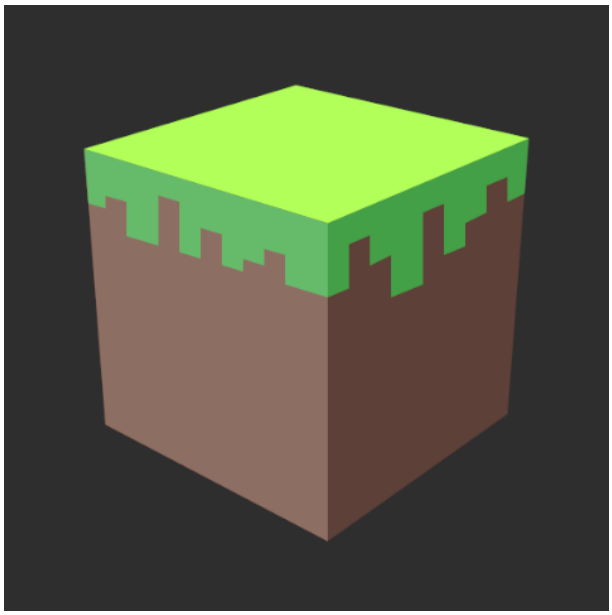
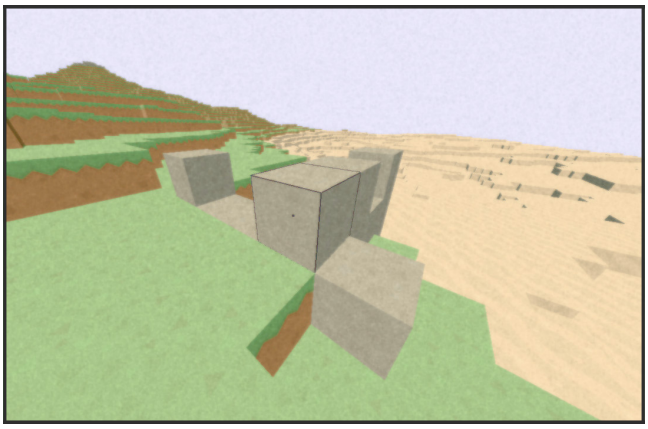


Computación Gráfica

Práctica

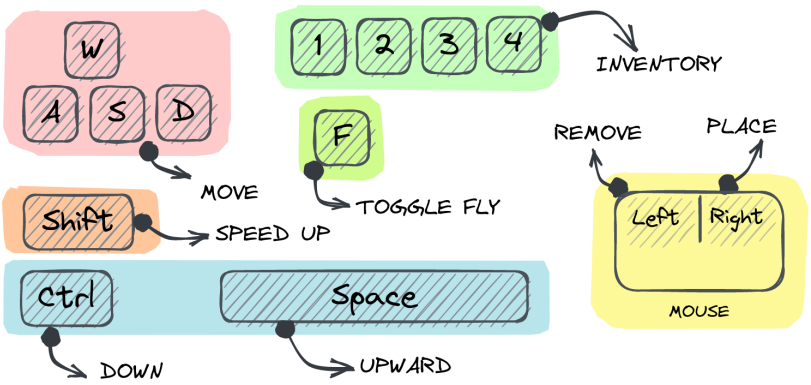
Eddie GERBAIS NIEF

JUAN FRANCISCO DE PAZ SANTANA



Introducción

El juego implementado es un *voxel sandbox* (como el juego *Minecraft*). El juego es implementado con el lenguaje *Rust* y utiliza *OpenGL*.



Funcionalidades

Comandos en el juego (desde el terminal).
Un terreno vóxel.
Generación procedural.
Poner y quitar bloques.
Primera persona con física.
Texturas.

Implementación

Make illegal states unrepresentable. Yaron Minsky

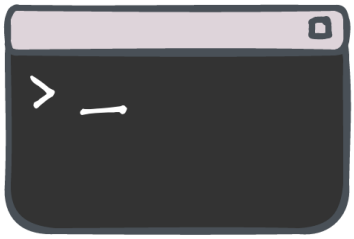
Técnicas	Computación con matrices y vectores. Multi-hilo, concurrencia y asyncronocidad. Ruido de Perlin para generación procedural. Creación de *shaders* y pipeline gráfico. Modificación dinámica del terreno. Detección de colisiones.

Particularidades del programa El programa es muy ligero debido que es hecho desde cero sin motor de juegos. Eso permite que se puede ejecutar en máquinas con especificaciones muy bajas y aun puede ofrecer ajustes altos. El programa es muy estable, porque el compilador del lenguaje *Rust* garantiza que no hay ninguna fallas de memoria (*segfault*, *null pointer*, *use after free*, *double free* o *memory leak*), ni carreras de datos. También, el sistema de tipos de *Rust* permite de reducir las fallas de lógica. El programa es disponible para todas las plataformas que soporte *Rust* y *OpenGL*, por ejemplo *Windows*, *Linux* y *Mac*.

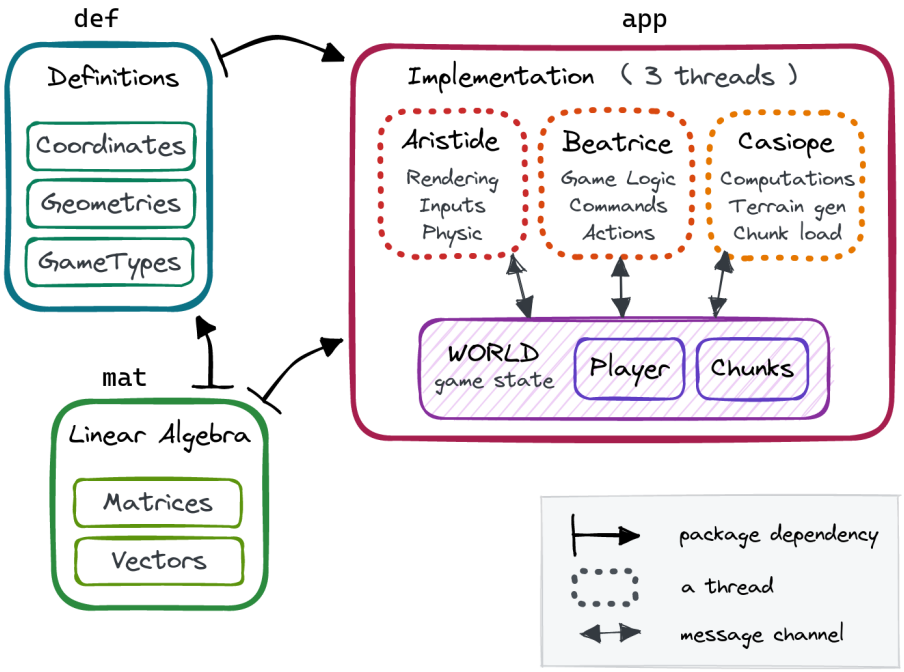
Bibliotecas utilizadas

`lalrpop`	(Analisador gramatical)	`https://crates.io/crates/lalrpop`
`glium`	(OpenGL wrapper)	`https://crates.io/crates/glium`
`tokio`	(Asíncrono runtime)	`https://crates.io/crates/tokio`
`noise`	(Ruido de Perlin)	`https://crates.io/crates/noise`

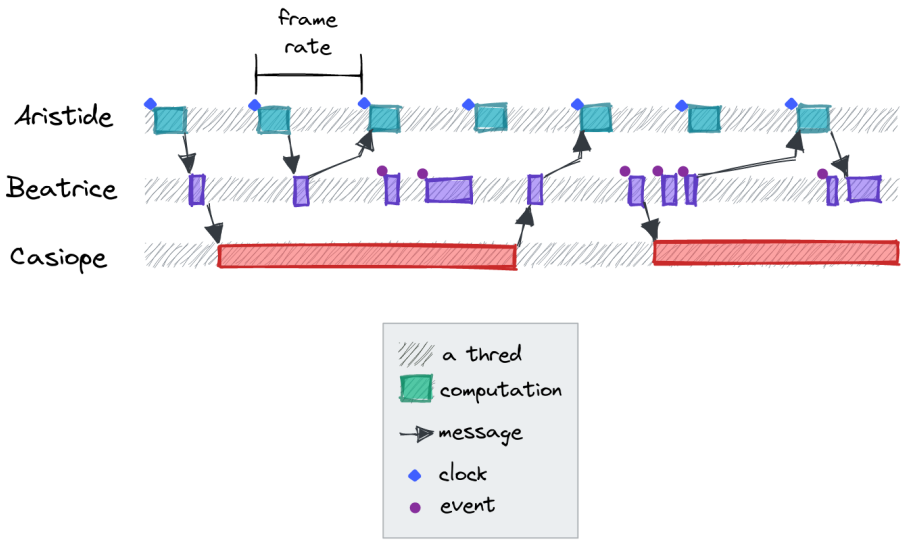
Terminal para comandos El programa lee comandos desde un terminal con un analizador gramatical *LR1*. Eso permite de mejorar el proceso de desarollo del programa.



`fly {true, false}`	Palanca volador.
`placing {stone, brick, glass, sand ... }`	Tipo de bloque colocado.



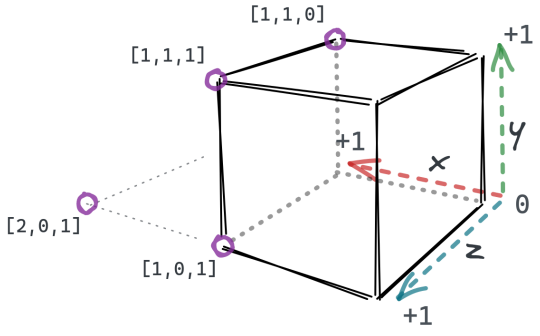
Paquetes El proyecto es separado en tres paquetes. El paquete `def` (corto para `definitions`) contiene informaciones sobre el juego y sus particularidades. El paquete `mat` (corto para `mathematics` o `matrices`) contiene computaciones con matrices y vectores. El paquete `app` (corto para `application`) contiene la implementación del juego.



Hilos El juego esta usando tres hilos para repartir la carga. Cada hilo tiene su nombre, que son *Aristide*, *Beatrice* y *Casiope*. El hilo *Aristide* maneja el renderizado, las entradas del usuario (teclado y raton) y la física del personaje. El hilo *Beatrice* acta como un coordinador. El hilo *Casiope* se encarga de las computaciones que necesitan mucho tiempo.

Vóxel coordenadas

El terreno esta compuesto solo de cubos (o bloques). El terreno es particionado en *chunks*. Un *chunk* contiene `16×16×256` bloques. Un bloque hace `1×1×1` metro de volumen. El **norte** es `z` negativo, el **este** es `x` positivo y la **ascendente** es `y` positivo.



Proyección de rayo

Para encontrar qué bloque apunta el personaje, una proyección de rayo es requerida (*ray casting*). Como el juego se basa sobre vóxel, puede ocurrir una colision (el rayo atraviesa un bloque) solo cuando se cruza un valor entero en uno de los axis (x, y o z), *Figura* **??**. Cuando se sabe eso, todos los futuros cruzamientos de enteros son previsibles porque ocurren a intervalos regulares (para cada dimensión), *Figura* **??**. Un intervalo mesura v_x^{-1} para el axis x dónde v es el vector representando la dirección del rayo (vector normalizado $\|v\| = 1$).

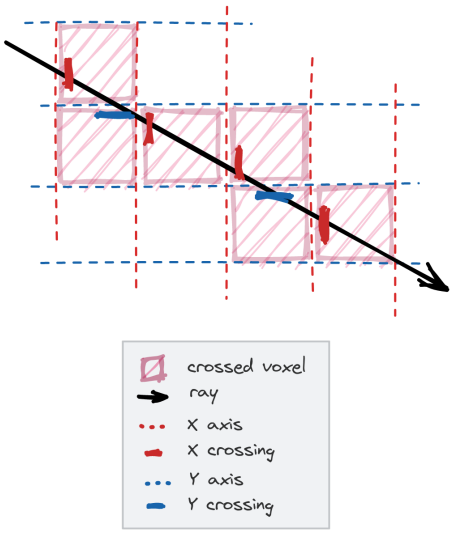


Figura 1: Proyección de rayos en dos dimensiones

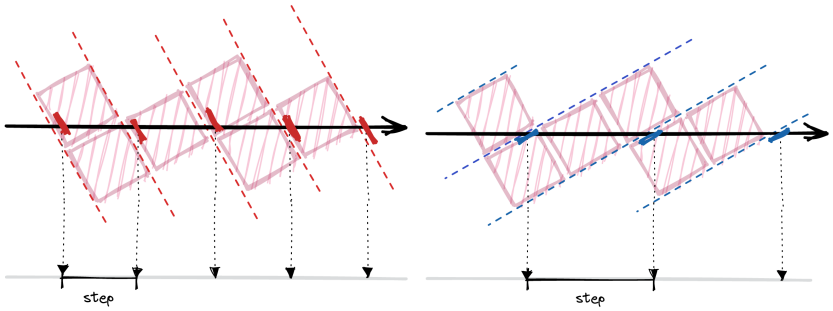


Figura 2: Intervalos regulares de cruzamiento

Colisiones

El personaje puede andar y saltar sobre los bloques sin pasar a través. Para garantizar eso, cada vez que el personaje se mueve, hay que buscar si su *hitbox* intenta superponerse sobre un bloque. Sea v el vector de velocidad y t la proporción de v que se puede lograr sin pasar a través de un bloque. Hay que buscar el t mínimum. Si no hay colisión, $t = 1$. Si hay una colisión $t < 1$.

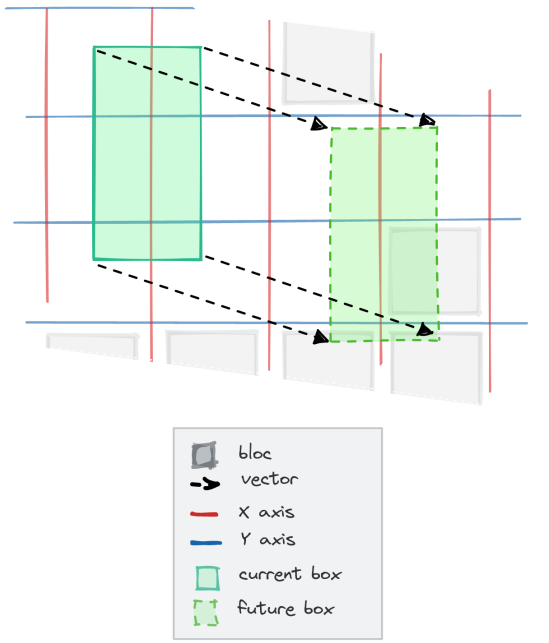


Figura 3: Colisión de *hitbox* en dos dimensiones

Para encontrar todas las posibles colisiones, se utiliza una proyección de rayo, entonces una detección de cruzamiento de axis. Pero, es mas complejo porque hay que proyectar el lado del *hitbox* a la posición predicada (sección del espacio). Si se hace en dos dimensiones, la proyección será en una dimensión, pero si se hace en tres dimensiones, la proyección será en dos dimensiones.

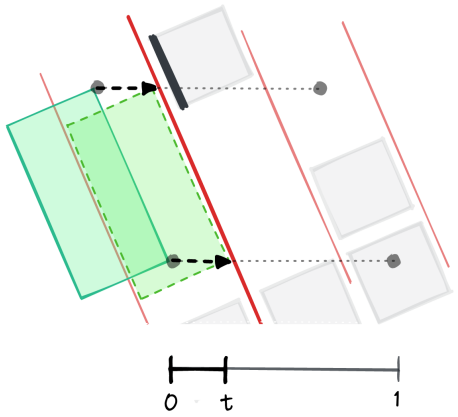
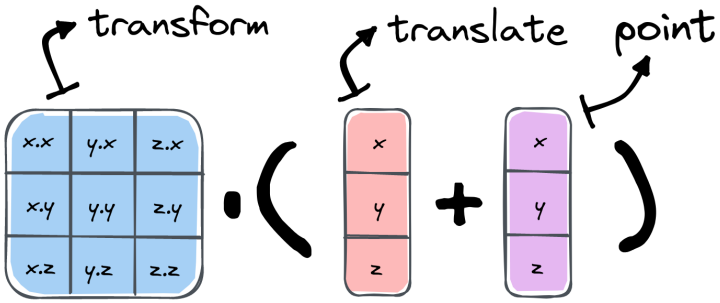


Figura 4: Buscar el t mínimum entre todas las colisiones

Rasterización



Vertex shader Para representar un mondo con objetos en la pantalla desde un punto de visto (cámara o personaje). Se almacena puntos que se llaman vértices (*vertex*) y se aplica una transformación lineal que los posicione en el espacio de la pantalla. Esta transformación es el inverso de la transformación del punto de visto. Por ejemplo, sea pos el vector de posición del punto de visto, y R su matriz de rotación. Sea p cualquier punto del espacio, p_w su vector de posición absoluto (en el mondo) y p_s su vector de posición en la pantalla, tenemos :

$$\begin{aligned} R \cdot (p_s + pos) &= p_w \\ p_s + pos &= R^{-1} \cdot p_w \\ p_s &= R^{-1} \cdot p_w - pos \end{aligned}$$

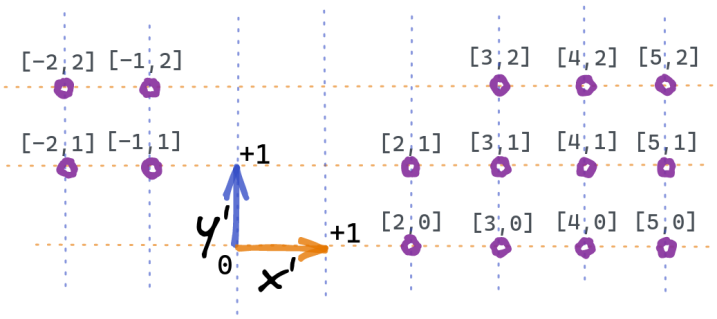


Figura 5: World vertices

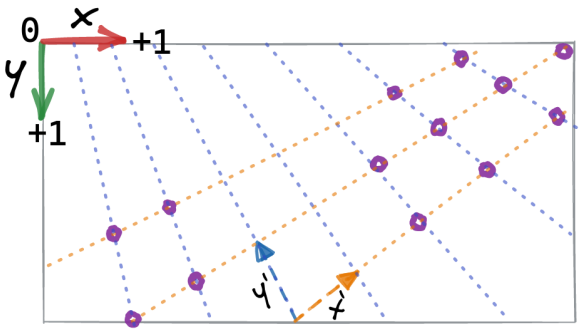


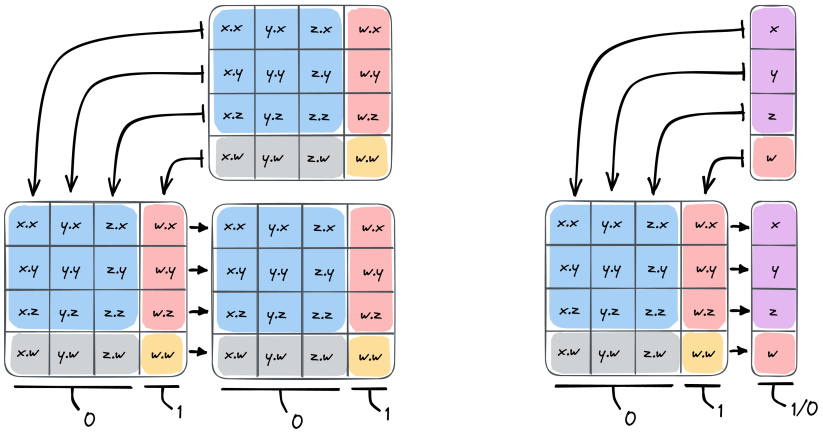
Figura 6: Screen projection

La etapa de posicionamiento de todos los puntos se llama el *vertex shader*, esa computación se hace en el procesador gráfico. Los programas que se ejecutan en el procesador gráfico se llaman *shaders*. Esa estrategia consiste a paralelizar con una técnica *SIMD* (*Single Instruction Multiple Data*) porque el calculo es lo mismo (instrucciones) para los numerosos vértices (datos).

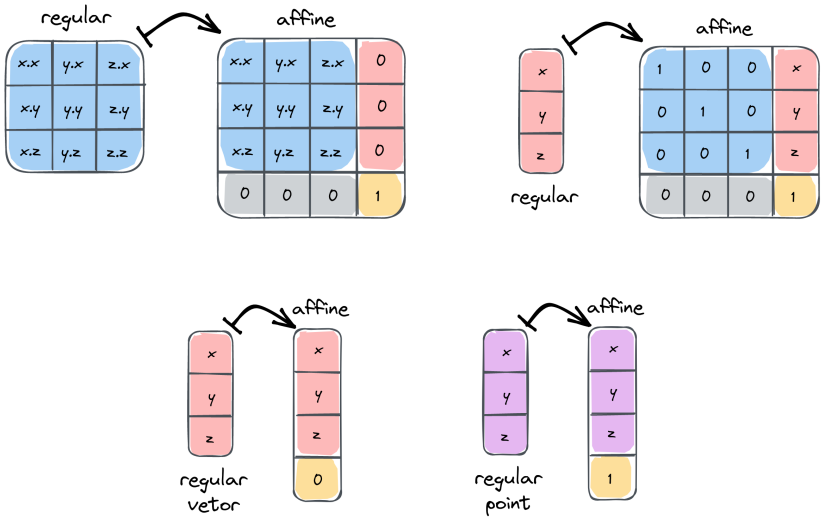
Con una transformación lineal sola, solo se puede obtener una proyección isométrica (conserva las lineas paralelas). Para producir un efecto de perspectiva (punto de fuga), una transformación polinomial (división por la profundidad) es requerida.

Además, cuando un objeto define su posición relativamente a un otro, la transformación se vuelve una cadena de transformaciones. Para evitar eso, se usa matrices y vectores de dimensión 4 para un espacio de 3 dimensiones. La cuaterna dimensión permite de acumular translación en una multiplicación de matriz. Esas matrices se llaman matrices afines.

Matrices afines



Si se añada una dimensión al los vectores (puntos y translaciones) y matrices (transformaciones), podemos codificar cualquier cadena de translaciones y transformaciones en una única transformación.



Esa cuaterna dimensión se llama w, los vectores de translación tienen $w = 0$ y los vectores de puntos tienen $w = 1$. Las matrices tienen $w = [a, b, c, 1]$ dónde $[a, b, c]$ es una translación, y $x_w = y_w = z_w = 0$.

Translación de un punto

$$\begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x+a \\ y+b \\ z+c \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} + \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x+a \\ y+b \\ z+c \end{bmatrix}$$

Translation de un vector (eso no afecta el vector porque tiene $w = 0$)

$$\begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix}$$

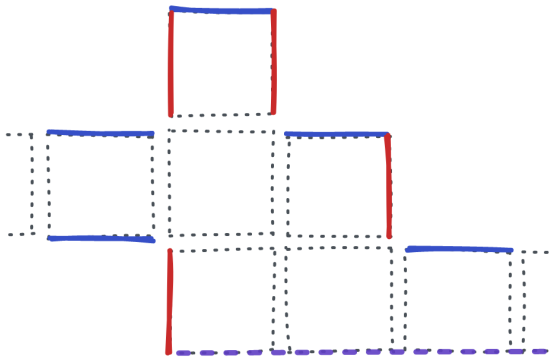
Transformación de un punto

$$\begin{bmatrix} x_x & y_x & z_x & 0 \\ x_y & y_y & z_y & 0 \\ x_z & y_z & z_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \cdot x_x + y \cdot y_x + z \cdot z_x \\ y \cdot x_y + y \cdot y_y + z \cdot z_y \\ z \cdot x_z + y \cdot y_z + z \cdot z_z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x_x & y_x & z_x \\ x_y & y_y & z_y \\ x_z & y_z & z_z \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x \cdot x_x + y \cdot y_x + z \cdot z_x \\ y \cdot x_y + y \cdot y_y + z \cdot z_y \\ z \cdot x_z + y \cdot y_z + z \cdot z_z \end{bmatrix}$$

Optimización de lados ocultos

Si un *chunk* contiene 1000 bloques con 6 lados compuestos de 2 triángulos y que hay $16 \cdot 16$, sea 256 *chunks* cargados, tenemos $1000 \cdot 6 \cdot 2 \cdot 256$, sea 3072000 triángulos en el procesador gráfico. Pero, la gran mayoría de esos triángulos están ocultos por otros. De manera mas general, un lado de bloque que está contra un otro bloque está oculto, y el lado bajo de los bloques con coordinada $y = 0$ no está oculto pero el personaje no necesita ver la parte inferior del terreno.



Una dificultad cuando se implementa esa optimización, es los lados de los *chunks* porque necesitan el acceso al los datos de los *chunk* vecinos. Y si la generación del terreno y la generación de los triángulos se hacen en una etapa atómica eso causaría una dependencia de vecinos infinita.

Para romper esa dependencia, los *chunk* se generan con diferentes etapas, que son *No Cargado*, luego *Cargado Sin Gráficos* y por fin *Cargado Con Gráficos*. Porque para transitar entre la etapa *Cargado Sin Gráficos* a la etapa *Cargado Con Gráficos*, los vecinos solo tienen que estar al lo menos a la etapa *Cargado Sin Gráficos*. Si alguno *chunk* vecino esta en la etapa *No Cargado*, solo genera su terreno y eso no requiere ningún dato de sus vecinos.

En el futuro, cuando se añadirá estructuras como arboles o ruinas y mazmorras en la generación, una etapa intermedia es requerida entre la etapa *No Cargado* y *Cargado Sin Gráficos*, y se llamara *Terreno Sin Estructura*. Porque poner una estructura sola puede distribuirse sobre 2, 3 o 4 *chunks*, y cuando se posiciona, afecta el terreno.

Fallas de implementación

Carga de *chunk*

Cuando un *chunk* se vuelve demasiado lejos del personaje, se descarga de la representación gráfica (liberación de memoria gráfica, transita de *Cargado Con Gráficos* a *Cargado Sin Gráficos*), pero si luego se vuelve de nuevo cerca del personaje, tiene que cargar de nuevo su representación gráfica (transitar de *Cargado Sin Gráficos* a *Cargado Con Gráficos*). Pero, en la implementación actual, hay una falla. Algunos de los *chunk* no hacen esa transición.

Causar el *bug* Alejarse en una dirección hasta que desaparece algunos *chunk*, y retrasar para causar la reaparición de esos *chunk*. Algunos no aparecen. Poner o quitar un bloque forza la generación de su representación.

Colisiones en los ángulos de bloques

Si el personaje esta perfectamente contra el ángulo de un bloque, puede pasar a través sin que ninguna colisión sea detectada. Ahora no tengo la explicación de lo que pasa, pero, tango una idea de dónde viene el problema. El *épsilon* de colisión influía el tamaño de la zona dónde no se detecta la colisión. Reducir el *épsilon* reduce ese tamaño hasta que es casi imposible de apuntarlo controlando el personaje.

Causar el *bug* Si se usa dos bloques para posicionarse precisamente al ángulo de un otro bloque, se puede pasar a través. Poner tres bloques al suelo con coordenadas $[0_x, 0_z]$, $[0_x, 1_z]$, $[1_x, 0_z]$, posicionarse en $[1_x, 1_z]$, andar en la dirección del bloque $[0_x, 0_z]$ hasta que los bloques $[0_x, 1_z]$ y $[1_x, 0_z]$ paran el personaje, romper esos dos bloques, y andar de nuevo en la dirección de $[0_x, 0_z]$.

Texturas con transparencia

Se puede poner bloques de cristal (*Glass*) y de agua (*Water*). El cristal tiene una textura con agujeros, $alpha \in \{0, 255\}$ y el agua tiene una opacidad, $alpha \in [1, 254]$.

Cristal En el caso de agujeros, el problema es que el *fragment shader* no ignora píxeles transparentes y inscribe sus profundidad en el *depth buffer*. Sea c_s el color de un píxel del la pantalla, d_s la profundidad de este píxel, c_f el color del fragmento computado y d_f su profundidad.

$$c_s := \begin{cases} c_s & \text{if } d_s < d_f \\ c_f & \text{if } d_f \leqslant d_s \end{cases} \qquad c_s := \begin{cases} c_s & \text{if } alpha = 0 \\ c_f & \text{if } alpha = 255 \end{cases}$$

Computación efectiva Computación deseada

Agua En el caso de la opacidad, la mezcla de dos colores no es una operación simétrica, pues hay que respectar el orden. hay que separar los triángulos del agua de los triángulos del terreno para poder ejecutar el *fragment shader* para el agua después de ejecutarlo para el terreno.

$$c_s := \begin{cases} c_s & \text{if } d_s < d_f \\ c_s \cdot c_f & \text{if } d_f \leqslant d_s \end{cases} \qquad c_s := \begin{cases} c_s & \text{if } d_s < d_f \\ c_s \cdot c_f & \text{if } d_f \leqslant d_s \end{cases} \text{ terrain}$$
$$c_s := c_s \cdot c_f \quad \text{water}$$

Computación efectiva Computación deseada

Causar el *bug* Poner bloques de cristal (tecla `3`) o agua (tecla `6`) y mirar el terreno a través. Todos los triángulos que están representados después del cristal o del agua no aparecen (ese orden es caótico 50 %).

Perspectivas futuras

Comunicación entre hilos y compartición de datos

Ahora, la implementación intenta de establecer una comunicación entre hilos basado en intercambio de mensajes con canales, pero no es el caso para algunos datos. Los *chunks* y el personaje estan almacenados en una estructura única (*World*) compartida entre todos los hilos, y está protegida de carreras de datos con algunos *mutex*. Esa manera de proceder conduce a un código mezclado, difícil de mantener y con poca perspectivas de evolución.

Se podría apoyar **únicamente** sobre mensajes a través de canales, pero impactaría la eficiencia (velocidad) del programa. Es por eso que la próxima etapa del proyecto es de encontrar un modelo de comunicación mas elegante y implementar tipos y métodos que imponen y facilitan el uso de este modelo, como capa de abstracción.

Entidades, Componentes y Sistemas

El modelo *E.C.S.* es excelente para los videojuegos. Las *entidades* son nodos para que se adjunte *componentes*, *componentes* son datos que representan propiedades (pertenencia a un clase de objeto) y un *sistema* es un comportamiento que se aplica a todas las *entidades* que tienen una cierta combinación de *componentes*.

Es el modelo que aplican *Unity*, *Godot*, *Bevy* y tantos otros... La razón es que ese modelo es muy potente y provee una capa de abstracción total sobre la organización del programa. El motor de juegos *Bevy* es implementado en *Rust* y su modelo *E.C.S.* se puede utilizar independiente del resto con la biblioteca *bevy-ecs* (`https://crates.io/crates/bevy-ecs`).

Pipeline gráfico moderno

Ahora, el programa utiliza directamente *OpenGL* que no soporte la personalización del pipeline (en realidad si, pero solo un poco y no es practico). Además, si quiero utilizar una otra interfaz gráfica como *Vulcan* o *DirectX*, tendría que desarrollar de nuevo casi todo lo del aspecto gráfico.

Existe una biblioteca nativa *Rust*, que se llama *wgpu*, que permite de programar un pipeline gráfico de manera agnóstica sobre la interfaz gráfica. Significa que el programa puede compilar (sin ningún cambio) para utilizar una interfaz que queremos. Permite también de compilar para el web (hasta *wasm*).

`https://crates.io/crates/wgpu`