per tal de facilitar la feina de texturització, i sobretot de animació. Posteriorment el cos, si s'ha modelat tot en un sol objecte, s'haurà de dividir en les 3 parts esmentades abans de procedir a la texturització i animació.



La tècnica de texturitzat també serà la mateixa que la empleada pels ítems, però aquesta s'explicarà amb més detall en el corresponent apartat de texturització. La única cosa a apuntar és la necessitat de que cada una de les parts del cos (cap, tronc i cames) disposi de la seva pròpia textura independent de la resta del cos.

En aquest moment es disposa d'un model de personatge dividit en 3 parts i en la postura de *Da Vinci*, totalment texturitzat. A continuació toca animar el personatge.

Abans d'animar el personatge seria convenient indicar els punts d'unió entre les diferents parts del cos i els punts que indiquen al motor del joc de quina manera unir les armes amb el model del personatge. La manera és col·locar uns cub (que no sortiran representats) en els punts d'unió entre les diferents parts del cos. La clau d'aquests indicadors està en el nom, sent aquest "*LOWER_POINT*" pel punt més baix de cada part del cos i "*UPPER_POINT*" pel punt més alt de cada part del cos. Així, per a unir dos parts del cos aquestes s'ajuntaran pel "*LOWER_POINT*" d'un i el "*UPPER_POINT*" de l'altre sent els pivots els punts concrets per on s'uniran.

Per a indicar la manera amb la que el cos agafarà l'arma calen dos indicadors més. Un és el "*WEAPON_POS*" que indica al motor del joc a quin punt del cos a de col·locar el pivot de l'arma. L'altre indicador es un cub anomenat "*WEAPON_DESTIN*" que informa de la direcció en la que mirarà l'arma, és a dir, que l'arma, partint del punt on hem indicat que situï el pivot de l'arma, estarà encarada cap a la direcció on haguem situat aquest segon indicador.

Finalment caldria vincular aquests diferents indicadors als ossos del bípede més adients perquè al animar l'esquelet els indicadors es moguin a l'hora amb aquest i no perdin la posició que han de tenir respecte les parts del cos.

La animació es pot crear amb la tècnica que mes convingui, però la feta servir per nosaltres a estat la del Bípedo del *Character Studio* (component del *3D Studio Max*). Així hem aprofitat la potencia que aquesta eina ens ofereix a l'hora d'animar un cos humanoide.



Primerament hem adaptat l'esquelet del bípede al model creat, un cop fet això hem passat a aplicar un modificador "*Phisique*" a cada part del cos per separat. Aquest modificador permet que la malla del model sobre el que l'hem aplicat s'enganxi a les parts del esquelet del Bípede que acabem d'adaptar al model. De manera que si movem o rotem una articulació o ós del esquelet es mourà amb ell el model que actua en forma de pell.

Un cop arribats a aquest punt el més convenient es conservar aquest arxiu tal i com està i treballar cada animació en un altre fitxer copia d'aquest. Després només caldrà anar movent cada part del esquelet per aconseguir els moviments desitjats i un cop ser el resultat del nostre agrado guardarem la animació en un fitxer del tipus **.BIP*, una opció del *Character Studio* per a guardar només la animació de l'esquelet. A més a més aquest **.BIP* ens servirà per animar l'arma en primera persona com es recorda més endavant.

Després agafant el fitxer original carregarem la animació **.BIP* guardada amb anterioritat i passarem a exportar la animació de cada part del cos.

La metodologia a seguir per crear les animacions de cada part del cos serà la mateixa que la feta servir per animar els ítems estàtics animats, on cada part del cos haurà estat animada independentment de la resta i les seves animacions exportades frame a frame de la mateixa forma que els ítems, diferenciant això si, cada part del cos (*head000.ase*, *body000.ase*, *legs000.ase*) i apuntant correctament de quins fitxers esta formada cada animació en el fitxer "*frames.txt*" i indiquen també si es una animació cíclica o no.

Les animacions creades han estat, pel tronc: repòs, caminant, atacant, amagant arma i traient arma, tot això repetit 3 cops, una per a cada tipus d'arma (cos a cos, arma de foc d'una ma i arma de foc de dos mans).

Per a les cames han estat: repòs, caminant endavant, caminant en darrera, caminant de costat dret/esquerra, corrents endavant, corrents endarrera, saltant i ajupint-se.

Per al cap només hi ha la animació de repòs.

El màxim de cares que el personatge tindria que tenir es de 1000 en el seu total.

La extensió d'aquest fitxers serà igual que els ítems *.mdh.

Armes

Les armes es divideixen en 2 objectes per arma:

El primer és l'objecte estàtic en forma de ítem agafable, que serà exactament igual que un ítem com ja hem explicat anteriorment. Format *.mdh.

El segon és l'objecte animat que està compost per les mans i l'arma en si i que es el que veurem nosaltres en 1a persona en representació a l'arma que estem fent servir en aquell moment. Aquests segons no són més que un ítem animat que només veurà cada jugador el seu. Aquest ítem ha de tenir les mateixes animacions que el model en el tronc (repòs, caminant, atacant, amagant arma, traient arma) i per a fer les animacions es pot fer servir el bíped i la animació empleada per al model del cos, carregant el corresponent *.BIP, encara que no és necessari que la animació coincideixi amb el model ja que la animació en 1a persona serà genèrica per a tots els personatges. Així si es veu q la mateixa animació que la feta servir per als elements en 3a persona no queda del tot be amb la vista en 1a persona, aquesta animació es pot modificar convenientment per tal de que dongui una bona impressió en primera persona.

El format d'arxiu per aquest objecte es la de *.wph.

Mapes

Els mapes consisteixen en objectes independents que formen l'escenari (incloent des de parets a objectes estàtics), objectes transparents pel jugador que indiquen la posició d'elements i uns altres objectes transparents pel jugador que indiquen on es troba cada portal utilitzat per indicar al motor de joc en quina habitació es troba i quines habitacions veu des d'aquesta.

Els objectes estàtics que formen l'escenari han de seguir unes pautes per la correcta interpretació d'aquest pel motor del joc.

Primerament cada un dels objectes independents no s'han de tocar entre ells compartint el mateix espai, perquè si es així el joc pinta la textura de ambdós a l'hora de manera que es sobre posen fent un efecte no desitjat. La manera més senzilla de controlar de que això no passi es fent servir la eina "*Alinear*" del *3D Studio Max* i anant amb cura de controlar-ho.

Un cop tenim tot l'escenari modelat i estant segurs de que cap objecte d'aquest comparteix ni una mica del mateix espai amb un altre objecte del escenari seria convenient de tenir els objectes del escenari agrupats per habitacions per tal de tenir l'escenari més ordenat i no cometre errors innecessaris.

En aquest punt hauríem de fixar-nos de que cap habitació comparteix un mateix objecte amb una altre habitació ja que això portaria a greus errors en les col·lisions. Si es el cas d'una mateixa paret per a dos habitacions (el cas mes comú) el millor que es pot fer és, havent previngut això, fer un pla en representació a cada cara de la paret i que cada pla sigui un objecte diferent al altre i aquests no es toquin entre ells per a cap punt. Si es qualsevol altre objecte que per la raó que sigui esta entre 2 habitacions, la solució es troba en partir l'objecte en dues parts, una per a cada habitació, sent partit aquest just per la línia que fa d'intersecció entre les dues habitacions.

Quan ja estem segurs de que cap objecte comparteix espai amb un altre i de que cap habitació comparteix objecte amb un altre, hem de passar a texturitzar cada element per separat desagrupant de nou cada habitació i tornant-la a

agrupar un cop tots els seus objectes hagin estat texturitzats. De manera que mai hi hagi més d'una habitació a l'hora desagrupada.

Un cop estigui tot el mapa texturitzat cal que cada habitació sigui un sol objecte. La manera de fer això és seleccionant tots els objectes que formen part d'una mateixa habitació, convertir-los en "*Malla Editable*" i fer un "*Attach*" amb tots ells de forma que només quedi un sol objecte que ha de tenir el nom de *HAB*. Si ho em fet be tindríem que tindrà un sol objecte que ha fusionat totes les textures que utilitzaven els objectes de la habitació, de manera que visualment no hagi canviat res.

Els objectes que indiquen la posició de ítems animats e inanimats i altres indicatius, són tan senzills com fer un objecte qualsevol, encara que es preferible que sigui un cub amb un sol segment per cara, i situar-lo a la posició on volem que aparegui l'objecte en qüestió, amb el nom concret acordat prèviament en el codi que indiqui de quin ítem es tracta.

Dins d'aquests objectes també cal incloure els objectes que indiquen la posició inicial que tindrà cada jugador, que són exactament iguals que els que indiquen la posició dels ítems. El seu nom és "*START_POINT*".

Aquests indicadors no seran representats pel motor del joc i no pot haver-hi un indicador d'aquests entre dues habitacions al igual que els objectes estàtics normals.

Finalment els portals s'han d'incloure en el límit de la habitació que fa frontera amb una altre habitació i que aquesta frontera té un punt per el que es visible la habitació amb la que fa la frontera, Ja sigui perquè hi ha una finestra com una porta com qualsevol altre connexió entre les dues habitacions, donant-li com a nom "*PORTAL??*", on ?? es el número de la habitació amb la que fa frontera. Només aclarir que en una frontera entre dos habitacions s'hauran d'incloure 2 portals, un per cada habitació i cada un amb el nom que correspongui.



Arribats a aquest punt ja només queda agrupar tots aquests elements per habitacions. De manera que agafarem tots els objectes indicatius que hem anat col·locant per l'escenari i els portals que estiguin en una mateixa habitació hi amb ells farem un grup anomenat "*HABITACION##*" on *##* es el número únic que identifica la habitació i que a de coincidir amb els números que hem adjudicat als portals de manera que no hi hagin incongruències.

Així al final el mapa tindria que estar format per un numero habitacions on, en cada una, tindríem agrupats el objecte *HAB* que es l'escenari en si mateix, els portals corresponents per aquella habitació i els objectes indicatius de posicions ítems i personatges.

Textures

La texturització de tot el joc s'ha de dur a terme en fitxers del tipus **.tga*, ja que aquest format d'imatges no fa servir compressió i d'aquesta manera es dels formats més senzills d'implementar en el motor del videojoc de manera que implementi correctament la textura sobre el model 3D.

Cada textura hauria de tenir unes dimensions d'igual amplada i alçada i que aquest valor fos múltiple de 8 (8, 16, 32, 64, 128, 256...) ja que un byte esta

composat de 8 bits, i són els únics tamany acceptats per les llibreries d'*OpenGL* fetes servir.

La profunditat de color utilitzada depèn sobretot de si aquesta textura disposa de transparències en ella com una barana d'un balcó, ja que en aquest cas la textura haurà de tenir una profunditat de color de 32 bits. Si la textura no disposa de transparències una profunditat de 24 bits serà la màxima necessària per una òptima representació d'aquesta en el videojoc. Així, tenint cura d'això podem reduir una mica les dimensions en MegaBytes del videojoc. Finalment una ultima consideració a tenir en compte a l'hora de crear una textura per al videojoc es que aquesta estigui indexada amb la paleta de colors *RGB* i no amb una altra paleta com *CMYK* o qualsevol altre.

Un cop està la textura correctament creada hi ha diverses tècniques per texturitzar els models en *3D Studio Max* sent més convenient una o altre depenent de l'objecte a texturitzar.

Si per exemple es tracta d'una superfície uniforme com pot ser qualsevol dels objectes estàtics que conformen l'escenari de cada una de les habitacions del mapa, ja sigui una paret (superfície plana) com una columna (superfície cilíndrica) com una cadira (superfície cúbica), mentre estigui pensat que l'objecte estigui texturitzat de manera uniforme per a totes les cares, la manera més senzilla de texturitzar aquests objectes es mitjançant el modificador de *3D Studio* "*Mapa UVW*" que aplicarem sobre l'objecte prèviament convertit en "*Malla editable*". Aquest modificador permet indicar quin tipus de superfície s'està texturitzant (plana, cilíndrica, esfèrica, cúbica, etc) i també ens permet estirar la textura o comprimir-la fins que agafi l'aspecte desitjat mitjançant la modificació del "*Gizmo*". D'aquesta manera si tenim d'una textura de paret i la apliquem en diferents parets, per tal de que aquestes parets tinguin les mateixes proporcions de textura haurem de comprimir o estirar una de les textures de les dues parets mitjançant el "*Mapa UVW*".

Aquest és el mètode més senzill de texturitzar.

L'altre mètode de texturitzat s'emplea en objectes més complexos com un cos d'un personatge on cada part de la textura ha d'estar en un lloc concret i on no es poden fer servir 10 textures per un sol model. D'aquesta manera es comprimeix en una sola textura, del tamany necessari, totes les textures necessàries per cobrir la totalitat del model. Així, després d'aplicar aquesta textura sobre el model, aplicarem un modificador "*Desajustar mapa UVW*" i anirem seleccionant els conjunts de cares que volem que tinguin una mateixa part de la textura, seleccionarem l'eix de coordenades més convenient per al conjunt de cares de manera que al fer un pla que cobreixi totes elles, aquest abarqui la major part de superfície possible, després editarem aquestes cares seleccionades sobre la textura, modificant vèrtex a vèrtex, recol·locant-los en el punt més convenient de la textura. De manera que finalment haurem indicat a cada cara de l'objecte quina part de la textura ha de mostrar. Tot això es fa des de el modificador de "*Desajustar mapa UVW*".



Aquest mètode és el que es fa servir per a texturitzar les diferents parts del personatge, les armes i els ítems i es un pel més elaborat del que es fa servir en el mètode de "*Mapa UVW*", només utilitzat per a texturitzar les diferents superfícies del mapa.

Sons

Els sons que ajuden a oferir una major immersió i ambientació en el joc, es poden extreure per dues vies.

La primera es enregistrant-los un mateix i modificant-los posteriorment amb un programa d'edició de so com el *Sound Forge* fins a assolir el resultat desitjat.

L'altra via es aprofitar les ingents bases de dades d'efectes especials professionals que podem trobar a internet.

El format empleat per al videojoc es el *.WAV que no fa servir compressió i és el més senzill d'implementar. També accepta el format *.OGG, format lliure que comprimeix l'àudio d'una manera semblant al *.MP3.

INTERFICIE GRÀFICA

En aquesta apartat de la part multimèdia del videojoc, s'engloba la part gràfica que ens mostra informació sobre el nostre personatge o sobre events que succeeixen al mon virtual o sobre el videojoc. També ens permet la comunicació entre usuaris del joc en la mateixa partida.

S'ha procurat que la interfície sigui molt intuïtiva i que d'un sol cop de vista quedi el màxim de clar el màxim d'informació possible. S'ha intentat seguir l'estàndard fet servir en la resta de jocs amb aquest tipus d'interfície, per tal de que l'usuari no tingui problemes d'adaptació. També hem procurat que la interfície gràfica tralles el mínim de visió, del entorn 3D del joc, al usuari, fent servir les cantonades de la pantalla i moderant-se amb les dimensions.

En quant als missatges del videojoc sobre events o la comunicació entre usuaris s'ha procurat seguir les mateixes pautes que per a la interfície.

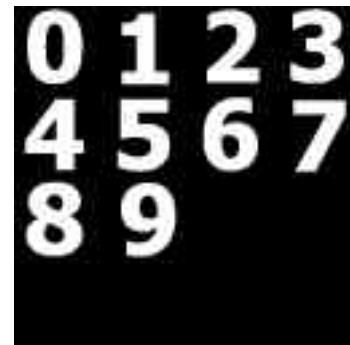
Per tal de crear interfície gràfica o les fonts fetes servir en aquesta o en la comunicació entre usuaris hem empleat TGAs de 24 bits amb dimensions iguals d'amplada i alçada i a que aquestes fossin múltiples de 8.

En el cas de interfície gràfica les dimensions escollides han estat de 256x256 pixels. Posant on corresponia cada element de la interfície i deixant aquests elements en tons blancs o grisos per tal de que posteriorment, mitjançant codi, es pugui pintar del color que es vulgui aquesta interfície. De color de fons s'ha de posar el negre, ja que el motor del joc interpreta aquest com a transparent i el blanc com a opac, així els tons intermitjos seran més translúcids com més quantitat de negre tingui el to.



Les fonts segueixen el mateix criteri de creació que interfície amb la nova pauta de que, per tal de que el motor del joc agafi correctament les fonts, l'espai entre caràcter i caràcter de la font, tant vertical com horitzontal, ha de ser el mateix.

Finalment comentar el punt de mira de l'arma que també forma part de interfície i ens permet saber en tot moment a on anirà la bala que disparem. El mètode de creació d'aquest es el mateix però s'ha assegurar de que el punt de mira estigui totalment centrat en la imatge.



MOTOR GRÀFIC

ALGORITMES DE RENDERITZAT

Les targetes acceleradores d'avui en dia són capaces de processar moltíssims triangles per segon. El problema està en que el bus AGP no pot tanta informació de cop i és forma l'anomenat "coll d'ampolla". Es justament per això que, a l'hora de pintar un món 3D, s'ha de intentar passar a la tarja el menor nombre d'informació possible, es a dir, només el que veiem. Justament per això es fan servir els algorismes de renderitzat.

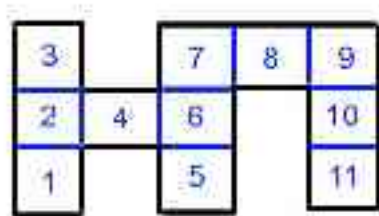
Principalment els podem dividir en dos grans grups: els algorismes per renderitzar interiors i els algorismes per renderitzar exteriors. Els primers

serveixen per pintar zones plenes d'habitacions i passadissos. El segon per pintar zones a l'aire lliure o de grans dimensions.

Donat que el nostre joc es realitza totalment en interiors o espais petits, prescindirem d'explicar el renderitzat d'exterior.

ALGORITMES DE RENDERITZAT D'INTERIORS

El primer tipus de renderitzat d'interiors que explicarem serà el *PVS (Potencial Visibility Sets)*. Aquest consisteix principalment en dividir el món en habitacions i, dins d'un fitxer, fer una taula que indiqui quines habitacions es poden veure des de cada habitació.



Dividim el mapa en habitacions

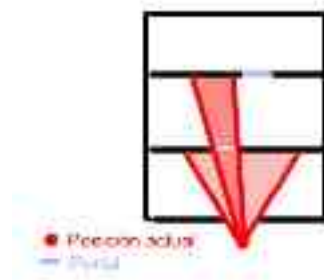


Indiquem que es pot veure des de cada habitació i pintem el que està dins de les que veiem des de on estem

El principal avantatge que té aquest sistema és la seva facilitat de realització i els seus resultats tan bons. El problema que té és que per a cada mapa que realitzes has de fer un fitxer a part, indicant que veu cada habitació.

Aquest té una segona versió que consisteix en, a més de tot l'anterior, mirar quines habitacions estan en la direcció cap on mires i pintar només aquelles.

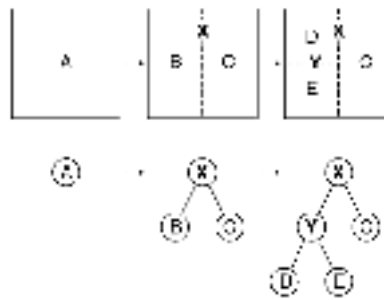
El segon tipus és l'anomenat *Portal Rendering*. Consisteix en dividir el món per habitacions (com en el cas anterior) i separar-les amb portals (petites portes invisibles). Després cal fer un con de visió entre el punt on et trobes i els màxims extrems que vegis de la habitació on estàs. Pintes tot el que veus en aquest con i comproves si hi ha algun portal dins. En cas afirmatiu, llences un altre con q passi justament pel portal fins al final de la següent habitació. Pintes tot el que hi en el segon con i mires si dins d'aquest trobes un altra portal. Així tota l'estona fins que no trobis cap portal més.



Llencem cons entre els portals fins que no en trobem cap més
i pintem tot el que està dins d'ells

Aquest mètode es molt més ràpid i eficient que l'anterior. En ell si que només pintes el que veus dins del teu con de visió. És el mètode utilitzat per la famosa saga *Unreal*.

El tercer i últim mode que explicaré són els arbres *BSP* (*Binary Space Partitioning*). Aquests particionen el mapa en parts cada vegada més petites formant un arbre en el que, de cada branca, en surten 2 més (les branques són petits espais dins de la branca de per sobre seu). A l'hora de comprovar el que has de pintar, mires si veus la branca més gran des de la vista que tens. Si la resposta es si comproves si veus alguna de les 2 branques que conté aquesta. Si la resposta torna a ser afirmativa mires les 2 que tinguin a l'interior. Així tota la estona fins que al final només pintes el que veus.

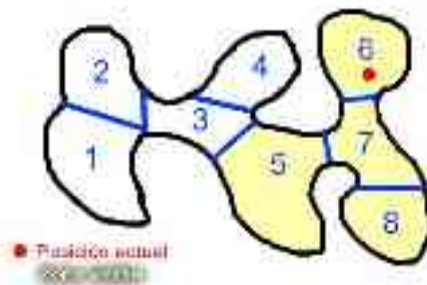


El món es divideix en parts cada cop més petites

Aquest mètode és extremadament veloç ja que si ,des de el principi, ja no veus la primera branca ja pots estar segur que mig món no el pintaràs. D'aquesta manera descartes ràpidament informació que sobre. Es molt important posar un límit a l'hora de dividir el món ja que, sinó, les branques s'estendran massa i acabarà relentitzant-ho tot. El major problema que té es la gran dificultat de creació que té (en comparació amb els altres). Es l'algoritme de representació d'interiors que fa servir la saga *Quake*.

Un cop explicats els 3 principals algorismes passaré a comentar per quin ens hem acabat decidint. Nosaltres fem servir una mescla entre *PVS* i *Portal Rendering*. Fem el fitxer amb la taula per a indicar quina habitació veu a quina com si d'un *PVS* es tractes però, a la hora de separar les habitacions, fem servir els portals del *Portal Rendering*. La raó per a fer servir aquests és per a aconseguir dividir les habitacions en parts de diferents tamanys i formes i aconseguir saber quan canvio d'habitació. Per a saber-ho detecto si col·lisio amb un portal i, en cas de fer-ho, aquest m'indica la nova habitació a la que he anat a parar. D'aquesta manera aconseguim un algorisme de renderitzat molt fàcil d'implementar, que funciona molt bé i ampliable (com ja estan creats els portals, es possible passar-lo a *Portal Rendering* en qualsevol moment).

La principal raó per la qual no hem fet servir un algorisme com *Portal Rendering* o *BSP* és que hem considerat que amb la mescla entre *PVS* i *Portal Rendering* ja servia molt bé als nostres interessos i no calia perdre temps implementant cap dels altres (sobretot el *BSP* que és molt difícil i llarg d'implementar).



Exemple de l'algorisme que fem servir (Les línies blaves indiquen els portals)

FORMA DE PINTAR

A l'hora de pintar els objectes en *OpenGL* podem fer-ho, principalment, de dues maneres: amb *glBegin* i *glEnd* o fent servir *Vertex Array*.

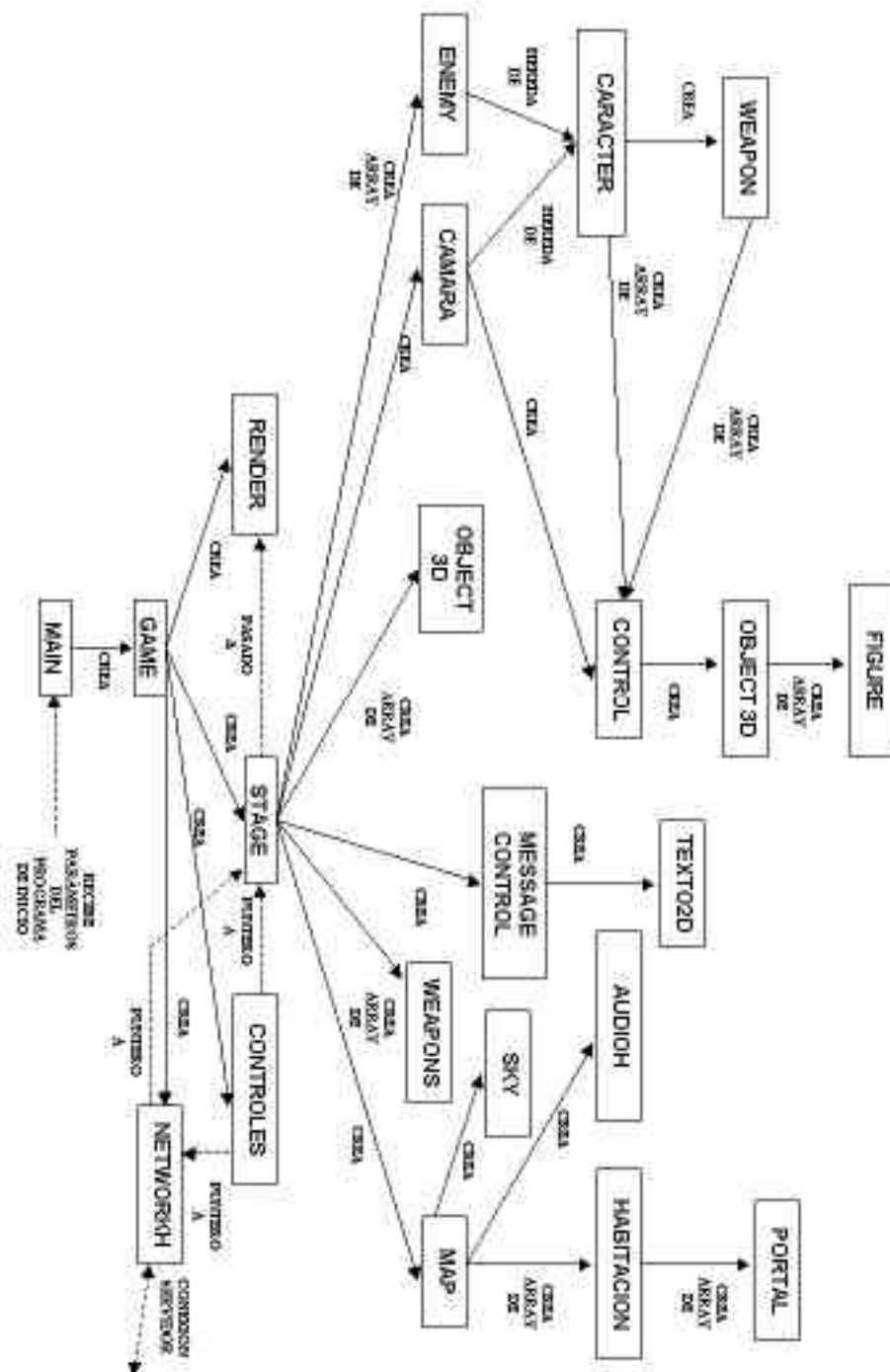
El primer mètode és el més fàcil d'implementar i serveix per pintar objectes de poques cares (ja que no és tan ràpid com el *Vertex Array*). *OpenGL* és una màquina d'estats i, per tant, indiques amb *glBegin* i el mètode de dibuix (triangles, línies, punts, etc.) que a partir d'aquest moment (fins que no trobi un *glEnd*) ho pinti tot tal i com l'hi has dit.

El segon mètode (*Vertex Array*) és un pel més difícil d'implementar, no per la dificultat de utilització, sinó per que estàs obligat a tenir ordenats per cares tots els vèrtex en un sol array. Aquest es el mètode més professional per pintar. Envia les dades a gran velocitat pel bus AGP de manera que els *FPS* (*Frames per Segon*) augmenten considerablement.

Nosaltres pintem els elements animats (items, enemics, etc.) amb *Vertex Array* ja que solen tenir molt poques cares (encara que tenim ordenats els seus vèrtex en un array per a poder pintar-los amb *Vertex Array* quan vulguis). L'escenari (que sol tenir bastants mes cares) el pintem amb *Vertex Array*.

CLASSES

En aquest apartat passarem a explicar, per sobre, quina és la funció de cada classe del motor gràfic.



GAME:

Es la classe que s'encarrega de crear tot el joc. Dins seu es crea el control de teclat, xarxa, renderitzat i s'emmagatzema tota la fase.

RENDER:

Es la encarregada de crear la finestra i pintar per pantalla. L'hi passes una fase i ell pinta tot el que el PVS li diu.

CONTROLES:

S'encarrega de capturar els events de teclat i mouse.

NETWORKH:

Fa la connexió amb el servidor. Es la encarregada d'enviar dades i rebre-les

STAGE:

Probablement es la classe més important del joc. S'encarrega de controlar tota la fase (els items, el PVS, les armes, els enemics, etc). En ella s'emmagatzemen tots els elements del nostre món.

WEAPONS:

Com el seu nom indica, crea una arma. Dins seu s'emmagatzemen els 2 models de la arma (el que es fa servir com a item i es col·loca a les mans del personatge i el que es fa servir en primera persona).

MESSAGECONTROL:

Gestiona els missatges de text que es pinten per pantalla. Els crea i elimina quan fa falta.

TEXT2D:

Emmagatzema la font i escriu per pantalla el que volguem, a la posició que volguem.

MAP:

Emmagatzema tota la informació 3D de la fase que no és mòbil (les habitacions, els portals, el cel, etc.).

AUDIOH:

S'encarrega de emmagatzemar l'àudio (el só de les passes, l'ambient, etc) i fer-lo sonar.

HABITACION:

Emmagatzema tota la informació estàtica d'una habitació (portals, vèrtex, textures, etc.).

PORTAL:

Emmagatzema la informació d'un portal (els seus vèrtex, a on apunta, etc.).

SKY:

No es una classe, es una estructura, i és la encarregada de guardar el cel.

CAMARA:

Hereta de caracter i guarda la informació sobre el teu personatge.

ENEMY:

Hereta de caracter i guarda la informació sobre els enemics. En aquest moment no s'utilitza per a res però s'espera que en un futur sí.

CHARACTER:

Guarda la informació 3D d'un personatge (cap, cos, cames), la seva arma i el seu Nick.

CONTROL:

Conté la posició, la rotació, la velocitat, etc. d'un objecte 3D mòbil o animat. S'encarrega de gestionar alguns efectes com la desintegració o explosió de l'element i també del canvi d'animació o del moviment d'aquestes.

OBJECT3D:

Guarda la informació 3D d'un element mòbil o animat i conté els seus figures (frames).

FIGURE:

Guarda els vèrtex d'un frame de la animació.

FUNCIONAMENT GENERAL

El primer que fa el joc és llegir els fitxers de configuració d'àudio. Tot seguit, engega el client de xarxa i es connecta amb el servidor. El joc està en espera fins que aquest ha comprovat que tens tots els fitxers per a poder començar a jugar. Quan ja els tens, es crea la finestra, s'inicialitza *OpenGL*, es creen els controls i comença a carregar-se la fase. Un cop carregada, envia un missatge al servidor avisant-li que ja estàs dins del joc i que ja pot començar a enviar paquets.

Passada aquesta primera etapa, el joc entra en un bucle infinit. Primer actualitza tots els objectes del món (animacions, etc.). El segon pas es, si cal, enviar la teva posició al servidor. El tercer es pintar el món. El quart i últim es comprovar el teclat i el mouse.

Tot això es repeteix infinitament juntament amb un fil en paral·lel que no para d'escoltar al servidor en el port que te assignat i de passar les dades necessàries al món.

Per a aconseguir la independència de velocitat en les animacions respecte als *FPS* (*Frames Per Segon*), el que fem es canviar de frame segons un temporitzador.

Per a resoldre el mateix problema amb el nostre moviment (que no ens movem més o menys ràpid depenent dels *FPS*), calculem el temps transcorregut entre l'últim frame i l'actual i multipliquem la velocitat del personatge per aquest offset. Això provoca que si vas a molts *FPS*, et belluguis moltes vegades però amb passes molt petites i, en el cas contrari, et belluguis poques vegades però amb passes molt grans.

ORGANITZACIÓ DEL CODI

El codi està organitzat de manera que sigui el màxim d'optimitzat possible. A l'hora de carregar objectes ,com ara items, primer carreguem tots els models en memòria i llavors la resta només són punters a aquests ja creats. El mateix succeeix amb les textures i el só.

A l'hora de tractar les armes fem servir un mètode bastant similar. Les guardem totes en memòria i després l'hi assignem a cada personatge/enemic una "copia" d'aquesta. Realment només es copien valors com tipus d'arma, munició, posició (d'aquesta manera podem canviar la arma de cada personatge de posició en tot moment), etc. Tot el que es informació voluminosa i independent de qui la porta, com ara el model de l'arma o la seva textura, es assignada mitjançant un punter.

Amb tot això es complica una mica més el codi (hi ha punters amunt i avall) però aconseguim carregar només una vegada els objectes repetits i disminuir notablement l'ús de memòria.

El codi també està preparat per, amb uns mínims canvis, permeti tenir diferents móns oberts i anar canviant entre ells quan volguem.

Els personatges estan dividits en 3 parts (cap, cos i cames). D'aquesta manera no cal que tinguem una animació per a cada possibilitat d combinació entre córrer, disparar, etc. directament podem dir-li que les cames corrin i el cos camini. Això ajuda a que el joc els personatges ocupin menys espai en memòria i sigui més fàcil controlar les seves animacions.

La següent raó per a fer servir aquest mètode es que (en un futur) quan un enemic miri cap amunt pots fer que el cos del seu personatge també ho faci (però deixant les cames quietes), que aquest pugui girar el cap, etc. Aquest mètode s'assembla bastant al que fa servir el *Quake 3* en els seus md3 (és el nom dels fitxers que contenen els personatges). No fa molt (en la epoca del

La tercera i última raó per a emmagatzemar personatges així es q es molt més fàcil poder aconseguir que facis més o menys mal depenent de a quina part del cos disparis.



38

El joc s'adequa a l'hora actual i, per tant, es va modificant automàticament l'escenari. Amb això volem dir que si són les 3 de la tarda, l'escenari es veurà amb un fons diürn. En canvi, si es plena nit, el joc es veurà amb un fons fosc. La transició no és immediata. Poc a poc, segons la hora, es va enfosquint o aclarint com si de la realitat és tractés.



Dia



Nit

XARXA

La implementació d'un sistema de comunicació, autenticació i sincronisme de dades presenta bastants complicacions dins de l'estructura de funcionament del programa.

El principal objectiu del projecte és el d'escriure un codi el suficientment estandar, per tant de fer-lo portable entre diferents sistemes i arquitectures, per tant qualsevol codi, funció o ordre que sigui específic d'un sistema (signals, windows API,...) caldrà que la marquem dins d'un bloc `"#if WIN32"` i aportar una solució amb `"#else"` per a l'altre sistema.

El funcionament del networkh es basa en un sistema client/servidor, lògicament el servidor reunirà totes les connexions dels clients, el principal problema al treballar amb *sockets* (connexions de xarxa), és que al ser més d'un cal realitzar una multiplexació, amb sistemes **nix* podem usar *select()* o *poll()* junt amb signals, pero una implementació molt més potent i portable és utilitzant fils d'execució, on cada fil controla una connexió i es comunica amb la resta a través de bloquejos *mutex* d'estructures.

Un altre punt que és important sobre el funcionament del networkh és el sistema de dependències, ja que ha de ser un sistema autònom i que sàpiga amollar-se a la situació de tots els clients en cada moment per tal de mantenir totes les dependències de fitxers iguals per a tots els clients, ja que és el servidor l'encarregat d'assegurar-se que tots els clients tenen tots els fitxers que necessiten (models, mapes, fitxes de rol, textures, sons, items,...).

Els clients, han de ser simples ninots, tant se val que siguin segurs, ja que qui controla la seguretat ha de ser el servidor, els clients són simples usuaris que mantenen una conversació amb el servidor. Per tal de mantenir una política de seguretats, cal realitzar unes especificacions de protocol prou clares per tal que cap client pugui enganyar al servidor o a altres clients i fer-los creure de

coses que no poden ser possibles (moviments bruscos de posició, canvis a armes no existents, etc...).

Per tant el servidor ha de comprovar la integritat de tots els paquets que es mouen per la xarxa gH, per evitar enganys i accions no-legals en el sistema.

Tanta complexitat en intentar mantenir una bona política de seguretat implica carregar el servidor del joc amb molt codi i moltes dades, tant dels usuaris, com de les armes, dels items, del mapa, etc. I comprovar-les totes en tots els moments, això ens porta a l'altre gran punt de la implementació del sistema: les col·lisions i els impactes.

Tractant els temes per sobre, podem parlar de dos tipus de col·lisions:

- amb els items.
- amb el món.

Les primeres són les més senzilles ja que es calculen amb un simple '*bounding*', és a dir, quan la posició de l'usuari esta a una distància "*bound*" de l'item, el servidor actua com si el usuari hagués tocat el item.

Les segones són bastant més complicades, ja que impliquen molts més càlculs entre l'array de vèrtex de la malla de l'usuari amb l'array de vèrtex del món o de l'habitació en que es trobi el usuari.

Els impactes són bastant més complicats ja que permeten accedir fora d'habitacions, i per tant cal calcular si travessen portals, col·lisionen amb parets o clients, i a quines distàncies estan els elements col·lisionats agafant el més proper com a element impactat.

El funcionament de la xarxa és la base del joc en si, ja que és la que junta la resta de parts i les fa funcionar conjuntament de manera comuna. És per això cal prendre bé les decisions d'aquest àmbit ja que sobre aquestes dependrà la resta del funcionament del joc, el rendiment i les possibilitats d'ampliació.

A nivell de protocols podem parlar de dos tipus de protocols, el de transport, de capa 4 i el de nivell d'aplicació, dins de la capa 7 del model OSI.

El protocol de capa 4 és el de transport de dades, dins d'aquest ens trobem davant d'uns quants diferents: *UDP*, *TCP* i *IPX*, lògicament el 3r no és compatible amb els primers ja que usa un sistema de redireccionament diferent (protocol *ipx*), en canvi *udp* i *tcp* fan servir el protocol IP per direccionar connexions.

IPX era un protocol molt utilitzat en jocs antics, pero actualment, i gràcies a internet s'han anat adaptant a protocols com *TCP* i *UDP* que funcionen sobre IP.

L'elecció entre *TCP* i *UDP* és força complicada, ja que *UDP* permet enviar dades sense realitzar una connexió prèvia i amb conseqüència sense assegurar que aquests han arribat. Això implica que les dades que s'enviïn d'aquesta manera haurem de tenir clar que no ens importarà que aquestes dades es perdin i per tant que si no passarà res si arriben o no. I a més existeix la dificultat afegida que es poden modificar paquets per enganyar al sistema i fer-se passar per un altre usuari de la xarxa, i per tant enganyar al servidor. Cosa no gaire agradable per la seguretat del sistema.

Normalment el que es fa servir és barrejar els dos protocols per optimitzar al màxim el funcionament de la xarxa. És a dir, els paquets importants (autenticació, Join, dependències, etc.) s'envien utilitzant *TCP* i els no importants i massius (moviment) s'envien per *UDP*.

La importància del *UDP* és que té una capçalera molt més petita que la del *TCP* i per tant envia menys dades per xarxa.

En el model actual no hem contemplat aquesta opció, ja que complicava bastant tot el funcionament general del sistema de xarxa. I en una xarxa local

tenint 10 o 100Mbps. Realment no cal utilitzar *UDP* per enviar les dades ja que implicaria una gran complicació de codi, una reducció del rendiment en el processament de paquets realment innecessària per les proves que hem realitzat de rendiment.

Tecnologies emprades

En aquest capítol explicaré les diferents tecnologies utilitzades durant el desenvolupament i quins han sigut els motius d'utilitzar-los, i com hem estructurat el seu funcionament del mòdul del networkh del gIH.

Com ja he comentat, la part de xarxa es divideix en dues parts: client i servidor.

Client:

La funció del client de xarxa en el gIH és la de connectar-se al servidor gIH i mantenir un diàleg a través de paquets (l'especificació del protocol l'explicaré més endavant).

Un cop connectat s'estableix una comunicació bidireccional entre el servidor i el client per tal d'autenticar-se contra el servidor, un cop autenticat el client anirà rebent tots els fitxers de dependències que necessitarà per carregar el mapa, els items, models, textures, sons, etc.

Un cop tingui totes les dependències que necessiti, el client rebrà un paquet del tipus "*PACKET_OKOK*" que indica al client que el servidor està a l'espera de rebre un paquet *OKOK* enviat des de el client.

Quan el servidor el rep, llavors li enviarà el seu *BeginPoint* i tots els *Join* de la resta d'usuaris, per tal que el motor els situï en el lloc adient i entra en el bucle principal de funcionament del joc, és a dir, a partir d'aquí és quan es començaran a enviar-se tots els paquets de moviment.

Quan s'afegeixi un nou usuari s'entrarà en un bucle per actualitzar totes les dependències entre tots els usuaris i un cop fet això enviar el paquet "*PACKET_JOIN*" en broadcast per informar que ha entrat aquest nou usuari. El client haurà de gestionar totes les dades del nou usuari afegit, situant-lo en l'espai i movent-lo junt amb els canvis d'animacions.

El client pot sortir de la partida quan vulgui o fins que el servidor el faci fora, el servidor ja controla quan un usuari tanca la connexió i ho notifica a la resta de clients i allibera les estructures de dades que identifiquen al client.

Tot el tema de dades del personatge les controla el servidor, de manera que el client simplement serà un mirall de les dades del servidor i no al revés, així s'evita que els clients puguin enganyar al servidor i a la resta de clients.

El mateix succeeix amb els moviments, i la resta de paquets que han estat dissenyats per evitar que el client pugui "enganyar". Tots aquests punts els tocaré amb més detall dins de la definició del protocol.

Al evadir al client del màxim de operacions ens permet que augmentin els fps (ja que la càrrega que suposa els càlculs geomètrics ho fa el servidor) i deixa lliures els clients perquè només facin comunicació per xarxa i generar l'escena amb OpenGL.

Server:

La funció del servidor és la de guardar tots els arrays de vèrtex, i gestionar totes les connexions dels clients, i realitzar tots els càlculs de col·lisions. I impactes de bales i col·lisions amb items.

El primer que fa es carregar el fitxer de configuració "*config.svr*" en aquest es defineixen opcions com tamany del buffer de transferència de dades, el port a escoltar (per defecte el 9999), el nombre màxim de connexions, la màxima

longitud del nick, el mapa a utilitzar, el tipus de LOGIN, el nivell de Verbose, que serveix per fer que doni més o menys missatges alhora de funcionar, molt útil per depurar el funcionament.

De tipus de login es permeten 3, de moment només hi han 2 implementats:

- anonymous

Aquest mode d'autenticació no comprova la contrasenya i dona a l'usuari una fitxa de personatge estandar, de manera que tothom està en igualtat de condicions i per tant no cal donar d'alta el usuari per començar a jugar, ja que tots els noms d'usuaris estan permesos.

- custom

Aquest mode permet que el servidor defineixi en un fitxer de contrasenyes (*config.pwd*) el login, password i fitxa de personatge relacionada amb aquests. Així podem tenir un llistat d'usuaris que es validin contra el servidor amb un login i contrasenyes propis guardant una fitxa de rol pròpia per a cada personatge.

- official

Aquest últim mode és el que permetrà validar els usuaris amb un servidor oficial, aquest servidor oficial tindrà un servidor web on es podran crear les fitxes de personatges a través de formularis i et generarà la nova fitxa de personatge, llavors el servidor glH quan rebi una petició de connexió de client, aquest es connectarà cap al servidor oficial i comprovarà que la contrasenya sigui la correcte (seguint el mateix funcionament d'autenticació de *crc* del sistema "custom").

Amb la gràcia que aquest sistema funciona de forma centralitzada contra un sol servidor, i permet tenir una fitxa única i poder canviar d'ordinador i seguir-la mantenint, d'aquesta manera es poden crear competicions i estadístiques de partides controlat en un servidor *pgSQL*.

Especificacions del protocol

El protocol glH de xarxa ha estat dissenyat des de zero agafant idees del funcionament d'altres protocols de xarxa per jocs de tipus '*shoot'm'up*', i els conceptes generals, arreglant alguns errors de protocols antics i adaptant-lo a les necessitats, en si el protocol ha estat fet des de zero i no té cap semblança amb cap altre protocol de xarxa per jocs.

El funcionament dels paquets és força simple, el primer byte indica el tipus de paquet i la resta de bytes que el segueixen són el contingut del paquet, que pot variar des de 0 fins a N bytes. Depenent del tipus de paquet i dels continguts d'aquest.

Un llistat dels tipus de paquets existents es pot trobar en el fitxer "*packet.h*" dins del directori "*common*/" de les fonts del servidor de la xarxa del joc.

```
enum {  
    PACKET_PING=0, // [ ] send a ping  
    PACKET_PONG, // [ ] reply a ping  
    PACKET_USER, // [x] Send username  
    PACKET_PKEY, // [x] Server sends the password key  
    PACKET_PASS, // [x] Send password  
    PACKET_ERRR, // [x] Error ;)  
    PACKET_VERS, // [x] Send server Version  
    PACKET_CHKf, // [x] Check file  
    PACKET_CRCf, // [x] Reply a crc32 of the file requested.  
    PACKET_FILE, // [x] Send a file to the client.  
    PACKET_NICK, // [x] Change my nick (n,"nick");  
    PACKET_TALK, // [x] Talk message :D  
    PACKET_MOVE, // [x] Move message...100% done  
    PACKET_PART, // [x] Send a broadcast message  
    PACKET_JOIN, // [x] JoinMessage  
    PACKET_ITEM, // [x] Item packet
```

PACKET_BEGN, // [x] Send the BeginPoint
PACKET_WEAP, // [x] Change to Weapon [n]
PACKET_CHWP, // [x] Change next weapon
PACKET_OKOK // [x] Ready to play?

Els quadres marcats amb [x] indiquen que ja han estat implementats en el client i el servidor i per tant ja són funcionals. Ara aniré definint paquet a paquet, les funcions que realitza cadascun, el cos del paquet, etc.

PACKET_USER

[1byte:strlen(nickname)][Nbytes:nickname]

Aquest paquet serveix per enviar el nom d'usuari a validar contra el servidor, durant una connexió només es pot enviar un sol cop aquest paquet, ja que si es torna a enviar el servidor ho conta com a un Relogin, cosa que no està ni permesa ni implementada per tal de simplificar el protocol.

PACKET_PKEY

[4bytes:rand()%256]

Aquest paquet l'envia el servidor cap al client un cop ha rebut el paquet PACKET_USER, per tal de donar una clau aleatòria de 4 bytes amb la que encriptar el crc32 del password i augmentar així la dificultat de capturar contrasenyes amb sniffers.

PACKET_PASS

[4bytes:password-key]

Aquest paquet al igual que el PACKET_USER només es pot enviar un sol cop per cada connexió. Aquest paquet és enviat pel client cap al servidor, i és el resultat de realitzar un xor entre la contrasenya en text pla sobre la clau de 4 bytes aleatòria que ha enviat el servidor amb el paquet PKEY i després realitzar un crc32 sobre la string resultant.

PACKET_ERRR

[1byte:strlen(errmessage)][Nbytes:errmessage]

El paquet ERRR serveix per enviar un missatge d'error al client.

PACKET_VERS

[1byte:version][1byte:strlen>HelloMsg)][Nbytes>HelloMsg]

Aquest és el primer paquet que s'envia un cop realitzada la connexió, en aquest paquet s'envia el número de versió del protocol i un missatge de benvinguda a la connexió. El client caldrà que comprovi si el número de versió equival al número de versió del protocol escrita en el client, si són diferents, el client pot intentar connectar-se però el més segur és que quan rebi un paquet no reconegut el servidor li tanqui la connexió de cop amb un missatge de *"Unexpected packet recived"*.

PACKET_CHKF

[1byte:strlen(filename)][Nbytes:filename]

Aquest paquet és enviat pel servidor per indicar al client que ha de comprovar l'existència del fitxer (si no existeix envia un *crc*=[0:0:0:0] i enviar el crc32 del fitxer. Llavors el servidor comprovarà que el *crc32* sigui correcte, si ho és li

enviarà el paquet CHKF del següent fitxer a comprovar i sinó doncs envia un paquet del tipus PACKET_FILE amb el fitxer adjunt.

PACKET_CRCF

[4byte:crc32]

Aquest paquet és el que respon el client cap al servidor al paquet CHKF, aquest conté el *crc32* del fitxer indicat pel paquet CHKF. El *crc32* pot ser o 0:0:0:0 (el fitxer no existeix o el *crc32* correcte del fitxer. El servidor li enviarà el paquet amb el contingut del fitxer o el següent fitxer a comprovar si resulta que el *crc* és el correcte.

PACKET_FILE

[1byte:strlen(filename)][Nbytes:filename]

[4byte:filesize][Nbytes:FileContents]

El paquet FILE serveix per enviar fitxers, envia primer el nom de l'arxiu, després el tamany de l'arxiu i a continuació el contingut de l'arxiu. El contingut de l'arxiu l'envia en paquets petits de tamany BUFFER (definit en el fitxer de configuració tant del client i del servidor), No cal que siguin el mateix, ja que els tamanyes es van agafant de manera asíncrona. El enviar-ho en forma de paquets de tamany petit és per dos motius, primer per no saturar la xarxa amb paquets molt grans que si arriben defectuosos caldrà tornar a enviar-los i després perquè si la funció `send()` no permet enviar paquets majors de 65KB.

PACKET_NICK

[1byte:strlen(nick)][Nbytes:nick]

Aquest paquet l'envia el client per canviar de nick en mig de la partida, en

principi aquest paquet no és recomanable utilitzar-lo ja que de vegades és molest veure com la gent es va canviant de nick i pot causar confusions entre els usuaris. Per això hi ha una directiva en el fitxer de configuració per permet habilitar o deshabitar aquesta funció.

PACKET_PART

[1byte:who]

Indica que el usuari amb el ID número "*who*" ha sortit del joc.

PACKET_MOVE

[?1byte:who] - només existeix de servidor a client.

[4bytes:pos_x][4bytes:pos_y][4bytes:pos_z]

[1byte:(int)rot/2][1byte:(>>)ani]

El paquet MOVE és el que serveix per enviar un paquet de moviment cap al servidor,

PACKET_TALK

[1byte:strlen(msg)][Nbytes:msg]

Serveix per enviar paquets de missatges, els típics "*Talk*" entre els usuaris d'un joc. El client només ha d'enviar el missatge i el servidor el reenviarà a tots els usuaris en broadcast adjuntant al missatge el nom de l'usuari i ": ".

PACKET_JOIN

[1byte:ID]

[1byte:strlen(nick)][Nbytes:nick]

[1byte:strlen(rph)] [Nbytes:rph]ph]

El paquet JOIN serveix per informar als usuaris que ha entrat un nou usuari en el joc, aquest paquet només s'envia quan el usuari ja té totes les dependències, llavors quan el client rep aquest paquet cal analitzar-lo i guardar-ho dins d'una array de Personatges on guardarà el ID el nick i l'RPH, a través de l'RPH el client ja sap quin és el model que necessita i fins hi tot les característiques del personatge.

A més aquest paquet s'utilitza per situar-se un mateix dins de l'array de personatges, comprovant el nick amb el seu i associant-lo al ID.

PACKET_ITEM

[1byte:DO][1byte:ACTION]

El paquet ITEM serveix per indicar quan s'ha col·lisionat amb un ítem, cada ítem té un valor DO i un valor ACTION. El DO defineix l'acció que fa i el ACTION com la fa o en quin grau la fa. En el fitxer "../common/items.h" hi ha definides les accions que poden realitzar els ítems, els més importants de cara al funcionament del client son els dos primers: "ITEM_HIDE" i "ITEM_SHOW" respectivament amb valors 0 i 1. I com a paràmetre ACTION se li passa el número d'ítem. Així el client sabrà que ha de fer aparèixer o desaparèixer aquell ítem.

PACKET_BEGN

[4bytes:pos_x][4bytes:pos_y][4bytes:pos_z]
[1byte:rot/2][1byte:hab]

Aquest paquet serveix per indicar a l'usuari que ha començat el joc i que ell està situat en un BeginPoint (un punt de començament dins del mapa), el servidor té un array de BeginPoints i quan l'ha de llençar es queda en un bucle agafant posicions aleatòries i comprovant que no hi hagi ningú dins d'un bounding al BeginPoint.

Quan troba un BeginPoint lliure li envia les dades, de posició X,Y,Z junt amb l'angle de rotació (al ser un byte dividim entre dos l'angle, per tal d'enviar menys dades per xarxa) i un altre byte definint en quina habitació es troba.

PACKET_WEAP

[?1byte:who] - Quan l'envia el servidor.

[1byte:Nweapon]

[?1byte:munition] - Quan l'envia el servidor.

Aquest paquet és un bon exemple del reciclatge de paquets que realitza el protocol glH, Té dues versions, la del client que només conté 1 byte de cos indicant a quina arma vol canviar i el que envia el servidor que conte 3 bytes de cos, un indicant a quin usuari canvia d'arma, el segon byte per indicar a quina arma canvia i el tercer byte indicant quanta munició té.

PACKET_CHWP

El paquet ChWp serveix per canviar a la següent arma, llavors el servidor mirarà a les estructures d'armes del client per veure les armes que té i amb quines municions, i el servidor li contestarà enviant-li un paquet PACKET_WEAP amb les dades corresponents, per lo contrari, no li enviarà cap paquet.

PACKET_OKOK

El paquet OkOk serveix per realitzar marques d'espera, quan el servidor envia un OKOK, la comunicació amb aquell usuari s'aturarà fins que el client li contesti un altre paquet de tipus OKOK, d'aquesta manera podem sincronitzar el client amb el servidor, ja que per exemple, un cop baixats totes les dependències, el client les haurà de carregar, d'aquesta manera el client quan ho tingui tot carregat li enviarà un paquet OKOK al servidor perquè aquest li enviï el BeginPoint i comenci el bucle de joc.

FITXERS DE CONFIGURACIÓ

Tant el client com el servidor es poden configurar a través de fitxers de configuració, els fitxers de configuració tenen el nom "*config.xxx*" on "*xxx*" indica de què és el fitxer de configuració:

svr - configuració de xarxa del servidor

cli - configuració de xarxa del client

vid - configuració de video del client

kbd - configuració de teclat del client

snd - configuració de so del client

config.svr

Aquest és un exemple de configuració del servidor:

```
# Defineix el port TCP a escoltar
```

```
PORT 9999
```

```
# Definim els límits
```

MAXCONN 11

MAXNICK 8

Definim el mapa

MAP castle.mph

Desactivem la funció de canviar nick

CHANGENICK 0

Nivell de Verbose

VERBOSE 0

Tamany del buffer de transferència (10KB)

BUFFER 10240

Definim un missatge de benvinguda personalitzat

HELLOMSG Benvingut al servidor glH :)

Ratio de proporció de vertex a món real.

- Es sol definir en el mapa, però per si de cas també

el podem definir aquí si el mapa no ho té marcat

RATIO 0.001

Definim el tipus d'autenticació:

#

anònima:

LOGIN anonymous(default.rph)

#

oficial

LOGIN official(glh.sf.net)

config.cli

Configuració de connexió

SERVER localhost

PORT 9999

Tamany del buffer de transferència de fitxers

BUFFER 10240

Autenticació (no és necessari guardar-ho)

LOGIN pancake

PASSWORD pop

config.snd

Habilitar el so?

ENABLE 1

Volum general del joc?

VOLUME 80

config.vid

Profunditat de color

BPP 32

Resolució

RES 1024x768

Texturització

TEXT 1

config.kbd

Definició de totes les tecles

WALK_F UP

WALK_B DOWN

TURN_L LEFT

TURN_R RIGHT

STRAFE_L Q

STRAFE_R W

FIRE CONTROL

JUMP SPACE

CHWEAP RETURN

TALK T

CENTERV C

COL·LISIONS

Les col·lisions són potser el punt més complexa de tot el projecte en sí, i potser la que ens hauria costat més de realitzar des de zero, no només per la quantitat de temps que necessitaríem per poder dissenyar, programar i provar un bon sistema de col·lisions, sinó pel pocs coneixements que teníem en un bon principi, i vista la gran complexitat del projecte general vam optar per utilitzar unes llibreries lliures anomenades *OpenDoor* realitzades per un estudiant de La Salle Enginyeria.

El utilitzar aquestes llibreries ens ha permès veure d'una manera molt més general i en poc temps el funcionament de diferents tipus de sistemes de col·lisions, friccions, rebots i una base del funcionament de sistemes de Intel·ligència Artificial.

Bàsicament aquestes llibreries simplifiquen en gran mesura la manera d'implementar un sistema de col·lisions en un simulador d'espais virtuals, a més gràcies a la flexibilitat d'us permet crear diferents elements amb característiques de col·lisions diferents i veure com interactuen entre ells.

Una visió global del funcionament seria la de d'inicialitzar un món amb un identificador, i dins d'aquest anar creant submons on se li passen les mides de les habitacions i els arrays de vèrtex de cadascuna.

D'aquesta manera el sistema internament ja pot fer neteja d'habitacions i saber en quina et trobes exactament i llavors calcular les col·lisions del personatge dins d'aquests "submons" o "habitacions".

Un cop creat el món i els seus submons, caldrà crear les entitats que treballaran alla dins, al crear-les se lis dona una posició X,Y,Z, i se li crea un objecte, es a dir, tant es pot crear com una esfera, un cilindre, o un conjunt de cares i vèrtex.

El sistema que hem utilitzat és el segon, en el que els personatges son esferes, ja que d'aquesta manera és una manera molt simple de realitzar col·lisions entre personatges i el món. I d'aquesta manera es poden realitzar col·lisions i friccions de manera senzilla i ràpida. Cosa molt important en el nostre projecte ja que com que totes les col·lisions les controla el servidor és important que el funcionament d'aquests càlculs sigui prou òptim i ràpid per tal de no relentitzar el funcionament del joc.

Un cop arribats a aquest punt podem fer un esquema de funcionament de moviments del joc:

- 1- El servidor envia un BeginPoint donant la posició inicial del personatge.
- 2- El client envia la seva nova posició cada cop que es mou.
- 3- El servidor fa un vector de moviment entre la posició anterior i la nova posició
- 4- Escurça el vector en relació a la característica de velocitat del personatge i en relació al nombre de paquets de moviment que rebi per segon. Per tal d'evitar que el personatge pugui corre més del compte.
- 5- El servidor resta sobre la component Y del vector moviment un valor

"gravetat", per tal de mantenir un sistema de gravetat estable. Les llibreries ja s'encarregaran de calcular les friccions i donar la nova posició.

6- Llavors el servidor crea el vector resultant, restant la nova posició a la antiga i li passa a les llibreries *OpenDoor*. Que ens retornaran la nova posició on es trobarà el personatge (calculades les friccions, Etc).

7- Envia un paquet en broadcast cap a tots els clients indicant la nova posició del personatge i guardant aquestes dades en l'estructura de dades del servidor.

El tema dels impactes l'hem enfocat d'una manera diferent. Quan es realitza un tret, es crea una nova entitat amb posició x,y,z origen i el farà moure cap a una direcció destí.

Aquestes llibreries al realitzar un moviment d'una entitat dins d'un entorn et retornen a quin lloc exacte han col·lisionat, és a dir, posició x,y,z de la col·lisió, número de cara de l'habitació, etc. Això ens permet treballar enllaçant les colisions d'impactes a través del mòdul de so (audioh), i fer sonar un so de tret quan la bala toca la paret per exemple, i fer-lo sonar en aquella posició exacte.

Però els impactes amb altres personatges els tenim plantejats diferent, primer colisiona amb el personatge i després a partir d'aquí tenim la posició x,y,z d'on toca, agafant la alçada del terra podem saber a quina part del cos toca, ja que el servidor a l'hora de carregar el .CAR emmagatzema les dades de peus, genolls, cadera, coll i cap. I llavors treballar a nivell de rol per tal de agafar els modificador d'armadura, defensa i demás i treure el dany resultant.

I per acabar queden les col·lisions amb els ítems que son comprovacions de col·lisions bàsiques a nivell d'esfera, sense comprovar friccions, simplement quan l'usuari s'apropa a una distància menor D el servidor contarà que aquell usuari ha agafat el ítem i li enviarà les dades corresponents i farà amagar el ítem durant X segons i després el farà tornar a aparèixer.

ROL

Per darrera de tota la base gràfica i tècnica hi ha una lògica d'accions basada en fitxes de rol. Cada personatge quan entra al joc se li assigna un fitxer ".rph". En aquests fitxers es defineixen totes les característiques del personatge.

El fitxer rph es troba en el directori "data/role", aquí és on es defineixen les directives del personatge, els punts de vida, els punts de defensa o constitució, la velocitat de moviment, la força i la punteria. I per una altre banda tenim els punts d'experiència i el nivell del personatge.

Tots aquests valors són els típics de la majoria de jocs de rol. D'aquesta manera ens permet crear un personatge no tant sols personalitzat per un model específic sino per un conjunt de característiques que el defineixen d'una manera més personalitzada.

Poques vegades s'acostumen a barrejar temes de rol en jocs *shoot'm'up*, però hem trobat interessat ajuntar-ho ja que ens permet, d'una manera relativament simple dotar de gran vida al projecte, podent crear un personatge al teu gust i poc a poc anar-lo evolucionant combat rera combat. Guanyant experiència i apujant nivells.

Quan es puja un nivell d'experiència, al final de cada partida, se li donaran uns punts a repartir entre les seves característiques, i els punts d'experiència baixaran a 0.

La dificultat d'aconseguir més punts d'experiència és a nivell exponencial, és a dir, la dificultat de pujar de nivell 8 a 9 és exponencialment molt més difícil d'aconseguir que apujar del nivell 1 al 2.

Així mateix totes les característiques del personatge es podran visualitzar en el client, tant la teva fitxa de personatge com la de la resta de jugadors.

Però per motius de seguretat serà el servidor l'únic que podrà modificar totes aquestes dades, i utilitzar-les. Per exemple la capacitat de AIM (punteria) servirà per realitzar un desajust aleatori en relació a la quantitat de punteria que tingui. O la velocitat servirà com a punt de referència per evitar que un usuari no es mogui massa distància en poc temps (diferència en el vector de moviment).

A nivell de servidor aquestes característiques són constants durant el transcurs del joc, ja que no variaran, el cert és que si que variaran, però no elles directament sinó uns modificadors (mod[]), d'aquesta manera es mantenen els valors originals i es pot jugar amb els seus valors de manera independent.

Anem a veure una per una que fa cada característiques:

STRENGTH

Aquesta característica s'utilitza com a modificador de dany en armes cos a cos i en certes armes pesades per saber si el personatge les pot portar o no. Tots aquests càlculs i comprovacions les realitza el servidor. Per tal d'evitar que el client pugui enganyar al sistema.

SPEED

Aquesta característica s'utilitza per a comprovar la velocitat de moviment del personatge i controlar que no sobrepassi el seu límit "físic". És a dir, a nivell de protocol, quan un usuari es mou, envia cap al servidor un paquet amb la x,y i z noves. Llavors el servidor calcula el vector de moviment entre la posició anterior i la nova i calcula el seu mòdul, i en relació al "LAG" del client, el servidor comprova que l'usuari no hagi realitzat un moviment massa llarg. I si és així que escurci el tamany del vector per evitar que es mogui massa.

DEFENSE

Aquesta característica serveix per definir la capacitat de resistència al dolor del personatge. Com major sigui el valor d'aquesta característica menys mal se li podrà fer rebre un atac. Funciona com a modificador parcial i aleatori en la rebuda d'atacs. Al igual que els modificadors d'armadures, que també donen defensa a l'hora de rebre atacs, però que les armadures serveixen com a defensa a nivell local, és a dir, que hi han armadures per al cap, per al cos i per les cames.

LIFE

Aquesta característica defineix els punts de vida del personatge, quan aquests arribin a zero haurà mort. Existeix el seu modificador que mai es podrà augmentar més del que té l'usuari com a base. D'aquesta manera el jugador podrà utilitzar ampolles i farmacioles per recuperar punts de vida.

AIM

La capacitat de punteria del personatge, el seu pols per mantenir un punt objectiu fixat al punt de mira. Quan el client dispara, el servidor rebrà uns valors que li permetran realitzar un vector de direcció, però aquest vector variarà en major o menor mesura (de manera aleatòria) depenent del valor d'aquesta característica.

Conclusions

Un cop finalitzat el projecte podem veure d'una manera més global tot el treball que hem realitzat.

Gràcies a la realització d'aquest projecte hem après bastants coses que abans ignoràvem, i ens han fet veure els jocs, des d'un altre punt de vista, ja que fins ara només veiem la part externa d'aquests i ara hem tingut la possibilitat de poder-ne realitzar un.

Hem pogut veure totes les complicacions que això representa, tant a nivell organitzatiu, com tècnic, com de disseny. I tot això marcat dins d'uns límits temporals que ens han obligat a marcar-nos unes pautes de treball.

També hem apreciat la dificultat de realització d'un projecte tant gran, ja que fins ara tots els projectes de programació que havíem realitzat eren força senzills i simples. Això ens ha obligat a plantejar-nos punts com el d'estructuració del codi, que és un punt molt important ja que a mesura que va creixent el projecte es fa més difícil desenvolupar-lo.

Les matemàtiques, geometria i física són un punt molt important, i potser el més complicat de realitzar. Malgrat haguem utilitzat unes llibreries ja creades, de bon principi, vam intentar escriure-les, cosa que ens ha permès entendre com funcionen internament tots aquests sistemes de geometria a l'espai, com col·lisions, rebots, friccions, impactes i la gravetat.

El treball en grup ens ha representat aprendre a repartir les feines i organitzar-les de manera equitativa a les capacitats de cadascun. I al treballar més d'una persona sobre el mateix codi implica una gran coordinació, ja que sinó el treball que fa un pot ser anul·lat per l'altre. El mateix succeeix a l'espai temporal, que

ens ha calgut marcar-nos prioritats dins del projecte i fixar uns objectius de feina realitzada de manera periòdica.

El haver-nos marcat com a objectiu crear un joc en multiusuari, ens ha fet aprendre també de les dificultats que comporta no tant sols el desenvolupament d'un joc en si, sinó l'estructuració d'un sistema en xarxa, els problemes de la lentitud de la connexió el temps d'espera entre diferents màquines més ràpides i més lentes, etc.

Hem après a utilitzar i ajuntar diverses llibreries com *SDL*, *OpenAL*, *OpenGL*, *Fltk*, *Pthreads*, *Bzip2*, *OggVorbis*, i escriure el codi provant-lo en diferents sistemes operatius (*NetBSD*, *DebianGNU/Linux*, *Windows*) i compiladors (*gcc/vc++*). Encara que malgrat la manca de temps no hem pogut realitzar tot el codi el 100% portable (de moment només funciona amb *windows*), amb totes les proves que hem realitzat no costaria gaire arreglar el codi per fer-lo portable.

El resultat de tot el projecte, malgrat el no haver pogut incloure tot el que volíem i què, ens ha fet veure els problemes que implica el començar, sense tenir els coneixements adequats. Però gràcies a això hem pogut aprendre molt i de molts temes, tant tècnics com organitzatius, fet que ens permetrà en un futur realitzar altres projectes semblants amb més experiència.

En un futur proper, ens agradaria acabar de retocar tots els punts que ens hauria agradat implementar, com les partícules, bumpmapping, lightmapping, multiplataforma i multiarquitectura, ús del protocol UDP enlloc del TCP, carga de fitxers comprimits (*OggVorbis*, *Bzip2*), entre d'altres.

Com a conclusió final podem dir que estem contents dels resultats del projecte, tant pel nostre compte (coneixements adquirits), com pel resultat d'aquest en si.

PROBLEMES CONEGUTS O COSES PENDENTS

GLH està pensat per a ser un joc en continua evolució i millora. És per això que encara no donem per acabat el projecte i només presentem una beta del que serà el joc en realitat.

Problemes coneguts:

1 – El PVS funciona bé però, algunes vegades, la detecció de col·lisió amb els portals falla i el joc no s'assabenta que hem canviat d'habitació. Això provoca que acabem trobant llocs en els quals no es pinta el mapa i no veiem res. Per evitar aquest problema s'ha inclòs un define per activar-lo i desactivar-lo.

Coses pendents per implementar:

1 – Actualment el joc no fa servir il·luminació. Podríem haver utilitzat la il·luminació que incorpora *OpenGL* però es extremadament lenta i no em tingué temps de posar-hi light-mapping.

2 – Falta posar efectes com, per exemple, bump-mapping (mitjançant el qual una textura “agafa” propietats semblants al 3D i aconsegueix que un objecte sembli q tingui més polígons dels que té en realitat).

3 – Implementar sistemes de partícules.

4 – El control del teclat encara no està del tot implementat.

Annex

Funcionament de l'accés a bases de dades

Introducció:

L'accés a bases de dades a través del servidor del joc es realitza amb la finalitat de validar usuaris creats a través d'una web (<http://glh.sf.net/index2.php>). I que aquesta internament treballa amb una base de dades MySQL accedint desde PHP.

El fet de simplificar aquest model per tal que el client no accedeixi directament a la base de dades MySQL, degut als problemes de seguretat que implica això.

Així que vam optar per crear un fitxer PHP que servís d'interfície entre la base de dades i el servidor del joc. D'aquesta manera el servidor realitza connexions HTTP passant els paràmetres amb el mètode GET i recollint els resultats.

La interfície permet tres tipus d'entrades definides en la variable "type" aquestes són: "auth", "kal" i "del".

MÈTODE AUTH:

Aquest mètode de connexió serveix per autenticar un usuari, per fer-ho, li passa tres paràmetres. Si la validació es correcta el servidor contestarà un ID d'usuari i la seva fitxa. A partir d'aquí el client (el servidor del joc) entrarà en un bucle de dependències per baixar-se tota la resta de fitxers que necessita per aquella fitxa (textures, vertex, etc.) utilitzant el mètode GET de l'HTTP1.0

usr

Aquesta variable serveix per indicar el nom de l'usuari que s'està connectant.

pwd

En aquest paràmetre se li passen els quatre bytes separats per comes del crc32 del password encriptat.

key

Aquesta variable contrindrà els quatre bytes que defineixen la clau aleatoria que ha creat el servidor en l'autenticació.

kal

Aquest mètode serveix per enviar una senyal "Keep Alive" per indicar que l'usuari encara es manté viu.

usr

Indica el nick de l'usuari del que vol validar per a mantenir el seu IDLE de conexio.

id

El numero d'identificador que el servidor li ha assignat a l'hora d'autenticar-se contra ell.

MÈTODE DEL:

Aquest mètode serveix per tancar la sessió de l'usuari autenticat. Caldrà passar-li els mateixos paràmetres que el mètode "kal".

usr

Indica el nom de l'usuari a des-loguejar.

id

El ID que identifica a l'usuari dins de la connexió cap al servidor oficial.

La part de servidor

La part del servidor està implementada mitjançant un script php que interactua amb la base de dades MySQL que ha estat creada desde uns formularis desde web.

Internament el servidor emmagatzema les dades dins de tres taules SQL:

```
mysql> show tables;
```

```
+-----+
| Tables_in_glh |
+-----+
| personal      |
| role          |
| users         |
+-----+
```

I cada taula emmagatzema certes dades de manera que s'accedeixi a la informació de manera ordenada i clara, i que per tal que els accessos a la base de dades siguin òptims ja que el volcat d'informació i l'accés a aquesta sigui només el necessari, sense carregar taules massa grans i agafar només 2 camps d'aquesta.

Totes les taules estan unides pel camp "nick" que identifica el nom de l'usuari.

El contingut de les taules és el següent:

taula personal:

```
mysql> describe personal;
```

Field	Type	Null	Key	Default	Extra
nick	varchar(12)	YES		NULL	
pass	varchar(12)	YES		NULL	
descripcio	varchar(255)	YES		NULL	
model	varchar(20)	YES		NULL	
data	varchar(20)	YES		NULL	

```
+-----+-----+-----+-----+-----+
```

Aquesta taula serveix per emmagatzemar totes les dades personals de l'usuari, és a dir el nom, la contrasenya, la descripció del personatge (aquesta descripció l'omple el jugador i és una informació pública), el model de personatge i la data de creació de l'usuari.

taula role:

```
mysql> describe role;
```

Field	Type	Null	Key	Default	Extra
nick	varchar(12)	YES		NULL	
life	int(11)	YES		NULL	
def	int(11)	YES		NULL	
str	int(11)	YES		NULL	
aim	int(11)	YES		NULL	
level	int(11)	YES		NULL	
exp	int(11)	YES		NULL	
ptt	int(11)	YES		NULL	

En aquesta taula és on s'emmagatzema la fitxa de personatge, és a dir, totes les característiques que el defineixen, aquestes són: punts de vida, defensa, velocitat, força, punteria, nivell del personatge, punts d'experiència adquirits i punts a repartir a la fitxa.

taula users:

```
mysql> describe users;
```

Field	Type	Null	Key	Default	Extra
nick	varchar(12)	YES		NULL	
id	int(11)	YES		NULL	
server	varchar(30)	YES		NULL	
idle	int(11)	YES		NULL	

first	int(11)	YES		NULL	
-------	---------	-----	--	------	--

Aquesta taula SQL serveix per emmagatzemar totes les dades referents a les connexions dels usuaris. És a dir, a través de la web es podran consultar tots els servidors oficials que estiguin en aquell moment connectats i veure els usuaris que hi han dins, el temps que porten jugant i les seves característiques, etc.

Quan un usuari es valida correctament contra el servidor aquest es apuntat dins d'aquesta taula, indicant el ID seu, el nom/ip del servidor en que està connectat, la hora en que s'ha connectat i quan ha enviat l'últim missatge de KAL (Keep ALive).

El servidor eliminarà el registre de l'usuari passats 3 minuts de connexió. Les variables de temps les emmagatzema amb un valor numèric que és el resultat de la suma de segons passats desde l'1 de gener de 1970 GMT.

Les comprobacions de delays de temps les realitza cada cop que es fa un llistat de servidors, així s'evita veure clients que s'hagin caigut i no hagin enviat més paquets de KeepALive...de totes maneres quan l'usuari envia un paquet "del" per eliminar la connexió tanca el seu registre sobre la connexió.

Un usuari només podrà estar connectat a un sol servidor, és a dir, que si un usuari ja està connectat en un servidor, la seva conta d'usuari no la podrà utilitzar ningú més, ja que en el temps d'autenticació li denegarà indicant que la seva conta d'usuari està essent utilitzada.

Col·lisions

Per a les col·lisions hem creat una classe totalment reaprofitable en qualsevol tipus de joc. Aquesta, de moment, detecta col·lisions amb plans i geometria simple. Els personatges son aproximats a esferes per a reduir la feina de la CPU i fer-les més optimes. Per aquesta mateixa raó, quan s'agreguen els vèrtex de les habitacions, la classe crea un bounding alrededor d'aquestes i només comprova les col·lisions del personatge amb els vèrtex de la habitació on és troba.

Pressupost

	octubre	novembre	decembre	gener	febrer	març	abril	maig
Sergi	servidor de xarxa			client de xarxa			so	database
Marc	disseny	model·lat				texturitzat		web
David	convertors	motor del joc						col·lisions

Contant que hem realitzat un treball d'unes 20 hores setmanals durant 8 mesos:

$$(20 \cdot 4) \cdot 8 = 80 \cdot 8 = 640 \text{ hores totals.}$$

Hem establert uns sous en vista a les responsabilitats i feina assignada dins del projecte:

Sergi - 1.200 pts/h

Marc - 900 pts/h

David - 1.200 pts/h

Que equival a uns sous de 96.000 / 72.000 / 96.000 respectivament al mes.

També hem afegit un sou mitjà com a gastos comuns d'unes 200pts/h. que fan un total de 16.000 pts/mensuals de gastos comuns.

Així doncs podem calcular un preu total del projecte i valorar-lo en:

$$((2 \cdot 96.000) + 72.000 + 16.000) = 280.000 \text{ pts /mes}$$

I multiplicant aquest preu pel número de mesos fan un total de:

$$280.000 \cdot 8 = 2.240.000 \text{ pts}$$