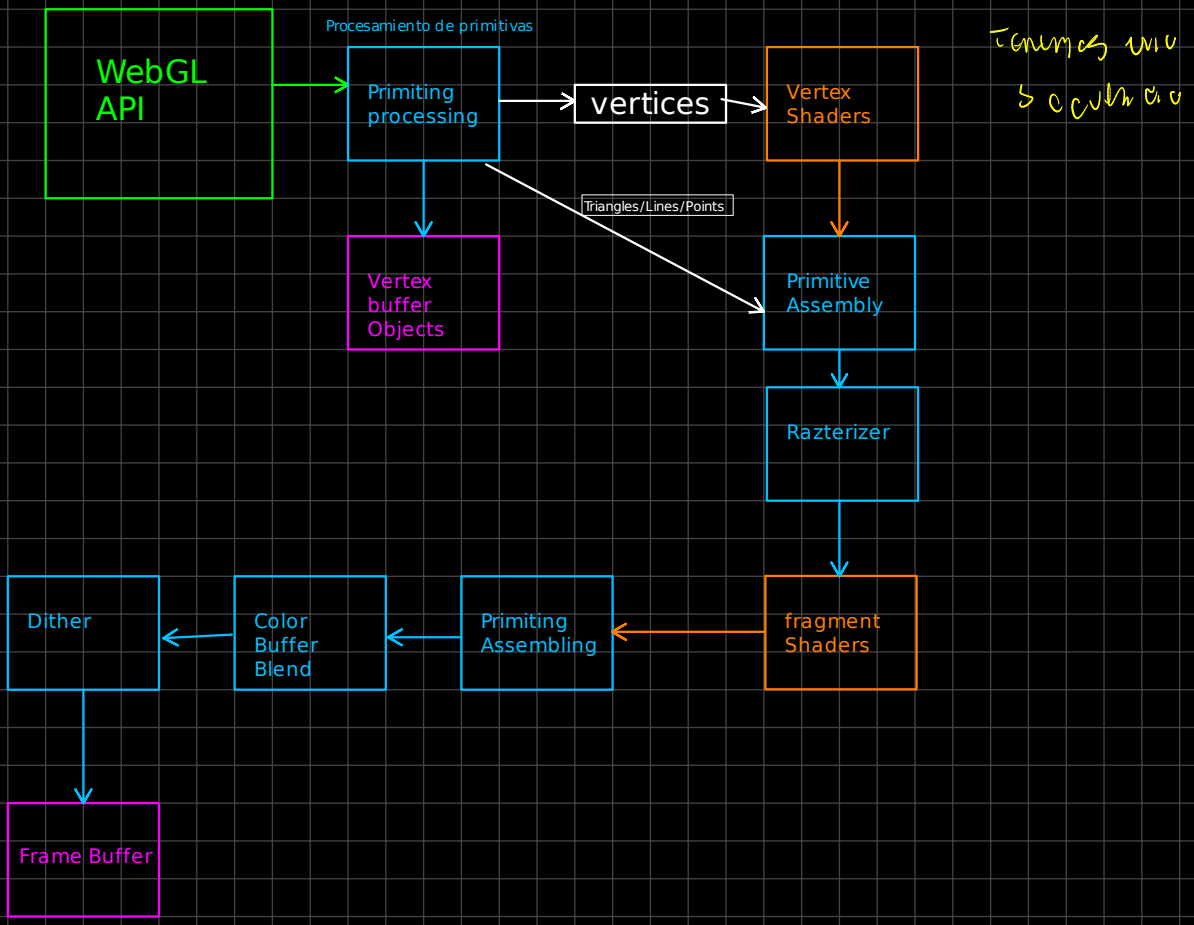
1. ¿Qué es el Pipeline Gráfico en el contexto de WebGL y Three.js?

El pipeline Grafico es el proceso por etapas de convertir datos 3D en una imagen 2D renderizada en pantalla, en el contexto de WebGL el pipeline Grafico sigue una secuencia estructurada, aunque podria decirse que THREEJS se abstrae gran parte de su complejidad.

En Resumen el Pipeline grafico es una secuencia de etapas para crear una representacion 2D rasterizada en una escena 3D.
Existen 2 grandes inputs de entradas para el pipeline grafico los:
Data streams (Por ejemplo: posicion, color, coordenadad uv, etc).
Comandos (Los comandos indican el como deben interpretar esos datos por ejemplo el Gl_Triangles permite dibujar triangulos a partir del conjunto de vertices definidos en el Data Streams)

Existian 2 tipos de pipeline grafico anteriormente el fixed ahora tenemos el programable.

2. ¿Cuáles son las etapas principales del Pipeline Gráfico?



3. ¿Qué función desempeña el vertex shader en el proceso de renderizado?

Su funcion principal es procesar los vertices de una malla geometrica para prepararlos antes de que se conviertan en pixeles en la pantalla.

Las funciones claves del vertex shaders son :

1 Transformacion de coordenadas
-Convierte las coordenadas 3D de cada vertice (del espacio del modelo) a coordenadas 2D en la pantralla (Espacio de pantalla).
-Usa matrices para aplicar
°Modelado
°Vista
°Proyeccion

Los Roles Fundamentales del vertex Shader son:
- Posiciona los vertices de una geometria 3D en la escena
- Aplica transformaciones de camara.
- Genera gl_Position para el que el rasterizador lo use posteriormente
- Produce elementos que luego aplica el Fragment Shader.

2 Pasar datos al Fragment Shaders
-Prepara y envia atributos como:
°Normales(paralluminacion)
°Coordenadas(para texturtas)
°Colores de vertices

3 Manipulacion de vertices(efectos Dinamicos)
-Puede deformar o animar vertices para crear efectos como:
°Olas en el agua
°Movimientos de personajes
°Efectos de vientos o explosiiones

En Resumen: el vertex shader es el Traductor Geometrico del renderizado donde este mismo convierte los triangulos en pixeles

4. ¿Cómo se define un vertex shader en GLSL, cuáles son las salidas mínimas necesarias (que valores debe retornar obligatoriamente)?

para definir un vertex shader en GLSL debemos de tener varias consideraciones importantes, tenemos un ejemplo completo y claro que nos permita analizarlo:

```
#version 300 es        //Esto es un indicativo que permite saber que se usa
                       //GLSL version 3.00 para OpenGL que es lo que usa
                       //WebGL 2.0, recomendable colocarla , de lo contrario
                       // se estaria usando la version de WebGL 1.0

in vec3 position;      //in es una palabra clave que me indica que es una variable
                       // de entrada por vertice, quiere decir que viene desde el
                       //buffers de vertices de la geometria
                       // vec3 tipo vector con 3 componentes
                       //position es el nombre de la variable representa la posicion
                       // del vertice  en coordenadas del modelo (espacio local).

uniform mat4 modelViewMatrix;

uniform mat 4 projectionMatrix;


void main() {
        //gl_Position es una variable reservada obligatoria, debe contener
        //la posicion final del vertice  en el espacio del clip
        gl_Position = projectionMatrix*modelViewMatatrix*vec4(position,1.0);
}
```

La salida minima y obligatoria que debe definir un vertex shader es gl_Position
Este es el programa de un vertex shader, es lo minimo e indispensable, para que el vertice se pueda dibujar.

En caso de estar usando WebGL 1.0 el programa seria el siguiente

```
atribute vec3 position;

uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;

void main(){
	gl_Position = projectionMatrix*modelViewMatrix*vec4(position,1.0);
}
```

5. ¿Qué datos se pueden pasar al vertex shader a través de atributos?

Los atributos (atribute en GLSL) son variables de entrada por vertice. Son los datos que se envian desde javascript/WebGL hacia el vertex shader, por cada vertice de una malla.
Cada vertice de una geometria pude tener asociado los siguientes atributos :

- -Posicion
- -Color
- -Normal
- -Coordenada de textuera(UV)
- -Tangentes
- -Pesos para animaciones entre otro.

En resumen son variables de solo lectura que se definen en el shader como:

```
attribute vec3 position;
attribute vec3 normal;
attribute vec2 uv;
```

6. ¿Qué es una variable uniform en el contexto de shaders?

Una variable uniform en un shader (GLSL) es una variable global y constante durante la ejecucion de un draw call, es decir :

- -no cambia mientras se dibuja un objeto
- -Es compartida por todos los vertices o fragmentos en ese momento

¿Para que sirven ?
Para enviar datos constantes desde el programa principal al shader por ejemplo:

- -Matrices de Transformacion
- -Color Global de un objeto
- -Tiempo para animaciones
- -Texturas
- -Parametros de Iluminacion(Posicion de la luz, intensidad, etc)

En Resume: Es una variable global y de solo lectura que se utiliza para pasar datos desde la aplicacion javascript a los shaders (vertex shaders o fragment shaders), mantienen el mismo valor para todos los vertices o pixeles dentro de un mismo dibujado

7. ¿Cuál es la diferencia entre un vertex shader y un fragment shader?

La principal diferencias entre un vertex shader y un fragment shader consiste en que el vertex shader trabaja sobre cada vertice de la figura geometrica triangular o geometria compleja, la principal tarea consiste en tomar los datos de entrada de cada vertice (Posicion, normal, coordenadas de textura uv, etc) y los transforma, su salida importante es el gl_Position pero tambien puede pasar otros datos del tipo varying para el fragment shader.
En cambio el fragment shader trabaja sobre cada fragmento que es lo que convertira en un pixel, recibe los datos del vertex shader del tipo varying y su salida tipica es un gl_FragColor(que consiste en el color final de un pixel)

En resumen:
vertex shader trabaja con los vertices
y el fragment shader trabaja con los fragmentos o pixeles de la pantalla

8. ¿Cómo se almacenan los valores de color de cada píxel en el fragment shader?

Un fragment shader se en carga de cada pixel y su salida obligatorio es una variable global gl_FragColor que es de tipo vec4, donde los 4 valores de rojo, verde, azul y alfa estan definido por un valor numerico de 0.0 a 1.0.
el color final del pixel queda definido por gl_FragColor

9. ¿Qué es un sampler2D y cómo se utiliza en un fragment shader?

Un sampler2D es un tipo de variable en GLSL que representa una textura en 2D, sirve para leer colores(o valores) desde una imagen desde una posicion determinada.
A que se refierer con una textura ? Basicamente es una imagen(jpg o png) que se aplica sobre una superficie 3D.

Entoces a que se refiere son sampler2D en GLSL?
Es una variable de tipo uniforme que le indica al shader: "Aca tenes una textura que podes consultar (samplear) como un mapa de colores en 2D"

¿Como se puede utillizar el sampler en un fragment shader ?
El procedimiento es sencillo el sampler2D se utiliza junto con unas coordenadas (u,v) para extraer el color especifico de una determinada coordenada, la linea de codigo mas importaten es la siguiente:

```
vec4 color = texture2D(myTexture,uv)
```

- myTexture es tu sampler2D(basicamente es la imagen que usas)
-uv es un vector de 2 coordenadas entre 0 y 1.
- El resultado es un vector de 4 coordenadas convalores RGBA.

Ejemplo en un Fragment shader:

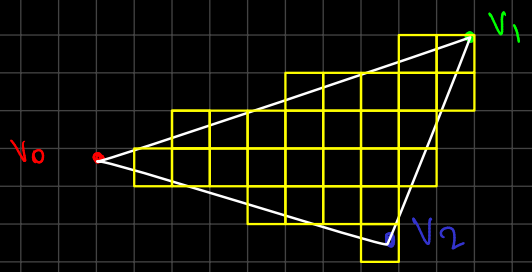
```
// Fragment shader
precision mediump float;

uniform sampler2D myTexture;  // la textura
varying vec2 vUv;           // coordenadas UV desde el vertex shader

void main() {
   vec4 texColor = texture2D(myTexture, vUv);// extrae el color de una textura
   gl_FragColor = texColor; // el píxel toma el color de la textura
}
```

10. ¿Cuál es el propósito del rasterizador en el Pipeline Gráfico?

El rasterizador es la etapa del pipeline grafico que convierte primitivas geometricas (triangulo) en fragmentos (es decir, pixeles candidatos a ser dibujados en pantalla).
Basicamente traduce las formas vectoriales en pixeles reales que la GPU puede colorear

Podriamos pensarlo como si se tratara de un puente entre el mundo 3D y la imagen visible 2D.

Supongamos que enviamos un triangulo



Interpolación
↓
"Calculo de valores intermedios, entre 2 o mas valores"

El rasterizador

1.- Rellena el Area del Triangulo

2.- Calcula que Pixel esta dentro

3.- Para cada pixel dentro de triangulo interpola el color entre los 3 vertices segun su posicion Relativa

4.- Genera un fragmento con ese color

El rasterizador No realiza lo siguiente :

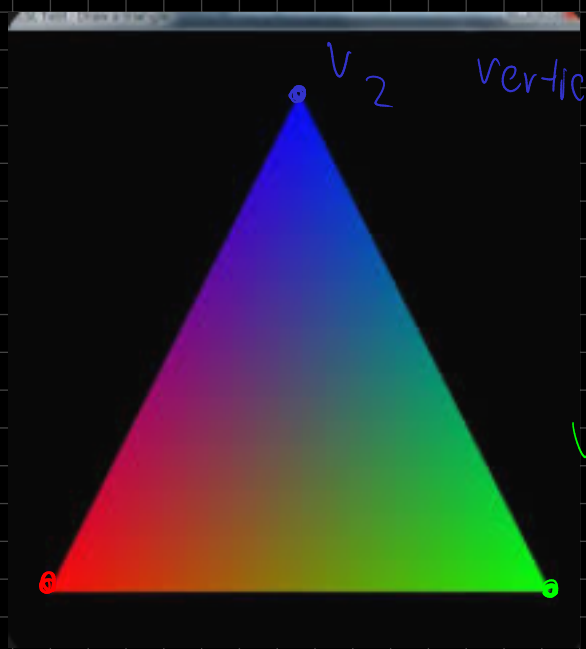
- No calcula el color Final del Pixel
- No aplica texturas ni iluminación

↓

Esto lo hace el fragment shader con la información interpolada que genero el rasterizador

Ejemplo Final al pasar el proceso del

vertex Shader → Rasterizador → Fragment Shader :



11. ¿Cuál es la función principal de la GPU en el Pipeline Gráfico?

La funcion principal de la GPU consiste en acelerar el procesamiento masivo y paralelo de tareas graficas.

- -Calcular transformaciones de vertices
- -Generar y procesar fragmentos(pixeles)
- -Ejecutar shaders(codigo grafico de a GPU)
- -Dibujar millones de elementos por segundo.

La GPU es responsable de procesar gran parte de las etapas del pipeline graifco, las funciones mas importantes son:

1.- Ejecuta shaders: Los shaders son pequeños programas escritos en GLSL que corren dentro de la GPU y se dividen en
vertex shader, fragment shader, entre otros.
La GPU ejecuta estos shaders en paralelo

2.-transformacion de veritices: Para cada vertice de una geometria, la GPU.

- -Aplica matrices(modelo, vista ,proyeccion)
- -calcula posiciones en pantalla.
- -Pasa informacion como normales,uv , colores, etc.

3.- Procesamiento de fragment shader: Después de la rasterizacion(hecha por la GPU) se crean fragmentos y la GPU.

- -interporla datos (como colores,uvs,normales,etc)
- -Ejecuta el fragment shader para calcular el color de cada pixel
- -Aplica efectos de iluminacion, transparecnia ,texturas

4.-Aplicacion de texturas y efectos visuales

- -Leer datos desde texturas (sampler2D)
- -Aplicar filtros
- -hacer combinaciones de colores con el fondo

5.-Paralelismo Masivo: Una GPU tiene millones de nucleos simples que pueden ejecutar millones de shaders al mismo tiempo, es ideal para:

- -renderizar millones de triangulos
- -Aplicar efectos por pixel
- -Simular, fluidos, particulas, etc.

En resumen

Función de la GPU	Explicación
Ejecutar shaders	Ejecuta programas para vértices y fragmentos
Hacer rasterización	Convierte triángulos en fragmentos
Aplicar transformaciones	Mueve objetos al espacio de cámara y pantalla
Calcular color por píxel	Iluminación, textura, transparencia
Trabajar en paralelo	Procesa miles de vértices y píxeles al mismo tiempo

12. ¿Cómo se extrae un valor de una textura en un fragment shader?

En un fragment shader (GLSL), se extrae un valor (generalmente un color) de una textura 2D usando la funcion
texture2D(mitextura,coordenadaUV)
A esto se lo conoce como samplear una textura
En conclucion se envia una imagen a la variable de tipo sampler2D junto con las coordenadas UV para poder enviarselo a la funcion texture2D(sampler2D,UV)
y al final gl_FragColor recibe el color obtenido de ese pequeño fragmento.
ejemplo:

```
precision mediump float;

uniform sampler2D miTextura;  // la textura
varying vec2 vUv;             // coordenadas UV interpoladas por el rasterizador

void main() {
    vec4 color = texture2D(miTextura, vUv);
    gl_FragColor = color;
}
```

13. ¿Qué son las variables varying en el contexto de los shaders?

Las variables varying en GLSL son usadas para pasar datos del vertex shader al fragment shader de forma interpolada.
Dicho de otra manera sirve para transferir informacion(como colores, coordenadas uv, o normales) desde cada verice hacia cada fragmento de manera automatica y continua.
Las varying son justamente variables compartidas entre los shaders.

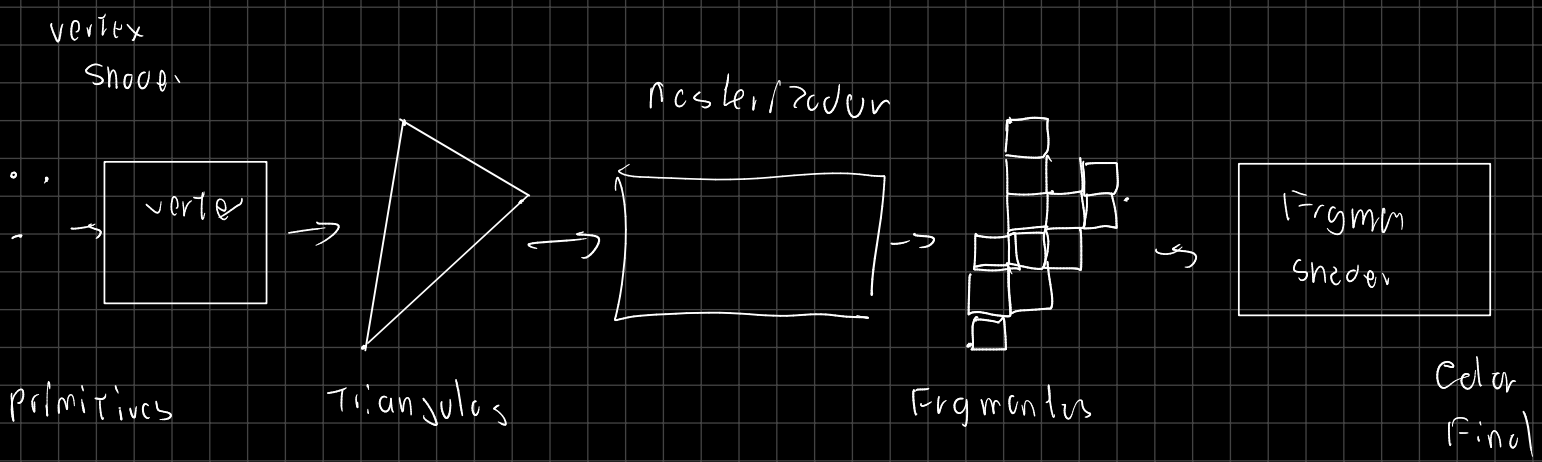
Concepto	Explicación
¿Qué es un `varying`?	Variable que conecta vertex shader con fragment shader
¿Quién la define?	El programador, en ambos shaders
¿Dónde se asigna?	En el vertex shader
¿Dónde se usa?	En el fragment shader
¿Cómo se transmite?	Interpolación automática por el rasterizador
¿Qué tipos de datos?	`vec2`, `vec3`, `vec4`, `float`, etc.
¿Para qué sirve?	Pasar datos como UVs, colores, normales, etc. por píxel

14. ¿Qué información se comparte entre el vertex shader y el fragment shader a través de las variables varying?

La informacion que se comparte entre los vertex shaders y el fragment shaders son:

-Coordenadas Uv	Para mapear tecturas correctamente
-Colores por vertices	Para crear degradados de color sin textura
-Normales	Para calcular iluminacionpor fragmento
-Posiciones del vertice	Para efectosespeciales (reflejos,profundiad,etc)
-Datos personalizados	Efectos Especiales (Desplazamientos,mascaras,etc)

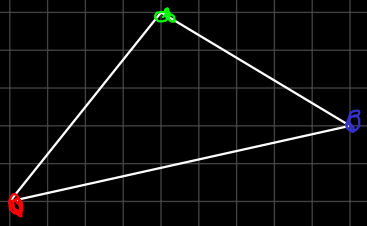
La interpolacion de valores sucede de la siguiente manera



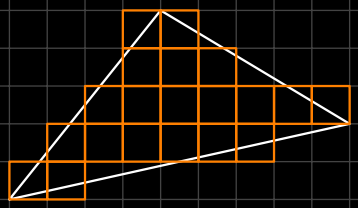
Supongamos que se tenga 3 vertices con 3 colores entonces, se realiza una interpolacion entre esos valores, se obtiene un degradado suave.

15. ¿Cómo se realiza la interpolación de valores de los vértices en un fragment shader?

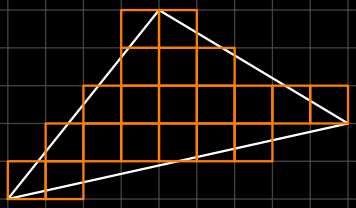
Primero se genera un triangulo con 3 vertices v0, v1 y v2.

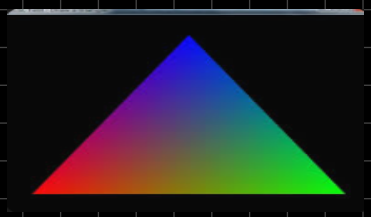


vertex shader



Rasterizador:Para cada pixel dentro del triangulo interpola los valores.





Fragment shader: Aplica el color final para cada pixel obteniendo un degradado suave

Es el proceso mediante el cual valores intermedios entre los vertices de una primitiva (como un triangulo), para que cada fragmento (pixel) tenga un valor adecuado segun su posicion dentro de la figura

Que valores se interpola?
cualquier valor pasado del vertex shader al fragment shader usando varying como:

- Coordenadas Uv
- colores de vertices
- Normales
- Posicion
- Datos perzonalizados

Matriz de Vista → Matriz de 4x4 → Transforma → objetos → Espacio de Mundo
↓
Espacio de Camara

Matriz de Modelado → Matriz de 4x4 → Transforma los vertices → Espacio de Modelo
↓
Espacio del Mundo

Matriz de Proyeccion → Mat 4x4 → transforma elementos → Espacio de Camara
↓
Espacio de Pantalla

16. ¿Qué es la matriz de vista?
17. ¿Qué es la matriz de modelado?
18. ¿Qué es la matriz de proyección?

16.- La matriz de vista es una matriz 4x4 que trnaforma los objetos desde el espacio del mundo al espacio de la camara (tambien llamado espacio de vista o view space).

Es como si movieramos todo el mundo para que la camara se quede quieta en el origen, mirando hacia adelante.

Analogia 1
Imaginá que estás en un set de filmación. En lugar de mover la cámara para hacer un paneo, se opta por dejarla fija, y en cambio, todo el decorado se mueve al revés frente a la cámara.

Analogia 2
En computación gráfica, en lugar de mover la cámara, movemos el mundo al revés. Por ejemplo, si la cámara está en la posición (10, 0, 0) mirando hacia el origen (0, 0, 0), la matriz de vista moverá todo el mundo -10 en X, y lo rotará para que lo que la cámara vea quede alineado con su eje Z.

¿Para qué sirve?
Porque los shaders y el pipeline necesitan saber cómo se ve la escena desde el punto de vista de la cámara. Para ello:
El objeto se transforma con la matriz de modelado (lo coloca en el mundo).
Luego con la matriz de vista (lo coloca respecto a la cámara).
Luego con la matriz de proyección (lo proyecta en pantalla).

17.- La matriz de modelado (model matrix o matriz de tranformacion del modelo) es una matriz de 4x4 que se utiliza en el pipeline grafico para transformar los vertices de un objeto desde su sistema de coordenadas local (espacio de modelo) al sistema de coordenadas del mundo(espacio global o espacio del mundo).

la matriz de modelado permite Trasladar, Rotar o Escalar un objeto, la combinacion de estas 3 matrices me permiten obtener la matriz de modelado.

matrixModel = mTras*mRot*mScale;
Ejemplo:
Podriamos tener un cubo en el espacio global , con lo cual podriamos aplicar una Traslacion(2,0,0) ; Escalado (3,3,3) ; Rotacion.z(90);
con lo cual la matriz de modelado modificiaria mi objeto en
una rotacion respecto al eje z , un escalado isotropico, y una traslacion de 2 unidades en el eje ex

18.- La matriz de proyecciones una matriz de 4x4 que transforma los puntos 3D (en el espacio de la camara) en puntos 2D listos para dibujarse en la pantalla (espacio de recortr o clip space)

El objetivo principal de la matriz de proyeccion es simular como los objetos se ven desde una camara, segun la perspectiva o en modo ortografico

Al aplicar la matrixModel y matrixView a un objeto todavia estan en en 3D, el objetivo de la matrixProjection es tranformar el objeto 3D a la pantalla 2D.

19. ¿Cómo se crea la matriz de transformación de la vista a partir de la posición y orientación de la cámara?

La matriz de vista esta asociado a los parametros de la camara, de que parametros estamos hablando ??

Se crea a partir de 3 vectores claves

-Posicion del ojo o de la camara: Este vector representa la coordenada (x,y,z) exacta donde se encientra la camara respecto a las coordenada del mundo.

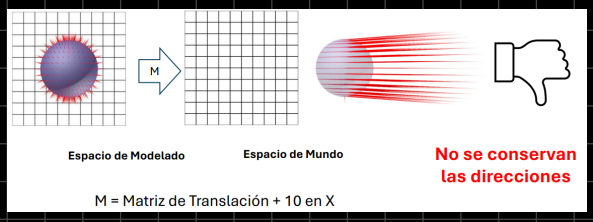
-Orientacion de la camara(punto objetivo o punto de mirada): Este vector define las coordenadas (x,y,z) de un punto en el espacio del mundo hacia el que la camara esta apuntando o mirando.

-vector Arriba: Este vector especifica la direccion arriba desde la perspectiva de la camara o relativa a la camara. Es importante para determinar la orientacion de la camara alrededor de su linea de vision, evitando que la la camara ruede o se inverita.

20. ¿Cómo se transforman las normales en un vertex shader para mantener su coherencia durante las transformaciones de modelo y vista?

Cuando se tiene una superficie particular necesitamos que las normales de las geometrias se mantengan perpendiculares a las mismas, para cada vertice.

Podemos aplicar varias transformaciones a una superficie por ejemplo las tipica transformaciones de modelado, si bien uno creeria que sus normales se mantinen, lo cierto es que no, porque en realidad sucede esta situacion.



Este tipo de incongruencias ocurren cuando aplicamos transformaciones de Escalado y de Traslacion.
La tranformaciones rotativas si deben afectar a las normales.

Nuestro problema consiste en mantener las normales de una superficie sin importar que tipo de transformacion estemos aplicando por ejemplo traslacion y escalado:

La solucion a este tipo de inconvenientes es utilizar una "Normal Matrix"

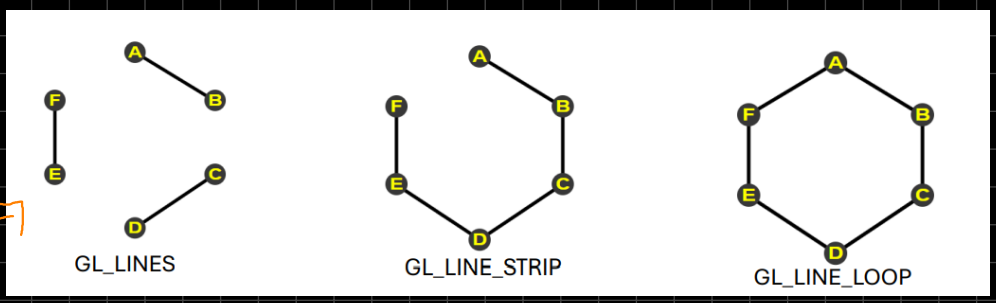
Usamos la matriz de normales para cada espacio puede ser el espacio del mundo o el espacio de camara.

NormalMatrix = ((MatrixModel)^-1)^T funciona para el espacio del mundo.
NormalMatrix = ((MatrixModelView) ^-1)^T funciona para el espacio de camara.

21. ¿Cuál es la diferencia entre gl.LINE_STRIP y gl.LINE_LOOP al dibujar líneas en WebGL?

La principal diferencia entre gl.LINE_STRIP y gl.LINE_LOOP, consiste en la conexion de una secuencai de vertices por ejemplo:
Secuencia de vertices A,B,C,D,E,F.

gl.LINE_STRIP Permite conectar los vertices en secuencia
gl.LINE_LOOP Permite hacer lo mismo con la unica diferencia que conecta el primer vertice con el ultimo.
Podemos apreciar un ejemplo practico:



22. ¿Cómo se dibuja una línea entre un par de vértices utilizando el modo gl.LINES De un ejemplo

Suponiendo que tengamos una secuencia de vertices {A,B,C,D,E,F}
Por cada par de vertices se dibuja una linea independiente .
podemos apreciar el ejemplo en la imagen de arriba.

23. ¿Cuál es el propósito de gl.TRIANGLE_FAN y cómo difiere de gl.TRIANGLES? Ejemplifique

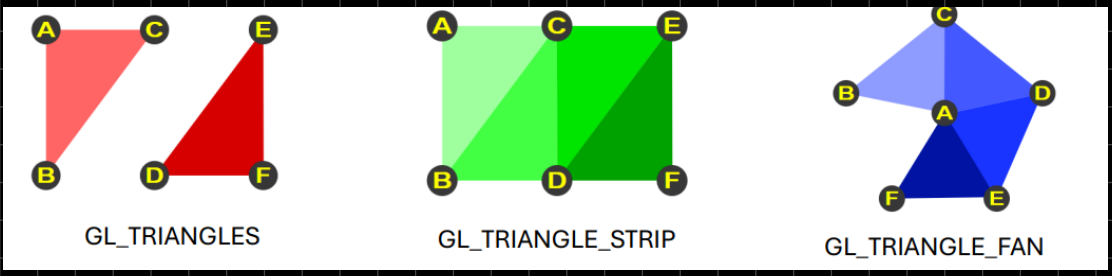
El proposito de gl.TRINAGLE_FAN es el crear triangulos a partir de un vertice central, suponiendo que tengamos la siguiente secuencia de vertices {A,B,C,D,E,F}, se crearan triangulos {A,B,C}, {A,C,D}, {A,D,E}, {A,E,F}

En cambio gl.TRANGLES permite generar triangulos independientes entre una secuancia de vertices {A,B,C,D,E,F}, generando un trinagulo {A,B,C} y {D,E,F}, ambos independientes respceto al otro.

24. ¿Qué ocurre cuando se utiliza gl.TRIANGLE_STRIP en lugar de gl.TRIANGLES? de un ejemplo

En gl.TRIANGLE_STRIP me permite generar una tira de triangulo en la cuales varios comparten vertices(permitiendo el ahorro de memoria) podemos pensarlo del siguiente modo, dado una secuencia de vertices {A,B,C,D,E,F} tendremos triangulos del tipo {A,B,C,},{B,C,D,},{C,D,E},{D,E,F}.

VEREMOS UN EJEMPLO GRAFICO PARA LAS PREGUNTAS 23 Y 24:



25. ¿Cuál es el papel del rasterizador en la transformación de primitivas en píxeles?

El rasterizador es un proceso intermedio de una secuencia de etapas del pipeline grafico puntualmente estamos hablando en la etapa intermedia entre el vertex shader y el fragment shader

vetex shader -> rasterizador -> fragment shader

El rasterizador me permite procesar datos de salida del vertex shader como primitivas(puntos, lineas, triangulos , etc), realizando una interpolacion entre los los datos de entrada para generar una salida que luego usara el fragment shader que me permite definir de un color determinado para cada pixel de la pantalla 2D.

1.-El vertex shader define:

gl_Position (posición del vértice)

varying vec3 color (color por vértice)

2.-El rasterizador:

Toma el triángulo proyectado.

Calcula qué píxeles están dentro del triángulo.

Para cada píxel, interpola los colores de los 3 vértices según su posición dentro del triángulo.

3.-El fragment shader:

Usa el color interpolado (vColor) y lo asigna a gl_FragColor.

26. ¿Cuál es el papel de los núcleos (cores) en una GPU y cómo se organizan?

Cual es el papel principal de los nucleos ??

- Podemos hablar sobre el procesamiento paralelo masivo: cada nucleo puede manejar una tarea especifica e independiente que el nucleo debe procesar. Cada hilo ejecuta una instancia de un conjunto de instrucciones(como calcular el color de un pixel o transformar un vertice), pero miles de hilos pueden ejecutarse al mismo tiempo gracias a la arquitectura paralela de la GPU

Los nucleos de una GPU se organizan principalmente en Streaming Multiprocessors(SM) que a su vez contienen contienen los cores o shaders cores.

27. ¿Qué es la memoria de video (VRAM) y cómo se diferencia de la memoria RAM convencional?

Que podemos decir sobre la memoria VRAM?

La memoria VRAM esta diseñada especificamente para manejar datos graficos, como texturas, modelos, iluminaciones, etc que la GPU necesita para generar imagenes o renderizar escenas.
La VRAM esta optimizada para transferir datos a la GPU a alta velocidad lo que permite que la GPU procese la informacion grafica de manera eficiente.

Que lo diferernia de la memoria RAM ?

Para empezar la RAM es la memoria principal del sistema que permite almacenar datos que la CPU necesita para realizar calculos y procesar informacion.

Su principal diferencia radica en que la VRAM es mucho mas rapida que la memoria RAM porque, lo que permite que la GPU procese los datos graficos de manera mas rapida que la CPU.

28. ¿Cómo se gestionan las variables uniforms en un shader y cuál es su propósito?

Las variables uniforms se definen en GLSL dentro del shader, pero se controlan y actualizandesde el programa principal(por ejemplo, en javascript con WebGL o THREEJS).
El proposito general de shader consisite en en mantener un valor constante TODOS USAN EL MISMO VALOR durante un renderizado.
en GLSL podriamos declara estas variables

```
uniform float time ;
uniform vec3 lightColor;
uniform mat4 modelViewMatrix;
```

En resumen gestionar una variable uniform es declarar la variable en el shader y enviarle un valor desde la CPU(tu codigo principal) . Ese valor se mantine constante para todos los vertices o fragmentos que se procesan en un solo Draw Call.

La Memoria VRAM es una memoria especifica de la GPU.
es mucho mas rapido que la RAM al procesar datos Graficos

Diferencia Puntual

VRAM es mucho más rapido que la Memoria RAM dado que permite procesar los datos graficos mejor que la CPU

El procesamiento paralelo me permite trabajar con multiples vertices o fragmentos en los cuales los multiples nucleos trabajan en multiples vertices o fragmentos.

29. ¿Cómo se logra el procesamiento paralelo en una GPU durante la ejecución de shaders?

La GPU logra procesamiento paralelo ejecutando miles de pequeñas unidades de computo (nucleos) que procesan multiples vertices o fragmentos simultaneamente. Esto es posible porque:

-Cada fragmento y cada vertice puede ser procesado de forma independiente.

-Entonces, la GPU distribuye

EL pipeline grafico permite esto, porque:
En el vertex shader cada vertice se procesa por separado.
En el fragment shader cada pixel se procesa por separado.

En resumen: No existe una dependencia directa entre un vertice y otro, o entre un pixel y otro.

30. ¿Cómo se puede optimizar el rendimiento en WebGL al minimizar el número de llamadas al pipeline gráfico?

Reducir el número de draw calls no es opcional si querés lograr una escena 3D fluida, especialmente con muchos objetos.En WebGL y Three.js esto se logra principalmente:

-Reutilizando recursos.

-Usando instanciación.

-Agrupando objetos.

-Optimizando texturas y shaders.

31. ¿Qué papel desempeña el atributo gl_Position en un vertex shader y cómo afecta el resultado final del renderizado?

Vamos a analizar que es el gl_Position es una variable predefinida en el lenguaje de sombreado GLSL,utilizada exclusivamente en los vertex shader
Es la unica salida obligatoria del vertex shader
Se espera que el vertex shader le asigne un valor a gl_Position,que representa la posicion del vertice en el espacio de recorte(clip-space)

El resultado final del renderizado es la imagen que termina en la pantalla, generada a partir de -posicion de los objetos
- su forma
-Sus colores
-Luces y texturas

En caso de que los vertices esten posicionados de manera incorrecta todo lo demas falla y por lo tanto produce un resultado indeseado.
En resumen:
Podés pensar en gl_Position como el "GPS" del pipeline: si la posición es incorrecta, nunca vas a llegar a destino (pantalla).

32. ¿Cuál es la diferencia entre el mapeo de texturas en coordenadas UV y coordenadas de proyección en un fragment shader?

Ambas tecnicas me se utilizan para mapear una textura sobre una superficie pero tienen enfoques, usos y resultados muy diferentes.

Que analogias podemos presentar para el mapeo de texturas en coordenadas uv?
Podemos pensa al mapeo UV como forrar un regalo con papel
Imaginemos que tenemos una caja en la cual tenemos que aplicar una determinado papel decorativo

- -Modelo 3D es la caja
- -El papel decorativo es la textura (nos referimos a la imagen 2D)
- -Se realiza una distribucion de papel de forraje para cada seccion de la caja

Esta es la analogia que se puede apreciar sobre las coordenadas UV.
En resumen cada vertice del objeto ya sabe que parte del papel (textura) le corresponde.

Analogia del mapeo por proyeccion
Imaginemos que tenemos un proyector de diapositivas que lanza una imagen sobre una estatua

- -El proyector es como el fragment shader con proyeccion
- -La estatua es el modelo 3D que no tiene ni idea de que tipo de textura va a estar encima de su suoperficie
- -La imagen proyectada cae sobre la superficie segun desde donde se proyecta.

Hay una serie de diferencias claves que podemos apreciar

Caracteristica	Mapeo UV	Mapeo por proyeccion
-Origen de coordenada	asignadas en el modelo	Calculadas en el shader
-necesita coordenada UV	SI	No necesariamente
-VIsual resultante	Es preciso y ajustado al modelo	Puede parecer una luz o una sombra
-Interpolacion	Es automativa entre vertices	Requiere correcion por perspectiva
- Aplicaciones tipicas	Personajes,ropa,objetos realistas	Sombras, luces proyectadas, efectos