IGVCRAFT



**Proyecto de Informática Gráfica y Visualización**

David Martínez Alcántara

dma00016@red.ujaen.es 77690690W

INTRODUCCIÓN

Características de igvCraft:

* [Visualización de **Bloques en Chunks**](#Chunks)
* [**Eficiencia** del Renderizado del **mundo**](#Eficiencia)
* [**Selección** de Bloques y Eliminación y Colocación por trazado de rayos + selección por color](#Seleccion)
* [**Iluminación** dinámica por normales](#Iluminacion)
* [**Texturizado** de bloques y el personaje](#Texturizado)
* [Modelado de un Enderman como **personaje** y sus movimientos](#Enderman)
* [Movimientos de la **cámara** y sincronización con el modelo del personaje](#Camara)
* [**Otras** Características](#otras)

Los **parámetros modificables** se encuentran en **Constantes.h**:

tamChunk Dimensión de un Chunk en X y Z

maxHeight Máxima altura de construcción del mundo

alturaMundo Altura a la que se genera el mundo al ejecutar

distanciaRender Cuántos chunks se visualizan alrededor del jugador

amplitudMapa A más pequeño más se distinguen los elementos en el mapa

alcance Distancia máxima de selección de bloques

precisión Precisión del trazado de rayos a la hora de seleccionar un bloque (a más pequeña menos eficiencia, menor margen de error)

sensibilidad Sensibilidad de la rotación de la cámara

velocidadMovimiento Distancia recorrida al pulsar una tecla de movimiento (flechas o wasd)

posInicialJugador Posición Inicial del Jugador

texturasBloques Array que almacena la ruta de las imágenes para texturas, puede añadirse más texturas

sol Fuente de Luz que simula el sol

Visualización de Bloques en Chunks

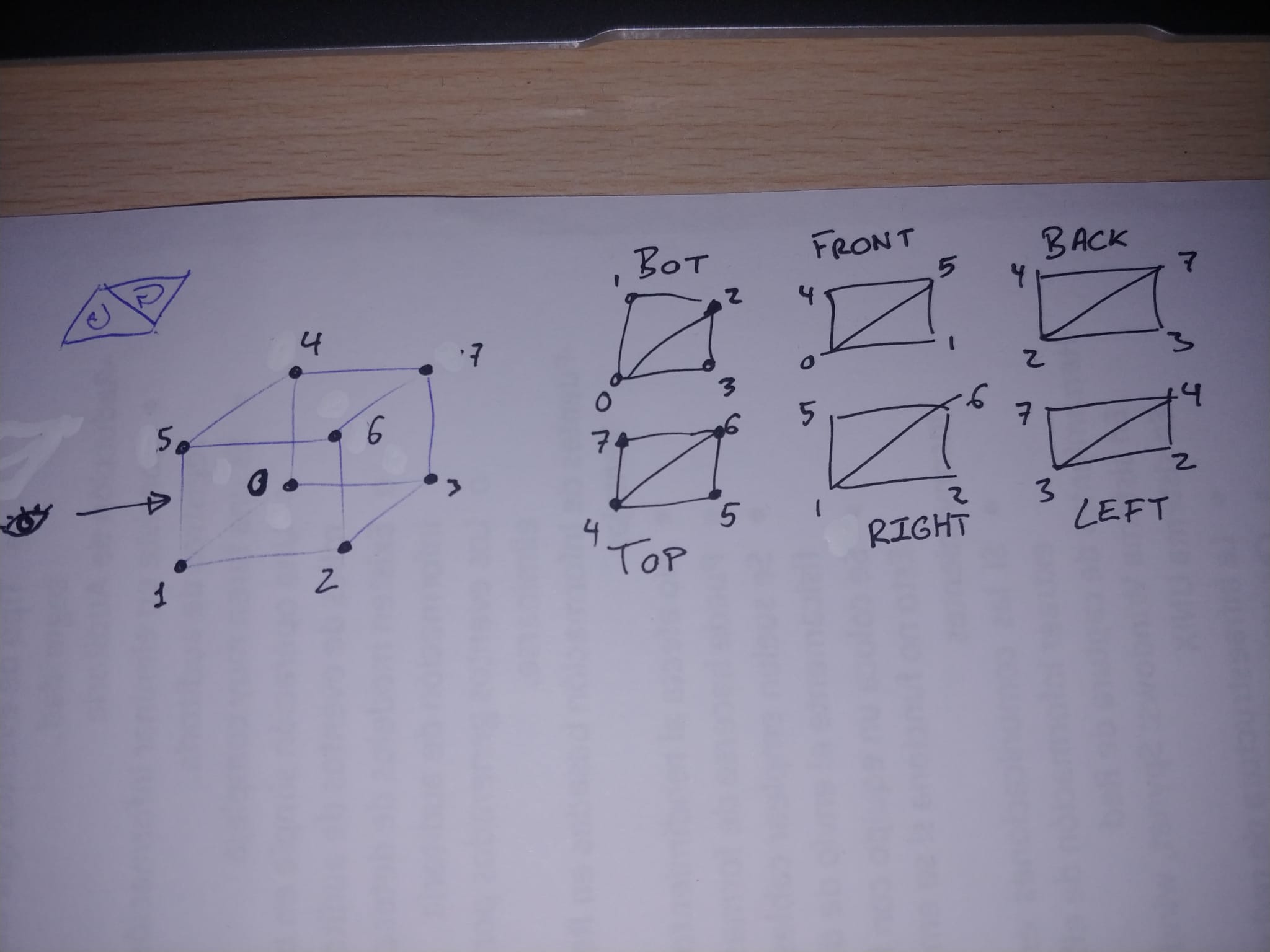
Los Bloques tienen una **coordenada global** que establece la posición del **vértice 0** en el mundo (el vértice de menor X, Y y Z).

Para visualizarlo uso **arrays de vertices**, un array **principal** con todos, que de primeras me sirvió para hacer un **visualizado completo**, y **6 arrays** más asignados a **cada una de las caras** para su visualización **independiente**.

Lo mismo para las **normales** que se asignan a partir de **un array del bloque completo a** **6 arrays de las caras** con *dividirNormalesEnCaras().*

Para dibujar el cubo usamos una **malla de triangulos**, que al aplicarla **cara a cara**, si los vertices se encuentran en el mismo orden, podemos usar un array de índices general que sirve para cualquier cara. Ésta se encuentra en una clase **Malla** que almacena tanto los **índices** como las **coordenadas de textura** en un array. Además tiene otros arrays que al principio utilicé para la visualización del bloque sin separarlo en caras.

Los **vértices se calculan teniendo en cuenta la** **posición del bloque** (coords) y el esquema de visualización sería este (el ojo mira en X positivo):



Los bloques tienen un tamaño de **1x1x1** y están situados en un Chunk, siendo éste de tamaño variable en Constantes.h, siendo el original en Minecraft 16x16, pero en mi caso he estado trabajando sobre un **Chunk 4x4**. La **altura** va de 0 a 64 pero no hay Chunks a más o menos altura, solo se representan con las 2 dimensiones horizontales.

Los **Chunks** son los **encargados de** **controlar cambios en los bloques**, **darles las propiedades**, **texturas y almacenarlos**. Al crear el Chunk, se crean **capas de Bloques** d**e diferente tipo según la altura**, siendo la 0 Bedrock, encima Roca, encima Tierra y arriba Césped. La **altura del mundo inicia**l puede ser modificada en Constantes.h.

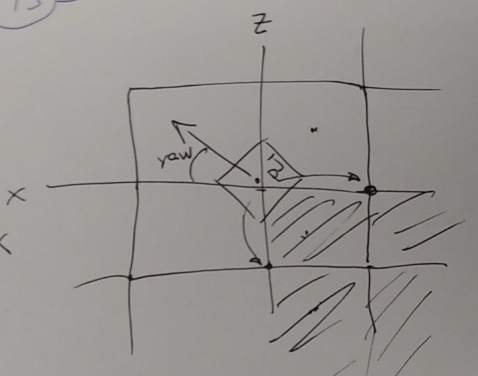
Cada **Chunk** tendrá una **coordenada asignada** correspondiente a la **esquina inferior** y una **coordenada relativa** que funciona como **ID** (***numChunk***), siendo el Chunk (0,0) el que va de 0,0 a 16,16, y el (-1,-1) el que va de -16,-16 a 0,0.

Esto permite saber en qué Chunk se encuentra un Bloque **convirtiendo su coordenada a la relativa** a los chunks como **floor(coord[X & Z]/tamChunk).**

Los **bloques** son almacenados en un **mapa** donde su **clave** es la **coordenada relativa** al Chunk, es decir, **coordBloque[X & Z] % tamChunk**.

Cada Chunk lleva un **puntero a sus 4 vecinos** en X y Z para posteriores cálculos de bloques adyacentes que se encuentren **al borde del chunk**.

Eficiencia del Renderizado del Mundo

El mundo tiene una **amplitud fija**, dependiendo de los chunks creados, es decir, no se genera conforme se avanza en el mundo, en este caso por defecto se crean 8 x 8 chunks, con centro en 0,0, es decir, **el mundo se expande a partir de 0,0** **en todas las direcciones** 4 chunks. De esos chunks se podrán **ver una cantidad limitada por la distancia de renderizado** para mejorar el rendimiento, con centro el jugador.

Además, se calcula la posición del Chunk respecto al jugador, y si éste **no está en su campo de visión**, que realmente está establecido como un ángulo de 270º porque los chunks son muy grandes. Así podemos **dejar de visualizar el 25% de los chunks**, los que están **detrás del jugador**, y algo de eficiencia se consigue, aunque con una distancia de renderizado pequeña no tiene mucho impacto, pero a grande escala son muchos chunks y carga de cómputo que se ahorra.

Por otra parte, al ser cada cara de un bloque independiente, podemos calcular si **una cara está siendo tapada por un bloque**, si es así **no se visualizará**. Esto si supone un aumento de la eficiencia muy grande. Se calcula en el método *ladoExterior()* de Chunk, que además usa los punteros a chunks vecinos en caso de estar en un borde.

Este control de caras externas o tapadas junto con el **Culling** aporta la mayor eficiencia al programa, ya que **evitamos la mayoría de cálculos de iluminación y texturas**.

Esto es perceptible desde la cámara Libre o la Panorámica (Pulsar C)

Selección de Bloques

(Construcción / Eliminación)

Para **seleccionar un** **Bloque** utilizo **Trazado de Rayos**:

Para simular un rayo utilizo un **punto detector** que comprueba si **en su posición hay un bloque**. Si no lo hay **se mueve en la dirección a la que mira la cámara** una distancia dada por el parámetro *precision,* hasta una distancia máxima dada por *alcance.* Si encuentra un bloque lo selecciona. Si se ha pulsado el **click izquierdo lo eliminamos**.

Para **seleccionar una** **Cara** utilizo **Selección por Color**:

En caso de haber pulsado **click derecho**, pasamos a una visualización por colores, donde se quita todo lo demás y **solo se visualizan las caras del bloque seleccionado con un color distinto por cara**. Una vez conocemos el color podemos saber su cara. Dependiendo de la cara seleccionada colocamos un Bloque donde corresponda.

El bloque colocado es controlado por el atributo *bloqueUsado* en igvInterfaz, que es el enum tipoBloque que utiliza cada Bloque para saber la textura que necesita aplicar.

La clase de Bloque se puede ver al manejar al jugador **agarrando el bloque con las manos**. Y puede **cambiar con la rueda del ratón**. Como es un enum, cada tipo de bloque tiene asignado un entero, y solo hace falta aumentar o disminuir ese entero conforme gira la rueda del ratón.

Iluminación dinámica por normales

Como ya he mencionado las normales de un **bloque aislado** son **diagonales** **desde el centro al vértice**, resultado de la **suma de las normales de las caras adyacentes**. Para calcular la iluminación se necesita conocer lo bloques adyacentes. Si una **cara se tapa por otro bloque**, esa cara **ya no sumará** para dar la normal resultante, por lo que si, por ejemplo, tenemos la esquina donde la normal es (1,1,1) y ponemos un bloque encima hemos tapado la cara superior cuya normal es (0,1,0), por lo que la normal del vértice resultante debe ser (1,1,1) – (0,1,0) = (1,0,1).

Así podemos deducir que **cada bloque que tape una cara supone la eliminación de una coordenada de la normal de los vértices que forman la cara tapada**. En caso de estar encima o debajo, la Y será 0, si está detrás o delante en la coord X, X = 0. Y para la Z se aplica lo mismo.

El **Chunk detecta bloques adyacentes** en *actualizarNormales()* y **controla si la actualización ha sido por la colocación del bloque o su eliminación**, siendo el parámetro *eliminado* el encargado de luego, en el método *actualizarNormales(bloque que se eliminó/colocó, eliminado)* de Bloque, el que controla que la coordenada de la normal se anule a 0, o vuelva a aparecer con 1 o -1.

En caso de bloques en el borde del chunk se utiliza el puntero al chunk vecino en su respectiva dirección.

Texturizado de bloques y el personaje

He aislado la carga de las imágenes en el constructor y a diferencia de la práctica de texturas, mis objetos igvTextura ahora guardan un **objeto SDL\_Surface**, para que **al aplicar pueda coger el array de pixeles** de la imagen **sin tener que recurrir a lectura de archivos**. Así puedo **cargar todas las texturas antes de iniciar el mundo** y guardarlas **en *Constantes.h***.

Para hacerlo aún más eficiente la textura no se aplica a cada bloque visualizado, ya que por la construcción del mundo en capas del mismo tipo de bloque no es necesario, así que el **Chunk se encarga en *visualizar()* de tener un iterador que vaya atrasado al iterador del bloque para comparar el tipo de bloque de uno y su anterior**. Si éste cambia se aplica la textura, pero si no, no hace falta.

En el caso de bloques con **texturas complejas**, que necesiten **más de una** como el **tronco** o el **césped**, se controla que antes de visualizar la cara se aplique la textura correspondiente. En el caso del césped, **arriba se aplica Grass, abajo Dirt y en los laterales GrassSide**.

Para facilitar esta tarea el enum de tipoBloque tiene para el césped un valor **Grass** y el siguiente valor es **GrassSide**. No puede existir un bloque de tipo GrassSide pero facilita el código, permitiendo que otros bloques con texturas complejas, como el tronco, funcionen igual.

En el caso del personaje he añadido unos métodos en *Constantes.h*, en concreto *drawSolidCube()* que sustituye la primitiva de openGL y añade las **coordenadas de texturas**. Así puedo aplicar la textura del Enderman a cada parte de su modelo **antes de transformarlas**.

Para los ojos de la **cara** utilicé una segunda textura que se aplica con otro método *drawFace()* que **adapta el anterior para solo dibujar una cara**, y la superpongo a la cara posterior de la cabeza por encima del cubo.

Modelado de un Enderman y movimientos

El modelo del Enderman es importado de la **práctica 3b**. Todos los movimientos están implementados y pueden activarse con los números (1-Andar, 2-Sentarse, 3-DAB, y los demás comandos que implementé en la práctica 3b, pero no es necesario para el juego).

El añadido aquí es su **textura** y la **animación de andar** que se **activa** si se pulsa las **flechas o wasd** y **para al soltarlas**.

Además, cuando se activa la **cámara en 1º persona** se adapta el modelo **eliminando el torso y piernas y mostrando los brazos levantados agarrando el bloque seleccionado**. Como mencioné antes, es el bloque que se colocará al hacer click derecho y puede cambiar rotando la rueda del ratón. Los brazos giran según las inclinación de la cámara para seguirla.

**La cabeza tiene una rotación sincronizada con la cámara verticalmente** y el **cuerpo** lógicamente **aplica la rotación horizontal de la cámara** y se mueve a la vez que la cámara con las flechas o wasd.

Movimientos de la cámara

y sincronización con el personaje

Hay **4 cámaras**:

**Cámara Cenital**: Se coloca justo encima del personaje y mira hacia abajo simulando un **minimapa**. Ésta se aplica junto a un **Viewport** pequeño en la esquina superior derecha de la ventana y **sigue al jugador**.

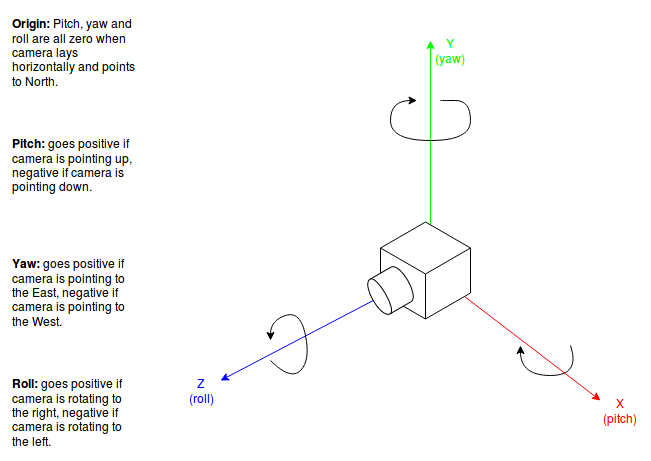
**Cámara Panorámica**: Da una visión panorámica de la escena y **sigue al jugador** a la par de la cenital.

**Cámara Libre**: **No está enlazada con el personaje** y permite ver todo el escenario. Con ella se puede comprobar como los objetos fuera de la visión del jugador no se visualizar.

**Cámara del Jugador**: Está sincronizada con el personaje y se ubica **en el centro de la cabeza del Enderman**.

A excepción de la Cenital todas pueden **rotar**. Para ello utilizo una especie de **esfera alrededor de la cámara cuyo radio es 1** y es la **distancia entre la cámara y su punto de referencia**, el cual va girando conforme muevo el ratón. Para ello he utilizado un modelo de rotación con **yaw** (horizontal) y **pitch** (vertical).

Al construir la cámara inicializo yaw y pitch como:



pitch = arcsen(r[Y] - P0[Y]);

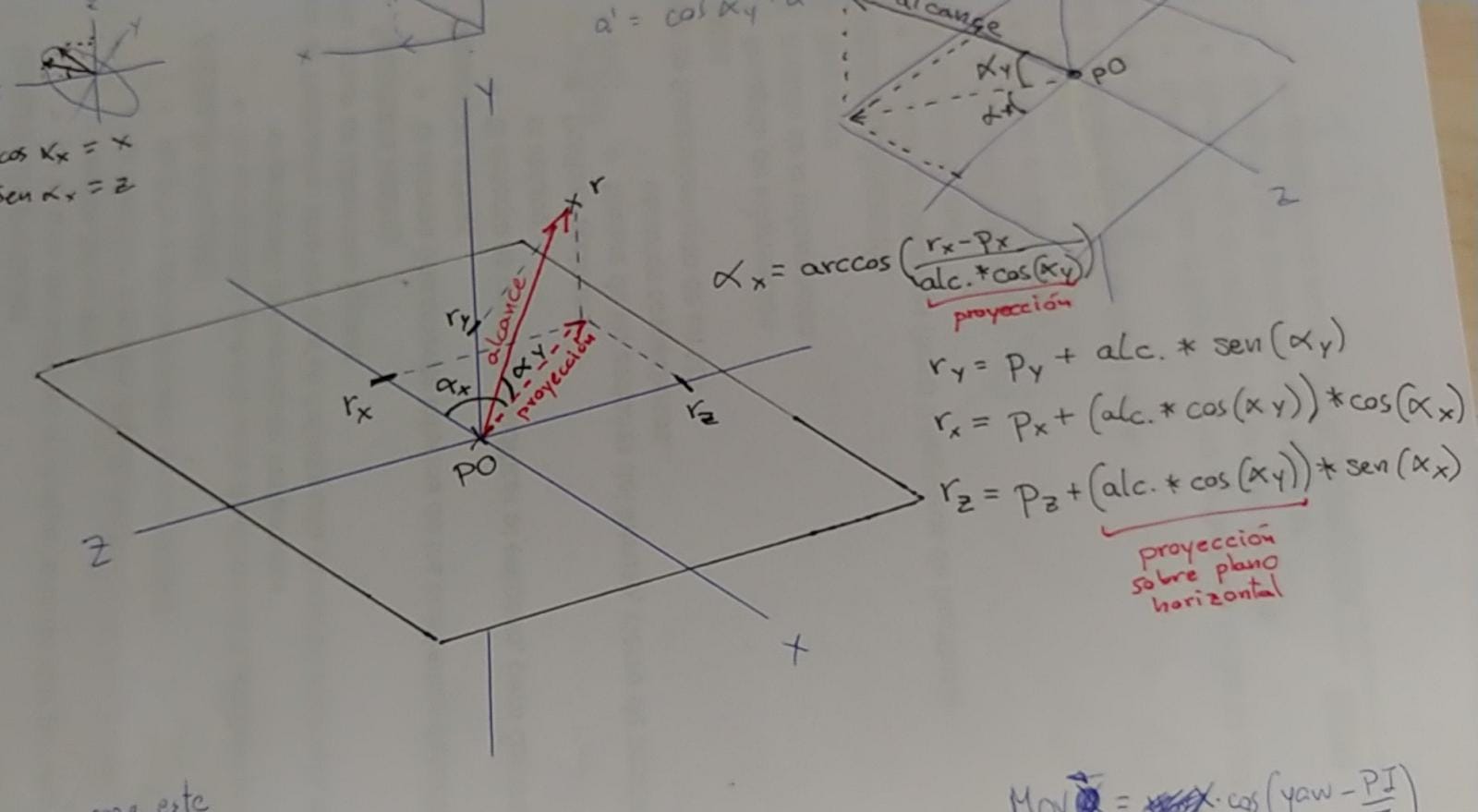
yaw = arcos((r[X] - P0[X]) / cos(pitch));

Y cada vez que muevo el ratón roto la cámara un ángulo calculado como la **diferencia del centro de la ventana a la nueva X o Y del ratón entre un factor sensibilidad**.

Cada ángulo **se suma al yaw y al pitch** y se **recalcula la posición del punto de referencia** que se debe de mover por la esfera de rotación que rodea la cámara tal que:

r[X] = P0[X] + cos(pitch) \* cos(yaw);

r[Y] = P0[Y] + sin(pitch);

r[Z] = P0[Z] + cos(pitch) \* sin(yaw);

Además, se **limita el pitch a 90º y -90º** para no permitir ver todo al revés al superar el límite.

Todas se mueven al usar las flechas o wasd y el Espacio o Shift+Espacio. Y utilizan el yaw para orientarse. Al moverse a un lateral, adelante o atrás se calcula por medio de cosenos y senos del yaw y **se multiplica por la velocidad de movimiento** que se haya establecido.

Y en caso de estar **moviendo la cámara del Jugador, la Panorámica o la Cenital**, se moverán las otras dos en la misma dirección respecto al yaw de la cámara usada para así **poder mover las 3 cámaras a la vez**.

El **yaw y el pitch** es también utilizado para **modificar la rotación de la cabeza** **y del modelo completo para sincronizarse con la cámara**.

Para poder **ver a través de la cabeza sin esconderla**, y poderla ver en el minimapa a la vez, utilizo un **plano de proyección zNear alejado** lo suficiente para colocarse **por delante de la cara** del Enderman.

Otras Características

* **Fuentes de Iluminación:**

He reciclado la clase **igvFuenteLuz** de anteriores prácticas y he simulado un **sol** con un poco de inclinación para poder apreciar el sombreado.

Y además creé un bloque que emite luz, la **Glowstone**:

Cada vez que un bloque de Glowstone es colocado se crea una fuente de luz **en su centro** que emite una **luz difusa de color amarillo** con coeficientes de atenuación que la hacen ver muy **tenue**. Pero como se puede ver en el vídeo Demo, al no haber sombras y traspasar la luz del sol los bloques, se superpone con la amarilla y se hace blanca en las caras donde la luz del sol golpea.

* **Materiales:**

Solo he usado un **material básico en blanco** que aplico a todos los bloques, esto es para poder usar las texturas con GL\_MODULATE y permitir que la iluminación afecte a la textura. En el caso de GL\_REPLACE no le afectaba la luz.

* **Enum tipoMovimiento:**

Ha sido muy útil para simplificar el código. En principio lo utilicé para distinguir los **movimientos de cámara** **y personaje** en todas las direcciones con un switch.

Después lo usé para trabajar en **cada cara de un bloque** independientemente y encapsular más el código en muchos switch. Para ello he considerado que adelante es la cara frontal, atrás es la trasera, y los demás valores se sobreentienden

* **Creación de Mundo:**

He **limitado el mundo a una amplitud de unos pocos chunks** (8x8), los suficientes para comprobar que funciona el renderizado conforme voy avanzando y el desrenderizado de los chunks que dejo atrás.

Además, está hecho por **capas** como uno de los presets de Minecraft al crear mundo, el mundo **Ultraplano**, que tiene una capa inferior de Bedrock, encima roca, encima tierra y el césped arriba. Se puede añadir capas modificando el parámetro *alturaMundo*.

* **Centrar cursor:**

He usado *glutWarpPointer(anchoVentana/2, altoVentana/2)* para implementar la **cámara en** **1º persona**. Por otra parte, como no he implementado un cursor propio, una especie de mirilla o algo así, no he usado *glutSetCursor(GLUT\_CURSOR\_NONE)* que permite esconder el cursor.

PROBLEMAS Y SOLUCIONES

Mi mayor problema ha sido el **rendimiento**. Sobre todo, trabajando en un PC prestado que mueve lo más básico porque el mío se rompió. Para ello he tenido que implementar un **límite de renderizado**, **ocultamiento de caras con bloques adyacentes**, **culling**, o**cultar los chunks detrás del personaje**, **precargar todas las texturas** y **solo aplicarlas cuando sea necesario**, etc.

He tenido muchos problemas para implementar la **rotación de cámara**, por precisión en los cálculos que hacía que se calculase el coseno de 1,0000000… > 1 por ejemplo y devolvía NaN. Tuve que buscar información de rotaciones hasta que leí sobre el **yaw** y el **pitch** y aprendí a usarlos.

Tuve que **separar los bloques en caras independientes** para poder **aplicar las texturas**, pero luego me vino bien para implementar el **ocultamiento de caras** adyacentes a un bloque.

Al principio añadía **parámetros que necesitaban otras clases** y acababa con errores de **multideclaración de variables por relaciones cíclicas con includes**, así que decidí crear el archivo **Constantes.h** donde almacené todos los parámetros que me hacían falta.

BIBLIOGRAFÍA

Clases **igvPunto3D, igvColor, igvInterfaz, igvEscena3D, igvCamara, igvMaterial, igvTextura, igvFuenteLuz** tomadas **de prácticas anteriores** y modificadas a gusto propio, añadiendo algún operador a igvPunto3D, modificando igvTextura para poder cargar la textura desde el principio y aplicarla cuando quiera sin lectura de archivos, añadiendo tipos de cámaras y movimientos a igvCamara, y igvInterfaz y igvEscena3D han sido las que más he modificado para hacer funcionar todo.

MANUAL DE USUARIO

[VIDEO DEMO](https://youtu.be/LbWHMD9NwMk)

**Parámetros globales** a modificar en **Constantes.h**

**Instrucciones:**

* **Flechas / WASD** - **moverse**
* **Espacio** - **subir**, **Shift+Espacio** - **bajar**
* Click **izquierdo** – **Romper** Bloque
* Click **derecho** – **Colocar** Bloque
* Scroll **Rueda** Ratón – **Cambiar tipo de Bloque** para Colocar
* c – Cambiar **Cámara**
* e – Ocultar/Mostrar **ejes**
* Animaciones: Pulsar 1/2/3