Universidad de Huelva

Tema 5

Dibujando en el espacio

Índice

- El proceso de renderizado se lanza al ejecutar los comandos *glDraw…()*. A partir de aquí se buscan los vértices necesarios para generar las primitivas y se ejecuta el VertexShader sobre los atributos de cada uno de estos vértices.
- El comando *glDrawArrays(mode,first,count)* lanza un proceso de dibujo de primitivas utilizando la lista de vértices del VAO. El argumento *mode* indica el tipo de primitiva (GL_POINTS, GL_LINES, GL_TRIANGLES, …). El argumento *first* indica el índice del primer vértice. El argumento *count* indica el número de vértices afectados. A partir de este número de vértices y del tipo de primitiva se calcula el número de primitivas a dibujar (por ejemplo, un triángulo necesita tres vértices).

- El comando *glDrawElements(mode,count,type,indices)* es una función de dibujo alternativa que se basa en una lista de índices. Es decir, en vez de tomar los vértices de forma consecutiva en el VAO se utiliza una lista de índices de vértices. De esta forma es posible utilizar varias veces el mismo vértice sin tener que duplicarlo en el VAO.
- La lista de índices debe estar almacenada en un BufferObject de tipo GL_ELEMENT_ARRAY_BUFFER.
- El argumento *mode* indica el tipo de primitiva. El argumento *count* indica el número de índices que se van a utilizar. El argumento *type* indica el tipo de datos que utiliza el BufferObject (por ejemplo, GL_UNSIGNED_SHORT). El argumento *indices* es un puntero al primer índice de la lista y suele colocarse a NULL.

- Además de los comandos *glDrawArrays()* y *glDrawElements()* existen las versiones *glDrawArraysInstanced()* y *glDrawElementsInstanced()*. Estas versiones incluyen un último argumento que indica el número de veces que se va a ejecutar el dibujo de las primitivas (número de instancias).
- Existen también la versión *glDrawRangeElements()*, que añade información sobre los límites (mayor y menor) de los posibles valores de los índices, y la versión *glDrawElementsBaseVertex()*, que añade una constante que se suma a todos los índices.

- Las posiciones de los vértices se expresan en coordenadas (x,y,z,w).
- La última coordenada suele tomar el valor 1.0 y permite realizar transformaciones de desplazamiento (usando la cuarta columna de las matrices de transformación) y transformaciones de tamaño (en las proyecciones en perspectiva).
- Para realizar transformaciones en las coordenadas se utilizan matrices de transformación.

$$\begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ w_0 \end{bmatrix}$$

- A la entrada de la etapa de rasterización, la posición de los vértices se transforma a coordenadas homogéneas dividiendo por la coordenada w:

$$\begin{bmatrix} x_n \\ y_n \\ z_n \end{bmatrix} = \begin{bmatrix} x_c / w_c \\ y_c / w_c \\ z_c / w_c \end{bmatrix}$$

- Las coordenadas homogéneas permiten definir el trozo del espacio que se utilizará para generar el dibujo. Este recorte espacial se conoce como *clipping volume*, por lo que las coordenadas homogeneas se conocen también como coordenadas Clip.

- El recorte consiste en dibujar solo aquello que se ecuentra en coordenadas homogéneas entre -1 y 1.
- En realidad, el recorte es siempre entre -1 y 1 en las coordenadas X e Y, pero el tipo de recorte de la coordenada Z es configurable.
- Antes de aplicar el recorte se realiza también un proceso de selección de primitivas (*Culling*). Este proceso permite desechar primitivas y también es configurable.
- Cuando no se utilizan los shaders de teselado ni de geometría la entrada del proceso de rasterización corresponde a la salida del VertexShader. Por tanto, en estos casos la posición de los vértices a la salida del VertexShader debe estar descrita en coordenadas homogeneas.

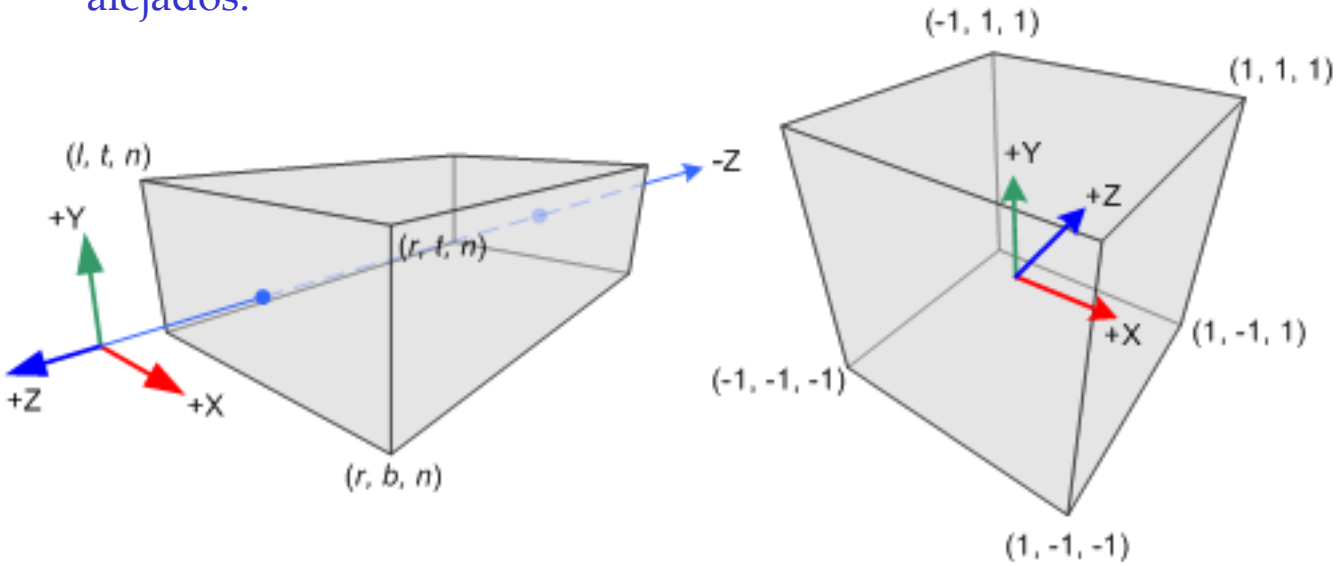
- Entradas predefinidas del VertexShader:
  - ***gl_VertexID***, contiene el índice del vértice que se está tratando en la ejecución del VertexShader.
  - ***gl_InstanceID***, contiene el número de instancia de la primitiva a la que pertenece el vértice.

    Cuando la función de dibujo utilizada es *glDrawArraysInstanced()* o *glDrawElementsInstanced()* cada primitiva se genera un cierto número de veces. La variable de entrada *gl_InstanceID* contiene el índice de la instancia que se está ejecutando en el VertexShader. De esta forma se pueden realizar transformaciones diferentes a los atributos de los vértices en función del número de instancia. Por ejemplo, se podrían ejecutar dos instancias y en la segunda transformar las primitivas para generar una imagen especular.

- Salidas predefinidas del VertexShader:
  - ***gl_Position***, debe almacenar la posición del vértice en coordenadas homogéneas.
  - ***gl_PointSize***, contiene el tamaño en pixels del punto a dibujar (si la primitiva fuera GL_POINTS o la propiedad PolygonMode tuviera el valor GL_POINT). Su uso es opcional. El valor por defecto de esta salida se puede configurar por medio del comando *glPointSize()*, que inicialmente tiene el valor 1.
  - ***gl_ClipDistance***, es un array que debe almacenar la distancia a los planos clip. Esta salida se utiliza en caso de querer personalizar el proceso de clipping. Su uso es opcional.
  - ***gl_CullDistance***, es un array que debe almacenar la distancia a planos cull. Se trata de una característica incluida en la versión 4.5 que permite personalizar la etapa de culling. Su uso es opcional.

- Aunque la salida del VertexShader debe estar descrita en coordenadas homogéneas, lo normal es que la posición de los vértices (introducida como uno de los atributos de entrada en el VertexShader) se describa en el sistema de coordenadas del modelo. Por tanto, el programa del VertexShader debe incorporar una transformación entre un sistema de coordenadas y otro.
- Esta transformación debe convertir el volumen visible del modelo (es decir, la zona del modelo que aparecerá en la imagen) en el volumen clip (es decir, un volumen normalizado entre -1 y 1).
- Esta transformación se conoce como ***Proyección***. Se suelen utilizar dos formas: proyección ortográfica o proyección en perspectiva.

- Proyección ortográfica:
  - Consiste en transformar un volumen de forma rectangular en el volumen Clip. Esto supone un cambio de escala en las diferentes coordenadas (X,Y,Z) para adaptarse al sistema de coordenadas homogéneo. La transformación es muy sencilla, pero provoca que los objetos se vean del mismo tamaño tanto si están cerca como si están alejados.



- Proyección ortográfica
  - La matriz de proyección en este caso es la siguiente:

$$P = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

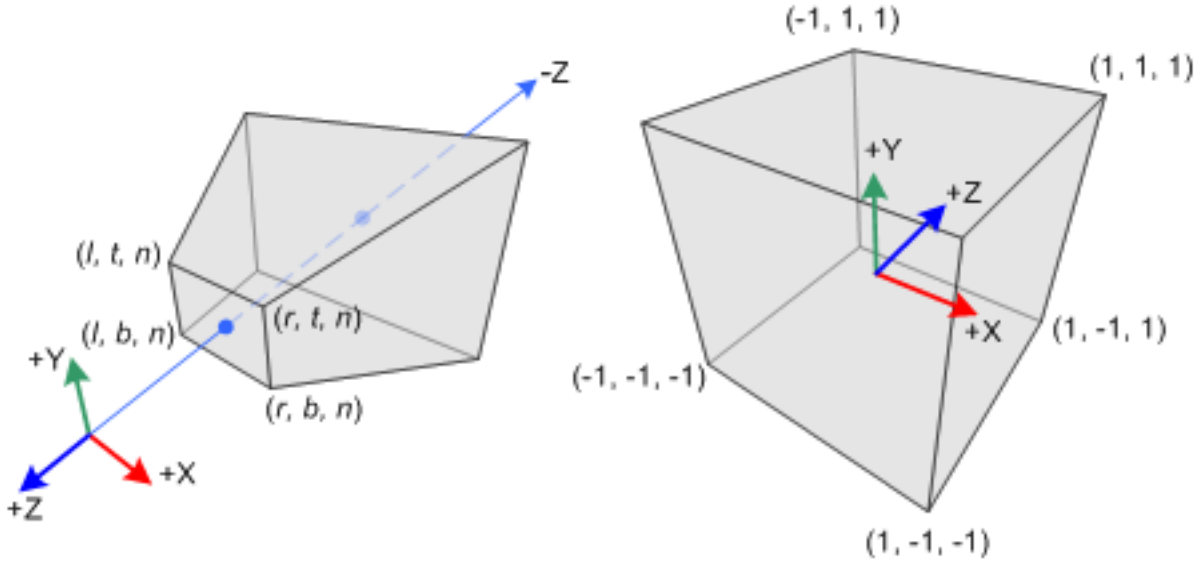
- Proyección ortográfica
  - Si el clipping volume es simétrico (r = -l y t = -b) la matriz queda

$$P = \begin{bmatrix} \frac{1}{r} & 0 & 0 & 0 \\ 0 & \frac{1}{t} & 0 & 0 \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  - Se puede generar la matriz de proyección utilizando la función de la librería GLM:

glm::ortho(left, right, bottom, top, near, far)

- Proyección en perspectiva:
  - Consiste en transformar el espacio que se ve desde el observador con un cierto ángulo de visión (*fov-field of vision*) en el volumen Clip. Esta transformación provoca que los objetos más cercanos se vean de mayor tamaño que los objetos más lejanos. La forma del volumen transformado es un tronco de pirámide (en inglés, *frustum*).



- Proyección en perspectiva
  - La matriz de proyección que genera la transformación en perspectiva es la siguiente:

$$P = \begin{bmatrix} \frac{2 \cdot n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2 \cdot n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & \frac{-2 \cdot f \cdot n}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

- Proyección en perspectiva
  - Si el volumen Clip es simétrico (r = -l y t = -b) la matriz queda

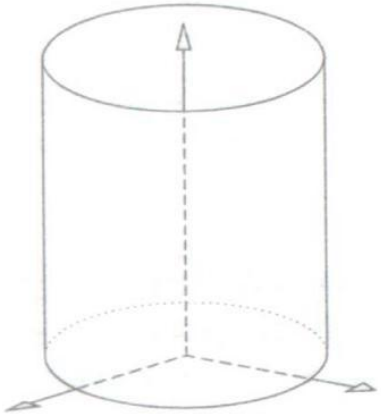
$$P = \begin{bmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & \frac{-2 \cdot f \cdot n}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

  - Esta matriz se puede generar con funciones de la librería GLM:

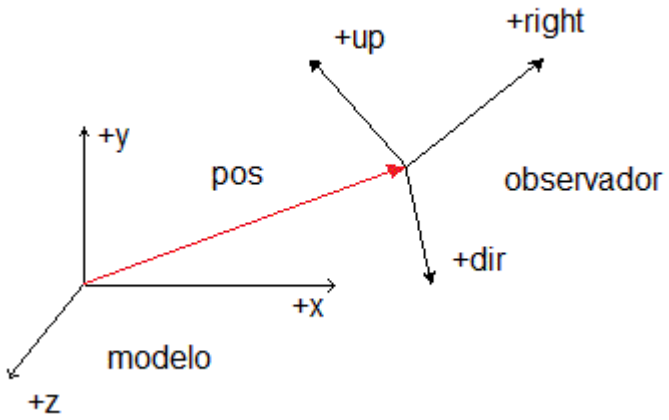
glm::frustum(left, right, bottom, top, near, far)

glm::perspective(fovy,aspect,near,far)

- Generalmente los atributos de los vértices de los objetos se expresan en un sistema de coordendas propio del objeto (sistema de coordenadas local). Este sistema de coordenadas suele tener su origen en un punto característico del objeto (por ejemplo, en su centro) y sus ejes de coordenadas suelen ser direcciones características del objeto.



- Para realizar la proyección, las posiciones de los vértices deben estar descritas en el sistema de coordenadas del observador. Por tanto, es necesario realizar una transformación entre el sistema de coordenadas local y el sistema de coordenadas del observador.



- Normalmente esta trasformación se realiza en dos pasos. En primer lugar las posiciones de todos los objetos se transforman a un sistema de coordenadas fijo del modelo. De esta forma las posiciones de todos los objetos quedan expresadas en un mismo sistema de coordenadas. Para transformar las coordenadas locales en coordenadas de modelo se utiliza una matriz de transformación conocida como *Model*.
- El segundo paso es transformar estas coordenadas de modelo al sistema de coordenadas del observador. Para esto se utiliza una matriz de transformación conocida como *View*.
- La transformación completa se conoce como *Model-View-Projection*.

- Si queremos que realizar una traslación, la matriz de transformación debe ser

$$M_{Tr} = \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

De esta forma

$$v_e = M_{Tr} \cdot v_0 = \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} = \begin{bmatrix} x_0 + a \\ y_0 + b \\ z_0 + c \\ 1 \end{bmatrix}$$

- Si queremos que realizar una transformación de escalado sobre el modelo, la matriz debe ser

$$M_{Sc} = \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

De esta forma

$$v_e = M_{Sc} \cdot v_0 = \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} = \begin{bmatrix} a \cdot x_0 \\ b \cdot y_0 \\ c \cdot z_0 \\ 1 \end{bmatrix}$$

- Las rotaciones afectan a las tres primeras filas y columnas de la matriz. Por ejemplo, para girar un ángulo α sobre el eje Z la matriz debe ser

$$M_{Rot} = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- De esta forma

$$v_e = M_{Rot} \cdot v_0 = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} = \begin{bmatrix} x_0 \cdot \cos(\alpha) - y_0 \cdot \sin(\alpha) \\ x_0 \cdot \sin(\alpha) + y_0 \cdot \cos(\alpha) \\ z_0 \\ 1 \end{bmatrix}$$

- La matriz de rotación sobre un eje arbitrario (a_x, a_y, a_z) es la siguiente:

$$c_\theta = \cos(\theta) \quad s_\theta = \sin(\theta)$$

$$M_{Ra}(\theta) = \begin{bmatrix} a_x^2 + c_\theta(1-a_x^2) & a_x a_y(1-c_\theta) - a_z s_\theta & a_x a_z(1-c_\theta) + a_y s_\theta & 0 \\ a_x a_y(1-c_\theta) + a_z s_\theta & a_y^2 + c_\theta(1-a_y^2) & a_y a_z(1-c_\theta) - a_x s_\theta & 0 \\ a_x a_z(1-c_\theta) - a_y s_\theta & a_y a_z(1-c_\theta) + a_x s_\theta & a_z^2 + c_\theta(1-a_z^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Se pueden generar facilmente las matrices de transformación por medio de funciones de la biblioteca GLM.

glm::translate(glm::mat4 M, glm::vec3 disp)

glm::scale(glm::mat4 M, glm::vec3 scale)

glm::rotate(glm::mat4 M, float angle, glm::vec3 axis)

- Estas funciones generan una modificación de la matriz *M* multiplicándola por la derecha por la matriz de desplazamiento, escalado o rotación correspondiente.

$$M_{new} = M_{old} \cdot M_{Tr}$$

- Ejemplo de VertexShader que contiene la transformación Model-View-Projection

```
#version 400

in vec3 VertexPosition;

uniform mat4 MVP;

void main()
{
    gl_Position = MVP * vec4(VertexPosition, 1.0);
}
```

- El proceso de clipping elimina los vértices que se encuentran fuera del volumen de recorte, es decir, los vértices cuyas coordenadas homogéneas se encuentren fuera del intervalo (-1,1).
- Tras ese proceso se produce una nueva transformación para calcular a que pixel corresponde cada vértice. Para eso se utiliza la información sobre el *Viewport*, es decir, el tamaño en píxeles de la imagen a generar.
- Para la coordenada X, el valor -1 se transforma en 0 y el valor +1 se transforma en *Width* (anchura de la imagen). Para la coordenada Y, el valor -1 se transforma en 0 y el valor +1 se transforma en *Height* (altura de la imagen). La posición del píxel se calcula como la parte entera de las coordenadas Viewport.

- Con respecto a la coordenada Z, por defecto no se utiliza. De esta forma, el comportamiento por defecto consiste en dibujar las primitivas en el orden en el que se van ejecutando, sin tener en cuenta si un determinado triángulo se encuentra delante o detrás de otro dibujado con anterioridad.
- Para considerar la profundidad (y de esa forma evitar que un triángulo que está alejado se dibuje tapando un triángulo más cercano) hay que activar la opción GL_DEPTH_TEST.

glEnable(GL_DEPTH_TEST)

- En el proceso de construcción de la imagen, el color de cada pixel se almacena en una estructura llamada *ColorBuffer*. El *ColorBuffer* es un array de píxeles, es decir, una representación de la imagen.
- Cuando se activa la opción DEPTH_TEST comienza a utilizarse una segunda estructura denominada *DepthBuffer*. Este buffer es también un array de píxeles, pero en vez de la información del color lo que se almacena en cada posición es la profundidad (coordenada Z) del punto representado en ese píxel.
- La información de profundidad se almacena como un valor de 16 bits. Es decir, el valor mínimo se describe como 0 y el valor máximo se describe como 2^{16}-1, es decir, 65.535.

- En la etapa de rasterización, antes de lanzar el FragmentShader sobre un píxel se estudia el contenido del buffer de profundidad para ese píxel. Si el píxel ya se había dibujado anteriormente como parte de otra primitiva la información del buffer corresponde a la profundidad del píxel anterior.
- Por defecto, si el nuevo píxel está más cerca entonces se lanza el FragmentShader pero si si el pixel a dibujar está más lejos que el que ya se había dibujado entonces no se lanza el FragmentShader.

- En realidad, la función para decidir si se dibuja o no el nuevo píxel se puede configurar con el comando **glDepthFunc()**.
- Los valores aceptados para este comando son GL_NEVER, GL_LESS, GL_EQUAL, GL_LEQUAL, GL_GREATER, GL_NOTEQUAL, GL_GEQUAL y GL_ALWAYS.
- La opción por defecto es GL_LESS, es decir, el nuevo píxel solo se dibuja si su valor de profundidad es estrictamente menor al valor del pixel anterior. Esta opción suele producir un problema conocido como *z-fighting* cuando se dibujan dos objetos que están en contacto.
- La forma más sencilla de evitar problemas de *z-fighting* es utilizar la función GL_LEQUAL e intentar dibujar primero los objetos situados al fondo.

- El problema de z-fighting se debe a que los valores de profundidad no se almacenan en formato flotante, sino como valores enteros en una escala de 16 bits.
- Si utilizamos la proyección en perspectiva, el valor mínimo de la componente Z se obtiene para la distancia *near* y el valor máximo para la distancia *far*, pero los valores intermedios no responden a una interpolación lineal. Esto se debe a que la proyección en perspectiva no es una transformación lineal.

- Después de la proyección en perspectiva, el valor normalizado de la componente Z queda

$$z_n = \frac{z_c}{w_c} = \frac{f+n}{f-n} + \frac{2 \cdot f \cdot n \cdot w_e}{z_e \cdot (f-n)}$$

- Para $z_e = -n$ y $w_e = 1$, $z_n = -1$. Para $z_e = -f$ y $w_e = 1$, $z_n = +1$.
- Para cuantizar z_n entre 0 y 65535 la fórmula sería

$$z_{vw} = 65535 \cdot \left(0.5 + 0.5 \cdot \frac{f+n}{f-n} + \frac{w_e}{z_e} \cdot \frac{f \cdot n}{f-n} \right)$$

- Despejando el valor de z_e queda

$$\frac{z_e}{w_e} = f \cdot n \Big/ \left(\frac{f-n}{65535} \cdot z_{vw} - f \right)$$

- El valor z_{vw}=0 se obtiene para z_e = -n .
- El valor z_{vw} = 65535 se obtiene para z_e = -f .
- Si tomamos n=0.2 y f=400,
  - el valor z_{vw} =1 se obtiene para z_e = - 0.200003050325
  - el valor z_{vw} =65534 se obtiene para z_e = - 388.16
- Si tomamos n=0.01 y f=1000,
  - el valor z_{vw} =1 se obtiene para z_e = - 0.01000015
  - el valor z_{vw} =65534 se obtiene para z_e = - 395.9

- Esto quiere decir que los valores pequeños de z_{vw} tienen mucha precisión en cuanto a la profundidad pero que los valores muy lejanos tienen una precisión muy baja.
- Para el caso n=0.01 y f=1000, todo lo que se encuentre a una distancia entre 396 y 1000 tendrá el mismo valor de profundidad. Por tanto, el test de profundidad provocará muchos errores en ese rango.