Efectos visuales de naturaleza: agua y fuego

PID_00249856

Rafael Pérez Vidal

Tiempo mínimo de dedicación recomendado: 3 horas

Universitat Oberta de Catalunya

Índice

Introducción

El grafismo por ordenador pasa por comprender perfectamente cómo funciona la tarjeta gráfica moderna de hoy en día. Una de las bibliotecas más extendida por los fabricantes es OpenGL. Os proponemos crear el efecto visual del agua de un lago, que está llena de detalles. Esto nos permite adentrarnos en cómo funciona el OpenGL, su máquina de estados y los *shaders*.

Empezaremos por enumerar la tecnología que hay en el uso de la biblioteca OpenGL y cómo se relaciona con la tarjeta gráfica. Definiremos los puntos más utilizados de la máquina de estados de OpenGL.

Una vez aprendidos, pasaremos a realizar un efecto visual concreto. Nos basaremos en el efecto de reflexión y refracción que podemos encontrar al intentar imitar la superficie del agua, aunque hay varias maneras de resolver este efecto.

Para este reto es necesario que se tengan conocimientos de programación en C ++ y una buena base de conocimientos matemáticos. Hace falta también tener mucha imaginación para que podáis ir construyendo vuestros propios efectos visuales.

Como compilador, trabajaremos sobre una base de proyecto de Visual Studio C++, pero como el código es C++ universal, podéis utilizar cualquier otro compilador de C++.

La biblioteca básica que emplearemos para realizar gráficos será SDL, Simple DirectMedia Layer (web oficial), y otras complementarias de OpenGL, como GLEW (web oficial) y GLM (web oficial).

Enlaces de información y conocimiento de OpenGL:

- Reference Card
- Especificació complerta de Open GL 3.3
- Llenguatge GLSL per Shaders

1. Introducción a OpenGL

OpenGL es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que producen gráficos 3D. Fue desarrollada originalmente por Silicon Graphics Incorporated (SGI). OpenGL significa Open Graphics Library, cuya traducción es 'biblioteca de gráficos abierta'.

Esta especificación está mantenida por Khronos Group. Es un consorcio industrial financiado por sus miembros para la creación de API estándares y abiertas, y libres de pagos por uso. Los fabricantes de tarjetas gráficas son los desarrolladores de las bibliotecas OpenGL.

Contenido complementario

Para saber el origen de OpenGL y su evolución, tenéis aquí un artículo resumen muy interesante:

Historia de los gráficos

Entre sus características, se puede destacar que es multiplataforma (hay incluso un OpenGL ES para teléfonos móviles) y que la gestión de la generación de gráficos 2D y 3D por hardware ofrece al programador una API sencilla, estable y compacta. Al ser multiplataforma, podemos encontrar OpenGL para muchas plataformas (Linux, Unix, MacOS, Microsoft Windows, etc.).

Para utilizar la API OpenGL, hay que tener conocimientos de C++, álgebra lineal, geometría y trigonometría.

1.1. *Core-profiler* frente a *immediate mode* (antiguo y desfasado)

Hay dos maneras de trabajar con OpenGL: *core-profiler* e *immediate mode*. Este segundo está desfasado y fuera de uso. Cuando busquéis información, debéis tener en cuenta la versión de OpenGL que queréis usar. En este PLAN trabajaremos con modo *core-profiler* de la versión 3.3. Las futuras versiones son ampliaciones que no cambian el *core*. Se denominan extensiones a OpenGL.

Aquí tenéis la documentación necesaria para utilizar OpenGL. Es la documentación oficial (web).

1.2. Máquina de estados

En sí, OpenGL es una larga máquina de estados, una colección de variables que definen cómo tiene que operar. Enumeraremos algunos de los aspectos que tiene la API.

- Hay funciones *state-setting*: para fijar una opción.
- Hay funciones *state-using*: para realizar una función tal como estén los *state-settings*.
- OpenGL está hecho en C, no en C++.
- Hace uso de *struct* como «objetos».

- No tiene funciones sobrescritas, adaptativas de parámetros, sino que hay una función para cada manera de acceder a parámetros

Para más concertación, consultad la explicación siguiente (web).

1.3. Creación de la ventana y contexto

Vamos a empezar a crear el contexto para trabajar con OpenGL. Antes que nada, hay que crear una ventana. Utilizaremos la biblioteca SDL.

Contenido complementario

Alternativamente, también podemos hacer uso de la biblioteca GLFW, de características similares. Podemos seguir este tutorial.

Reto 1

Utilizando la biblioteca SDL y siguiendo el tutorial de la web Lazy Foo, vamos a crear una ventana y el contexto de OpenGL necesario.

Tendréis que tener en cuenta la inclusión de la biblioteca GLEW para localizar las funciones. GLEW es una biblioteca localizadora de funciones. Cada fabricante hace lo que quiere y coloca las funciones donde quiere. Esta biblioteca permite automatizar la búsqueda. Si no, tendríamos que hacer un trozo de código para localizar cada función de OpenGL con wglGetProcAddress.

Fijaos bien en el tutorial de LearnOpenGL para no dejaros ningún paso.

Reto 2

Haced que la pantalla cambie de color alternativamente y haciendo unos cambios de colores según un patrón sinusoidal.

Funciones OpenGL que nos harán falta:

- glViewport
- glClearColor
- glClear

2. El primer triángulo

Vamos a ver ahora todos los conceptos iniciales de OpenGL que hay que saber llevando a cabo el primer reto, donde usamos ya OpenGL para pintar.

Reto 3

Dibujad en pantalla un triángulo con tres vértices.

Fijaos en el capítulo «Hello Triangle» de este enlace.

A continuación, enumeramos los conceptos que tendremos que asimilar con el reto:

- *Graphics pipeline* de OpenGL.
- *Shaders* -> OpenGL Shading Language (GLSL).
  - Variables, parámetros y compilación.

- Primitivas de dibujo:
  - GL_TRIANGLES, GL_POINTS, GL_LINE_STRIP...

- VBO, *vertex buffer objects*.
- VAO, *vertex array objects*.
  - Compendio de VBO + configuración de atributos.

- EBO, *element buffer objects*.
  - Se ha de asociar dentro del VAO, al igual que el VBO.

- Uso de la función glDrawElements como primitiva de pintado.

Modificación: haced uso de glPolygonMode para pintar en modo *wireframe*.

3. *Shaders* y texturas

3.1. Conceptos básicos del lenguaje de los *shaders*

Un *shader* es un pequeño programa que se ejecuta en los *cores* de la tarjeta gráfica. Hay que enviar el *shader* hacia la tarjeta gráfica usando nuestro código de programa principal de ejecución en la CPU. Habrá que compilarlo antes de poderlo usar.

La sintaxis del *shader* es similar a la de C, pero con cosas hechas a medida para gráficos 3D, como vectores y matrices.

Hay que tener claro el flujo de datos entre *shaders*. Repasad el capítulo «Hello Triangle» de este enlace.

Fijaos en el capítulo «Shaders» de este enlace.

A continuación, enumeramos los conceptos que tendremos que asimilar con este tutorial:

- #version, in, out, uniform, main()
- *Vertex attribute*:
  - límit de *in*, 16 mínim GL_MAX_VERTEX_ATTRIBS
  - tipos de variables: *float*, *int*, *double*, *uint*, *bool*
  - vectores: vecN, bvecN, ivecN, uvecN, dvecN
  - .x .y .z .w .r.g.b.a .s.t.p.q
  - *swizzing*

- *Vertex shaders* recibe los IN del vértice *data* y hay que decirle cómo están organizados estos datos desde el programa de la CPU:
  - layout (location = 0)
  - gl_Position

- Fragmento *shader* requiere un OUT de tipo vec4, que es el color.
- Envío de cosas entre *shaders*: IN y OUT, con el mismo tipo y nombre.
- uniforms
  - Son las globales de los *shaders*.
  - Una vez iniciados, se quedan con este valor hasta que lo hacemos variar.
  - No hay que declararlo en todos los *shaders*; solo en aquellos de los que se hace uso.

Contenido complementario

Estas variables que empiezan por ***gl_*** son varibles Build-in del propio OpenGl. Existen de distintos tipos y en cada Shader encontraremos de diferentes.

Mira el siguiente enlace de varible Build-in para conocerlas todas.

Es recomendable no definir ninguna varible de Shaer que empiece por ***gl_***, ya que podría ocasionar conflicto con alguna variable interna que tubiera el mismo nombre por casualidad.

3.2. Interactuación con nuestro programa

Reto 4

Construid una clase Shader para gestionar los *shaders*. Esta es su definición básica:

```
class Shader
{
public:
        Shader ();
        ~Shader ();
        void init ( const  GLchar *filename);
        GLuint getID ()        { return mIDprogram; };
        void Use ()             {
                if (mAllOK)
glUseProgram (mIDprogram); }   ;
        void deleteProgram ()            {
                 if (mAllOK)
glDeleteProgram(mIDprogram) ;   } ;
  private:
        GLuint            mIDprogram;
        bool mAllOK;
};
```

Una vez hechas las conexiones y los *shaders*, como mínimo el *vertex shader* y el *fragment shader*, podremos compilarlos y juntarlos para crear un programa, como ya vimos en capítulos anteriores. Como este proceso es siempre el mismo, vamos a hacer el reto 4 para tenerlo todo más ordenado.

Una vez construido, debemos tener claro cómo interactuar con el *shader*: cómo pasarle datos, cómo dar valores a los *uniforms* y cómo activar y desactivar los atributos que del VAO entrarán en él.

Contenido complementario

En la documentación oficial de OpenGL, hay unos tutoriales que nos pueden ajudar a comprender mejor los *shaders*. Son los Clockworkcoders Tutorials.

Hay que tener claro el uso de las siguientes funciones OpenGL para hacer el siguiente reto, que es múltiple:

- glGetAttribLocation
- GlGetUniformLocation
- glUniform
- glVertexAttribPointer
- glEnableVertexAtribArray

Reto 5

1. Pintad un triángulo en el centro de la pantalla dándole a cada vértice un color diferente.

2. Cambiad de color un triángulo con valor sinusoidal calculado en CPU y pasado a GPU por Uniform.

3. Moved a derecha e izquierda sinusoidalmente el triángulo especificando un *offset* por Uniform al *vextex shader*.

EXTRAS

- Moved un ristra de diez triángulos de tonos de colores diferentes como si fuera un gusano a derecha e izquierda. Haced uso de Uniform para darle un valor de desfase.

- Dibujad la posición del triángulo en función de la posición del ratón. Recoged las coordenadas del ratón y pasadlas por Uniform al *shader*, que calculará la posición donde pintar el triángulo.

- Encended el triángulo de color blanco con Fade a un color base cuando se haga clic con el ratón. Pasad el clic del ratón por Uniform al *shader* y que en el *fragmentshader* se realice el cambio de color. La función seno se tiene que hacer en el *fragmentshader*, y el tiempo necesario para el cálculo tiene que venir por Uniform desde la CPU.

3.3. Añadiendo texturas

Las texturas hacen que nuestros polígonos pasen de tener un aspecto plano de color a una vistosidad más amplia. Hemos de pensar que las texturas no solo servirán para dar color, sino que también se usarán para llevar a cabo efectos o para utilizarlas como transportadores de datos hacia la gráfica. Nos permiten hacer reflejos, distorsiones, efectos de luz, etc.

Las texturas deben ir mapeadas en coordenadas de textura. Estas coordenadas van de 0 a 1; están normalizadas. Y desde el VAO le pasaremos a cada vértice cuáles son las coordenadas de textura que tiene. El *fragment shader* usará estas coordenadas para interpolar el color o la información que debe coger de la textura para aplicarla a lo largo del polígono.

Hemos de tener cuidado con un cosa: en OpenGL, las texturas tienen un punto inicial de 0,0 abajo a la izquierda.

Fijaos en el capítulo «Textures» de este enlace.

3.3.1. *Wrapping, filtering* y *mipmaps*

Haciendo uso del comando glTextParameter, podemos variar las opciones aplicables a texturas. Estas opciones son las siguientes:

- *Wrapping*, modos de repetición.
- *Filtering*: *nearest* y *linear* son los más importantes. Se pueden aplicar cuando maximizamos o minimizamos texturas.
- *Mipmaps*.

3.3.2. Carga de textura a gráfica

A las texturas, como a todos los elementos de OpenGL, se les tiene que asignar un ID, hacer un *bind* y cargar hacia la gráfica.

También tendremos que declarar nuevos *vertex attributes* para pasar las coordenadas de textura hacia el *shader*.

Hay que tener claro el uso de las siguientes funciones OpenGL:

- glTexParameter
- glGenTextures
- glActiveTexture si tenemos más de una: GL_TEXTURE0, GL_TEXTURE1...
- glBindTexture
- glTexImage2D

Dentro del GLSL, hay que estudiar los siguientes tipos de datos y funciones:

- sampler1D,2D,3D, que se usa en el *fragment shader* como *uniform*.
- texture().
- mix(): mezcla de texturas.

Hemos de hablar aquí del concepto de *texture units*: OpenGL tiene diferentes «compartimentos» para trabajar con texturas. Se denominan *texture units*. Y en la implementación de OpenGL, se garantizan un máximo de 16 texturas simultáneas para usar en el pintado de cada vértice.

Hay que decir también que si al *fragment shader* le hacemos una multiplicación de la textura por un color vec4 dado, el resultado será la mezcla del color de la textura con el color que coloquemos.

Contenido complementario

La transparencia es un punto interesante y hay que comprender bien cómo activarlo. Aquí podéis encontrar unas FAQ bastante útiles a manera de resumen de cómo hacerlo.

Reto 6

1. Pintad un rectángulo texturizado con el logotipo de Batman sin ningún color más añadido.

2. Modificad el pintado del logotipo de Batman para que haya cuatro logotipos en el mismo rectángulo.

3. Pintad el logotipo de Batman y la cara del Joker en el mismo rectángulo mezclados, y utilizando las teclas del cursor, cambiad si se ve más el logotipo de Batman o la cara del Joker.

EXTRA

Si queréis usar transparencia en texturas, hay que activar el proceso de mezcla *blending*. Podéis encontrar más información en el capítulo de este enlace.

4. Transformaciones y sistemas de coordenadas

En OpenGL se utilizan los vectores y los cálculos con matrices para realizar los movimientos, rotaciones y traslaciones de los objetos 3D. También se usan para calcular la cámara, la corrección de perspectiva, la luz y el color; es decir, casi para todo. Hemos de entender y tener presente cómo se hacen los cálculos y la base matemática. Aunque una biblioteca hará el trabajo por nosotros, somos nosotros quienes tenemos que colocar correctamente las llamadas a funciones y entender el porqué.

4.1. Vectores y matrices

Antes que nada, hagamos un repaso de la base matemática, que hemos de tener clara: vectores y matrices. Leed el siguiente enlace resumen. De él os tienen que quedar claros los siguientes puntos:

1) Vectores
- **a)** Componentes: dirección y magnitud.
- **b)** Operaciones escalares (un solo dígito en cada componente) + – * /.
- **c)** Operaciones con vectores
  - Suma.
  - Resta: da el vector entre dos puntos.
  - El módulo. Pitágoras.
  - Normalización del vector: dividir por su módulo cada componente. Vector unitario.
  - Multiplicación escalar de símbolo · (el punto).
    - Módulo de V1 * módulo de V2 * coseno del ángulo entre ellos.
    - Si los dos son unitarios, su producto escalar es el coseno del ángulo.
      - 0 si son de 90 grados, 1 si son de 0 grados.
    - Multiplicación vectorial de símbolo ×.
      - Se calcula con los componentes contrarios del que queremos saber.

2) Matrices
- Operaciones escalares.
- Suma, resta y multiplicación escalar con matrices (por cada componente individualmente).
- Multiplicación de dos matrices. Dos normas:

  - Solo se pueden multiplicar si hay N columnas en la primera y N filas en la segunda.
  - No tiene propiedad conmutativa.

- Multiplicación de matriz × vector. Es para hacer las transformaciones. En OpenGL se suele trabajar con matrices de 4 × 4 y vectores de 4 × 4.
- Matriz identidad. Diagonal llena de 1 y el resto 0. No hace cambios en un vector al multiplicarlo.

Cuando trabajamos con 3D, hay tres operaciones básicas que hacemos con los objetos: traslación, rotación y escalado. En OpenGL, estas tres operaciones se hacen utilizando matrices de cuatro componentes. Se hacen con cuatro componentes y no con tres por dos motivos: es más fácil para un procesador hacer cálculos con múltiplos de base 2, y el cuarto componente se utiliza como corrector de perspectiva en ciertas operaciones.

Contenido complementario

Por suerte por los que no dominamos demasiado los cálculos matemáticos, existen librerías que nos hacen el trabajo sucio. En este caso GLM se la librería de definiciones de matrices y vectores que necesitamos en C++.

Las tres operaciones (T-R-S) se hacen sobre una única matriz, que hay que preparar. Una vez preparada, la pasamos al *vertex shader* como *uniform* para multiplicarla por cada posición de cada vértice en el espacio. Para montar esta matriz, es importante el orden en que la montamos, el orden de las funciones para irla modificando, ya que la multiplicación de matrices no tiene la propiedad conmutativa. En el código siempre haremos Traslación × Rotación × Escalado (en este orden), que a efectos reales en el universo 3D, se aplicarán a la inversa. Sobre el origen de coordenadas local del objeto, que al principio coincidirá con el origen del espacio 3D, primero se escalará, después se rotará, y finalmente, se trasladará a su posición.

Reto 7:

Recuperad el pintado de un quad a pantalla y utilizando la librería GLM para tener las definiciones de matrices en C++ aplicad los siguientes movimientos:

1. Trasladarlo a una esquina de la pantalla.

2. Dibujar el mismo quad 5 veces en posiciones aleatorias de la pantalla.

3. Conseguir que los 5 quad roten sobre ellos mismos a diferentes velocidades.

4. Hacer que los 5 quad roten sobre ellos mismos a diferentes velocidades y que tengan diferentes medidas (escalados).

4.2. Los cinco sistemas de coordenadas y las tres matrices

Ya conocemos la importancia de tener las coordenadas de nuestros objetos normalizados con valores enteros 0 y 1. Llegar de estas coordenadas a unas coordenadas de pantalla se hace siguiendo un proceso paso a paso en el que transformamos estas coordenadas a diferentes sistemas de coordenadas hasta las coordenadas de pantalla. Esto hace que el cálculo sea muy sencillo de efectuar.

Nos harán falta un total de cinco sistemas de coordenadas:

- Coordenadas locales.
- Coordenadas del mundo.
- Coordenadas de vista (*view space* o *eye*) [cámara].
- *Clip space* (recorte visible - o*clusion*).
- Coordenadas de pantalla (2D).

Estos cinco sistemas necesitarán tres matrices que tendremos que construir para elaborar todo el proceso:

- Matriz modelo (T - R - S).
- Matriz de vista (cámara).
- Matriz de proyección (perspectiva u ortogonal).

Leed el siguiente tutorial para ver cómo se calculan y cómo se hace uso de la biblioteca GLM para crear cada matriz.

Destacamos que:

- Al hacer la multiplicación en el *shader*, el orden tiene que ser este:

$V\_clip = M\_projeccion \cdot M\_view \cdot M\_modelo \cdot V\_local$

- OpenGL usa un sistema de coordenadas de mano derecha.
- Hay que activar y utilizar el *Z-buffer*: test de profundidad.
  - glEnable(GL_DEPHT_TEST).
  - Se tiene que limpiar en cada *frame* con glClear(GL_DEPTH_BUFFER_BIT).

Reto 8:

Recrear el texto introductorio de Star Wars utilizando un quad que se desplace. Se puede o mover el objeto por matriz de modelo o mover la cámara por la matriz de view .

5. Cámara en un entorno 3D

Definir una cámara en OpenGL es definir un sistema de coordenadas nuevo en el punto en el que tenemos la cámara. Por lo tanto, nos hará falta:

- Un punto en el espacio: la posición.
- Un vector de dirección.
- Un vector hacia la derecha.
- Un vector perpendicular que indique su parte superior.

Veamos cómo extraer cada uno de estos parámetros.

La cuestión es simplemente declarar un vector en el punto donde queremos la cámara. El vector dirección es la resta normalizada del punto de posición al punto hacia donde mira la cámara. Para hacer el vector hacia la derecha, nos podemos inventar un vector perpendicular en Y y hacer con él el producto vectorial contra el vector dirección. Ya solo nos queda calcular el vector perpendicular real. Esto lo hacemos con el producto vectorial de los dos vectores calculados: dirección y derecha. La matriz resultante será la que ocupará la matriz de vista que comentábamos en el apartado anterior.

Contenido complementario

Todas estas operacions con la biblioteca GLM se resumen en una sola instrucción: glm::lookAt. Consultad sus parámetros.

Reto 8

Montad las matrices y las capturas de tecla y ratón necesarias para tener una cámara en primera persona que nos permita movernos en un universo 3D. Usad las típicas teclas WASD o los cursores.

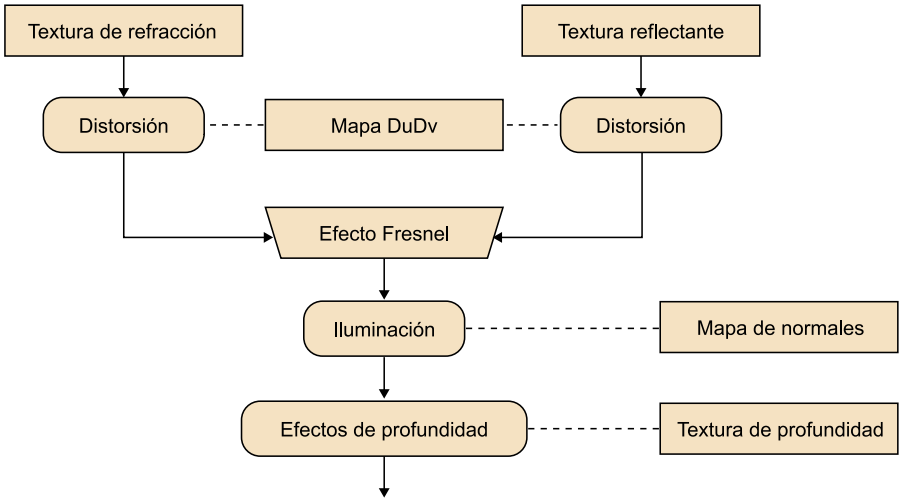
6. El efecto de agua

Contenido complementario

Para realizar los efectos de luz y reflejo sobre el agua, tendremos que preparar unos *shaders* y unos valores de vectores para tener luz en la escena. Echad un vistazo a los seis tutoriales sobre luz de LearnOpenGL.

Vamos a ver cómo crear un efecto de agua muy realista haciendo uso de OpenGL. Crearemos varios efectos sobre una superficie poligonal hasta darle la apariencia de agua; efectos como ondulación de la superficie, superficie reflectante, refracción de la luz en el agua o cómo darle un toque espumoso.
Antes de ponernos a trabajar en este punto, hace falta y es imprescindible que tengamos claros todos los conceptos básicos que hemos visto durante los puntos anteriores del PLAN; conceptos básicos de OpenGL, su máquina de estados y el lenguaje GLSL.
El agua que realizaremos es básicamente un polígono *quad* plano, hecho de dos triángulos, en el que pondremos diferentes técnicas de sombreado con *shaders* para hacerlo parecer un superficie ondulada y con reflejo.

Gráfica 1.



Lo primero que haremos es pintar la escena sobre otras texturas, como hacer una fotografía, a fin de utilizarlas para hacer efectos como la reflexión. Tomaremos dos fotografías de la escena. Tendremos dos cámaras, una de las cuales será para captar todo lo que hay bajo el agua.

Esta nos servirá para hacer la refracción del interior del agua. Y la otra fotografía la tomaremos en una posición opuesta, de forma que capte el efecto de reflejo de los objetos de fuera del agua. Estas dos texturas las usaremos para pintar el *quad* que comentábamos.

No obstante, antes las distorsionaremos. Usaremos mapas DuDv, que explicaremos más adelante. Esta distorsión cambiará constantemente sus parámetros por un factor de tiempo, lo que le acabará de dar el efecto de ondulación. A continuación, mezclaremos las dos texturas distorsionadas para pintarlas usando una técnica llamada efecto Fresnel.

Lo que hace es determinar la cantidad que tenemos que ver de las dos texturas en función del ángulo con el que miramos hacia el agua, es decir, si más transparente o más efecto espejo de los objetos en superficie. Unos cálculos de luces que utilizan mapas de normales sobre el *quad* acabarán de dar el efecto de que el plano no es plano, sino rugoso. Y finalmente, un poco de luz especular le dará un efecto de brillo.
Adicionalmente suavizaremos los bordes, y podríamos hacer que las partes más profundas se vean algo más turbias utilizando textura de profundidad, que podemos recrear a partir de la textura de refracción de objetos interiores.

Reto 9

Dibujad el *quad* plano con un *shader* simple que haga un color azul opaco. Tiene que estar en perspectiva y quedar como plano en el suelo (Y = 0).

6.1. Renderizado a textura - *Frame buffer objects*

Habitualmente, cuando hacemos el pintado de una escena, lo hacemos sobre la pantalla final de resultado. Pero también podemos pintar el resultado sobre una textura interna de la tarjeta gráfica y guardarla allí para usarla como textura normal en otros objetos 3D. Estas texturas pueden ser constantemente actualizadas y cambiadas de contenido en tiempo real. Este «truco» lo usaremos para hacer el reflejo de los objetos sobre el agua. Para esto, hay que colocar correctamente la cámara para que el reflejo sea creíble a la hora de hacer el pintado de la textura.

Estas texturas se guardan en FBO (*frame buffer objects*), que son como superficies de pintado, lienzos. Estos FBO tienen dos *buffers*: el *color buffer*,que son los píxeles en sí, y el *depth buffer*, que guarda la profundidad 3D que tiene el píxel pintado. Cualquiera de los dos *buffers* puede usarse como textura de otros objetos independientemente.

Contenido complementario

Notaremos que nuestra textura sale invertida. ¿Sabéis por qué?

Para llevar a cabo nuestro efecto, necesitaremos dos FBO, los dos con *color buffer* y *depth buffer*. Uno será para la reflexión del agua y el otro para la refracción.

Reto 10

Dibujad en un lateral de la pantalla una copia de la escena en la que tenemos el *quad* azul a menor resolución (una quinta parte) y pintad la escena en un FBO. Después utilizadlo como textura de otro *quad*.

Contenido complementario

Para saber más sobre *frame buffers*, consultad este enlace:
Framebuffers a LearnOpenGL

A continuación, enumeramos las funciones OpenGL importantes para este punto:

- glGenFramebuffer
- glBindFramebuffer
- glDrawBuffer
- glFramebufferTexture
- glGenRenderbuffers
- glBindRenderbuffer
- glRenderbufferStorage
- glFramebufferRenderbuffer

Nota

Cuando hagamos el pintado en nuestro FBO, hemos de utilizar glViewport para ajustar el pintado a la medida de la textura que estamos guardando.

Contenido complementario

Open GL tiene muchas variables predefinidas para usar en sus *shaders*. Podéis consultar la lista entera aquí:

Variables Build-In

6.2. Planos de recorte - *Clipping planes*

Los planos de recorte permiten especificar un plano 3D en nuestra escena que hace que todo lo que queda fuera de él no se pinte. Esta cualidad nos es muy útil para especificar, cuando llevamos a cabo los pintados de las texturas, qué utilizaremos para hacer la refracción y la reflexión del agua. En la primera, nos interesará solo coger todo aquello que hay debajo del plano del agua, y en la segunda, todo aquello que hay en el plano superior. Estos planos también son importantes por un tema de rendimiento. Así evitamos que los *shaders* se entretengan pintando cosas que no queremos.

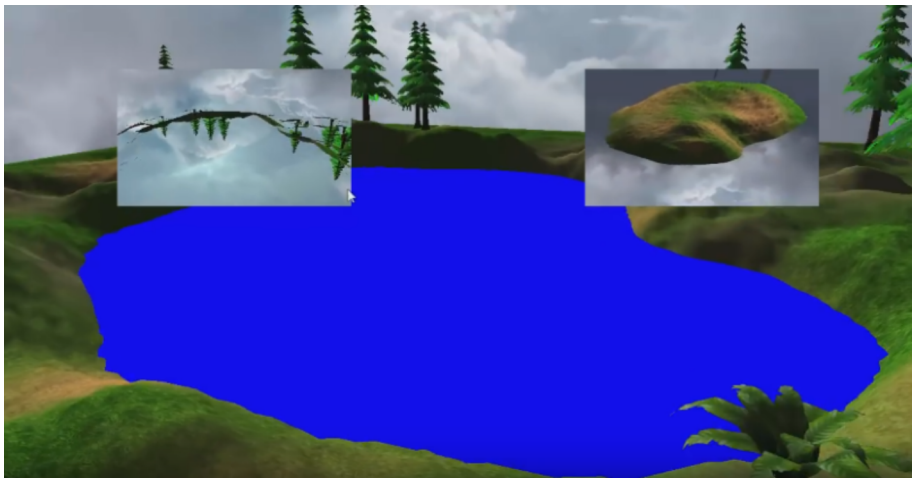
Para utilizar estos planos, usaremos una de las variables Built-in que hay en el *vertex shader*: gl_ClipDistance[]. También hay que hacerle un *enable* de esta característica. Estudiad el tema del signo que toma la distancia de lo que queremos pintar hasta el plano de corte, ya que la distancia que dé negativa respeto al punto estudiado no será pintada. Recordad cómo es la ecuación de un plano y el concepto de normal de un plano: ecuaciones del plano, Wikipedia.

Entramos por Uniform y cogemos en el *vertex shader* el dato que define cómo es el plano de corte que queremos hacer, un vec4. Lo que haremos es calcular el producto escalar entre el plano de corte y el vértice que nos entra con la sentencia de GLSL dot(). La distancia resultante es la que se asignará al gl_ClipDistance[0].

Como nuestro plano está orientado a Y, la normal del plano de corte es sencillamente (0,1,0) en el caso de querer renderizar la parte positiva, y por lo tanto, lo que se verá reflejado en el agua, y (0,–1,0) cogerá todo aquello que hay debajo del agua, y que será sensible a la refracción.

Lo último que tendremos que determinar es la situación de la cámara para hacer los dos *renders*. Para realizar la textura de reflexión, situamos la cámara por debajo de la cámara de visión actual perpendicularmente, en concreto el doble. Y habrá que invertir el sentido del *pitch* de la cámara, puesto que la imagen tiene que salir invertida.

Imagen 1.



Ejemplo de dos viewports y pintado haciendo dos planos de corte. Derecha: Plano superior - Reflejo. Izquierda: Plano inferior - Refracción.

Reto 11

Definid y renderizad en dos *viewports* las texturas que serán utilizadas para hacer la refracción y la reflexión del agua haciendo uso de los planos de corte.

6.3. Mapeo de textura proyectiva

Vamos a colocar nuestras nuevas texturas sobre el plano del agua. Lo que haremos será añadir el código necesario al *fragment shader* para hacer este pintado y el posproceso necesario.

Necesitaremos dos texturas, que, como siempre, entrarán por Uniform, y serán del tipo sampler2D. En el cálculo, relacionamos cada textura con las coordenadas de texturas que nos entran y preparamos 2 vec4 con los colores procesados. Utilizando mix(), hacemos la mezcla de los dos colores al 50%.

Llegados a este punto, hay que tener claro cómo activar múltiples texturas a la hora de hacer el *render* y el uso de las *texture units* de OpenGL. Repasad el capítulo 3 de este PLAN o consultad el siguiente enlace: Textures OpenGL en LearnOpenGL.

Haciéndolo de este modo, lo que veremos no es lo que deseamos, puesto que las texturas se pondrán tal como han sido realizadas. Hemos de utilizar la técnica de la texturización proyectiva. Lo que queremos es solapar los dos renderizados desde un punto de vista de la pantalla. Nos hace falta un reajuste de coordenadas U,V de la proyección de la textura en función de su posición respeto a la pantalla. Por lo tanto, tendremos que hacer un cambio de espacio, de coordenadas.

No usaremos coordenadas de textura que nos puedan venir dadas por nuestro VAO del plano del agua. En vez de esto, tomamos las coordenadas que resultan del cálculo de las transformaciones de los sistemas de coordenadas del *vertex shader*. Así tenemos la proyección sobre la pantalla y estas serán para nosotros nuestras coordenadas de texturas U,V para las texturas de reflejo y refracción. Las pasamos al *fragment shader*. Una vez en el *fragment shader*, normalizaremos las coordenadas X e Y y haremos una división con la corrección de perspectiva W de las coordenadas que vienen del *vertex shader*. Después hay que hacer un ajuste de las coordenadas, puesto que recordemos que las texturas empiezan en cero en la esquina inferior derecha y la pantalla en el centro: dividir entre 2 y multiplicar por 0,5. De estas coordenadas invertiremos la Y para la textura de reflejo, obviamente.

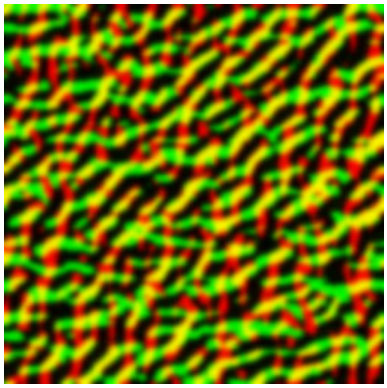
Reto 12

Aplicad correctamente las coordenadas de textura sobre el plano para visualizar el reflejo y el fondo del agua.

6.4. Mapas DuDv

Ya tenemos nuestras texturas colocadas sobre el plano. Ahora tenemos que distorsionarlas. El agua no es un plano quieto en la realidad. Simularemos que tiene cierto movimiento suave, como un lago. Para conseguir este efecto, lo que haremos será variar en el tiempo las coordenadas de textura con las que aplicamos las texturas. Y también tienen que tener valores diferentes en cada punto de la superficie del plano.

Imagen 2.



Pie de foto: Textura DuDv: representa valores de vectores U-V.

El cambio de valores nos lo proporcionará una textura DuDv que podemos construir nosotros. Se trata de una representación visual de valores de los componentes de un vector de dos dimensiones: rojo por U y verde por V. Por eso tiene este tono. Sería ideal que la textura que usamos tuviese un buen efecto de mosaico. Estos valores llegarán normalizados al *fragment shader*. Nos interesa que tenga también valores negativos, por lo que haremos la transformación de multiplicar por 2 y restar 1. Por lo tanto, queremos esta textura también en el *fragment shader* para Uniform y ocupando una de las *texture units*.

Contenido complementario

Para saber más sobre texturas DuDv, consultad el siguiente enlace:

Uso de texturas DuDv

Hay otros tipos de texturas auxiliares para hacer efectos visuales y tener datos en la gráfica, como por ejemplo los *normal maps*, que se usan para dar volumen falso a una superficie que es plana en origen.

Para esta textura sí que usaremos las coordenadas que teníamos calculadas para nuestro plano del agua, puesto que esta no va proyectada contra la pantalla, sino que la distorsión seguirá la proyección del plano del agua en sí. Haremos que estas coordenadas tengan mosaico multiplicándolas por un factor grande 4 o 6 para tener la textura como miniaturizada sobre el agua. Las coordenadas resultantes en el *fragment shader* solo las tendremos que sumar a las calculadas por proyección anteriormente.

Si obtenemos demasiada distorsión, la cuestión no queda real. Hay que cambiar la fuerza de la distorsión como si fuera una onda. Será un factor multiplicativo de la coordenada de distorsión antes de sumarla a las coordenadas de reflexión de las texturas. Este valor tiene que ser bajo, del orden de 0,01 o 0,02. Esta distorsión nos provocará un *glitch* en los bordes de la pantalla. Esto es debido a que OpenGL hace el *loop* de textura y nos quedan unos colores no válidos. Para evitarlo, lo que haremos es un recorte de valor a la hora de hacer el cálculo de las coordenadas una vez sumadas a la distorsión. No dejaremos que estos valores finales lleguen a 0 ni a 1. Por lo tanto, un recorte de 0,001 a 0,9999 sería correcto. Hay que tener cuidado con la Y, que tiene que ser invertida. Con esto ya tendríamos la distorsión hecha, pero el agua está quieta como una balsa de aceite.

Imagen 3.

Reflejo y distorsión de los árboles sobre el agua.

Para hacer el movimiento del agua, lo que haremos es mover la textura DuDv con un desplazamiento controlado por el tiempo mediante un *uniform* en el *fragment shader*. Si vemos que no nos acaba de gustar, podemos calcular más variantes de distorsión: aplicar a X a Y y juntarlo todo en un solo valor.

Reto 13:

Modificad los *shaders* de forma que podamos utilizar una textura DuDv y realizad las distorsiones de coordenadas necesarias para hacer que parezca agua.

6.5. Efecto Fresnel

Cuando miramos una superficie de agua, cuanto más perpendiculares al plano estamos, más transparente es el agua, y cuanto más paralelos a la superficie, un mayor reflejo se produce. Esto es un efecto óptico del efecto Fresnel.

Si recordamos nuestro *fragment shader*, haciendo uso de mix(), hacemos la mezcla de las dos texturas con un valor fijo de 0,5. Lo que haremos será variar este valor en función de la posición de la cámara. El producto escalar de la normal del plano con la dirección hacia donde mira la cámara nos dará este factor de mezcla. La transparencia se efectúa fragmento a fragmento.

Recordad que...

... el vector que buscamos de la cámara es la resta entre la posición de los vértices y la posición de la cámara. El resultado habrá que normalizarlo y después efectuar el producto escalar.
Si encontráramos que es demasiado transparente, podemos hacer un factor exponencial antes de aplicarlo a la mezcla. Quedaría más parecido a un espejo.

6.6. Mapas de normales

Nuestro plano de agua es completamente plano. Le falta cierta rugosidad para parecer más un agua real. Podríamos añadir más polígonos distorsionados y movidos, una malla de polígonos para representar el agua. Pero tendríamos mucho más cálculo. Vamos a calcular cierta iluminación para darle la rugosidad.

La luz en pintado 3D se hace a partir de las normales de los polígonos, pero como nuestro plano es únicamente un plano, solo tenemos una normal. Utilizaremos un mapa de normales (tutorial de mapa de normales) para adquirir más normales sobre la superficie. En estos mapas cada píxel RGB representa un vector normal en el punto XYZ respectivamente. La manera de hacerlo será similar al mapa que hacíamos de DuDv. Como los datos vendrán en expresión de 0 a 1 y queremos que el reflejo sea de –1 a 1, haremos la ya típica conversión de multiplicar por 2 y restar 1. Las coordenadas que hay que utilizar para hacer el mapeo serán las mismas que las del mapa DuDv.

Contenido complementario

En esta parte quizás quedéis un poco confundidos. Aquí podéis encontrar un pequeño tutorial de cómo entender mejor la luz especular, la que se refleja del ambiente.

Echadle un vistazo para entender mejor la matemática que hay detrás: Tutorial Specular Lighting.

Necesitaremos pasar hacia los *shaders* la posición (hacia el *vertex shader*) y el color de la luz (hacia el *fragment shader*) que tenemos. Lo haremos con *uniforms*. La posición –vector ha de llegar al *fragment* del mismo modo que hemos calculado para hacer el efecto Fresnel.

Reto 15

Modificad los *shaders*, añadid el mapa de normales y realizad los cálculos necesarios.

6.7. Suavización de bordes

Seguramente nuestro efecto tiene una serie de errores, *glitches*, por los bordes del agua. Miraremos de solucionarlos. Para hacerlo, usaremos la información del *buffer* de profundidad del FBO, donde hemos renderizado la textura para hacer la refracción. Aquí, aparte del *buffer* de color, está el *depth buffer*, que nos informa de la profundidad a la que se encuentra cada píxel, de manera que el blanco es el más lejano y el negro el más cercano. Esta información nos da la distancia a la que está el fondo del agua respecto a la cámara.

Inspeccionaremos y utilizaremos la variable Build In gl_FragCoord, concretamente el componente *z*, que es el que contiene esta información de profundidad. Haremos la resta con la distancia que tenemos hasta el plano de superficie del agua. Así obtendremos la distancia de profundidad que hay del agua, que marca la cantidad de agua que hay en el punto. Esto nos servirá para hacer efecto de profundidad en el agua.

Para hacerlo, antes que nada declararemos al *fragment shader* un nuevo *uniform* sampler2D para captar el *depth buffer* de la textura de refracción. Esto hará que tengamos ya cinco texturas declaradas para hacer el agua: las dos de refracción y reflexión, el mapa DuDv, el mapa de normales y este nuevo mapa de profundidad de refracción. Cuando hagamos el cálculo de textura en el *fragment shader*, solo habrá que usar el componente R, puesto que es a base de grises y todos los valores son iguales. Tenemos que saber que los valores 0-1 del *depth buffer*

no son lineales, sino exponenciales. Por tanto, nos toca hacer un recálculo. Fijaos en este enlace para más información sobre este cálculo: Conversión del valor *z* del *depth buffer*; o también en este otro: Real depth in OpenGL / GLSL.

Obtenido este cálculo, pasamos a hacer la resta del vector para obtener la cantidad de agua que hay bajo el plano. Si lo hemos hecho bien, podemos ver, si pintamos el color obtenido, cómo la parte más profunda queda de tono blanco (1,0) y los bordes del agua quedan negros (0,0). Con esto podemos hacer que los bordes del agua sean más transparentes que las partes más profundas.

Contenido complementario

Para entender completamente la parte de mezcla final de OpenGL, el *blending*, miraos el tutorial de LearnOpenGL dedicado al tema.

Activaremos la mezcla por alfa, el Alphablending de OpenGL, y utilizaremos el resultado para el alfa del color de salida. Quizás lo tendremos que ajustar numéricamente para obtener un mejor resultado. Este número obtenido también lo podemos aplicar a la distorsión que hemos calculado para dar un toque suave y no tener algunos pequeños defectos en los bordes dados por la distorsión. Y lo mismo para la iluminación especular.

Reto 16

Modificad los *shaders* y el código base para dar una apariencia más suave a los límites del agua.

7. El fuego

Reto final evaluativo: realizar un efecto de fuego

Utilizad todas las técnicas aprendidas y obtenidas en el plan para realizar un efecto relacionado con el fuego.

- Llamas.

- Distorsión de la visión por el efecto del calor.

- Iluminación.

- Deformaciones.

- Reflejos.

Resumen

Tal como hemos visto, el pintado en OpenGL requiere conocimientos matemáticos relativamente sencillos sobre matrices y vectores. No es objetivo de este PLAN entrar en los puntos importantes de las matemáticas que se aplican en él, sino elaborar con imaginación efectos interesantes y espectaculares en gráficos 3D, ya sean reales o de pura fantasía.

Si bien es cierto que todo está inventado y que la mejora de la visualización realista 3D está casi imitada perfectamente, es importante que entendáis cómo se hace y cómo se aplica esto, ya que este conocimiento nos puede ayudar a hacer otros efectos más espectaculares, irreales quizás, pero bonitos o necesarios a la hora de crear nuestro videojuego.

Comprender el pintado 3D desde un punto tan abierto y de tan bajo nivel de implementación os da la libertad de hacer todo lo que queráis. La ventaja es absoluta respecto al uso de motores 3D prefabricados. Os alentamos a seguir investigando, a seguir aprendiendo muchos de los algoritmos que ya hay hechos e investigados para el pintado de efectos 3D. Su comprensión os llevará a crear cosas espectaculares. Además, el lenguaje C++ es un lenguaje de programación de tipo universal que perdurará siempre. Su capacidad para acercarse a cómo son y cómo se mueven los datos en las CPU lo mantendrán siempre a la vanguardia y a tener el control total de la máquina que se invente ahora y en un futuro.

Bibliografía

Sellers, Graham; Wright Jr., Richard S.; Haemel, Nicholas (2015). *OpenGL SuperBible7*.ª edición. Boston: Addison-Wesley Professional.

Shreiner, Dave; Sellers, Graham; Kessenich, John; Licea-Kane, Bill (2013). *OpenGL Programming Guide. The Official Guide to Learning OpenGL*. Boston: Addison-Wesley Professional.

Rost, Randi J. (2006). *OpenGLShading Language*2.ª edición. Boston: Addison-Wesley Professional.

Madsen, Robert; Madsen, Stephen (2016). *OpenGL Game Development by Example*. Birmingham: Packt Publishing.