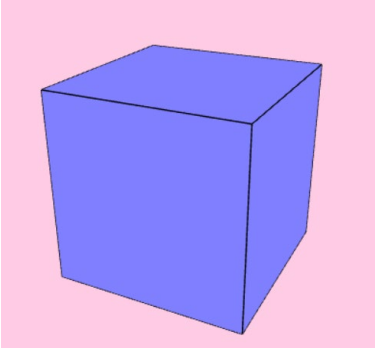
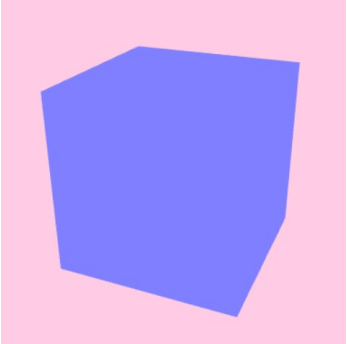
CLASE 5 IG: Iluminación

Práctica 5

Partimos de una versión anterior en Glitch, con un solo cubo en perspectiva flotante, y posibilidad de rotación y zoom con el ratón.

El código tal cual pintaba las caras y las aristas también. Pero en la vida real, las aristas no se pintan. Si comentamos el código que dibuja las aristas en la función glRenderCubeIBO quedaría un pegote azul.



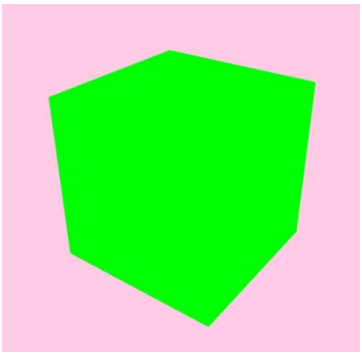


Iluminación difusa rápida usando la coordenada 'y'

Pintamos color uniforme, que se lo pasamos desde el vertex shader

El color es algo que vamos a calcular en el vertex shader, en función de cada vértice, y se lo pasaremos al fragment shader. Así que lo primero es declarar una nueva variable en el vertex shader, donde vamos a calcular el color, y ponerla como varying (con la palabra reservada 'out') para que le llegue al fragment shader:

```
out vec4 vVertexColor;
…
vVertexColor = vec4(0,1,0,1);
```



Y en el fragment shader quitamos el uniform que estábamos usando para leer el color, y lo recogemos en una variable que nos llega desde el vertex shader:

```
in vec4 vVertexColor;
…
fragColor = vVertexColor;
```

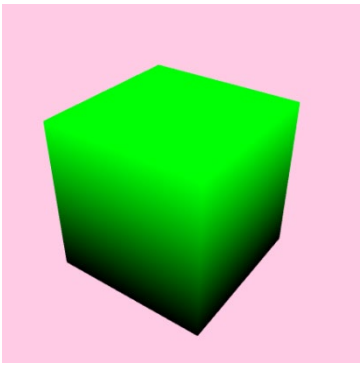
El resultado es un pegote igual que antes pero en verde. No notaremos la 3D. Así que necesitamos calcular el color en función del vector normal

Elegimos color en función de la coordenada y

Vamos a hacer MAGIA ahora. Pongamos el color por ejemplo en función de la coordenada 'y' original de cada vértice. Recordemos que las coordenadas originales, que nos llegan en el array_buffer, antes de que sean multiplicadas por la modelMatrix, estaban entre 0 y 1.

Entonces simplemente ponemos en el vertex shader un nuevo valor de color, donde el tono de verde sea el valor de la coordenada ‘y’ original:

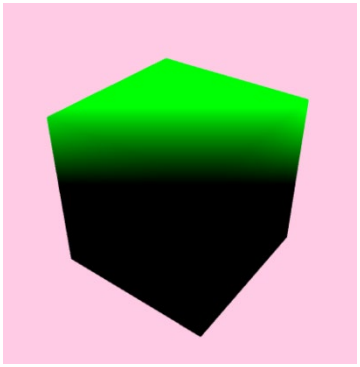
```
vVertexColor = vec4(0,aCoordinates.y,0,1);
```



Y luego el fragment shader hace la magia, porque interpola el valor de cada pixel en el interior de cada triángulo. Al ponerlo en función de la coordenada ‘y’ es como si hubiera una bombilla en el polo norte.

Posición de luz fija o se mueve con el ratón?

Si nos fijamos al rotar la esfera, la parte brillante de la esfera se mueve con ella. Es decir, la bombilla del polo norte está anclada a la esfera. Pero si quisiéramos que la bombilla estuviera quieta en el techo, y que sólo rotáramos la esfera?



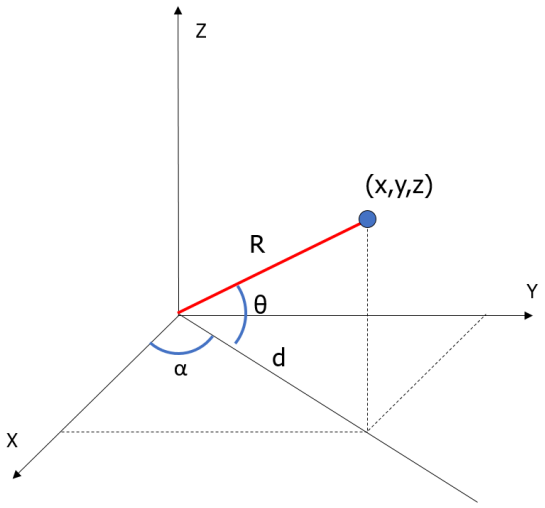
Es muy fácil. En lugar de calcular el verde en función de la ‘y’ de los vértices originales, lo calculamos a partir de la ‘y’ ya transformada con la modelMatrix y la viewMatrix:

```
vVertexColor = vec4(0, gl_Position.y,0,1);
```

Cambiando cubo por esfera

Para notar mejor los efectos de la iluminación, pasemos a una esfera en lugar de un cubo. En el código ejemplo ya está incluido el código para pintar una esfera por una malla de triángulos, usando la primitiva de dibujo triangle_strip, y utilizando coordenadas esféricas.

$z = R \sin \theta$
$d = R \cos \theta$
$x = d \cos \alpha = R \cos \theta \cos \alpha$
$y = d \sin \alpha = R \cos \theta \sin \alpha$



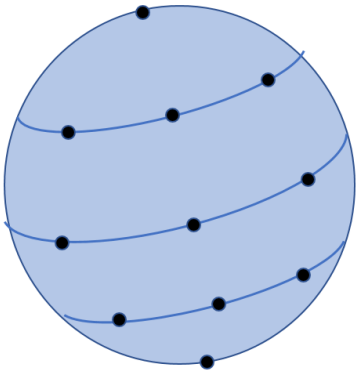
N indica el número de strips

Por ejemplo n=4 sería 4 bandas como en la figura

Dicho de otro modo, un paralelo cada pi/n radianes

A lo largo de cada paralelo irían 2*n puntos

Todas las bandas se pintan con triangle_strip, excepto las que tocan los polos, que serían triangle_fan



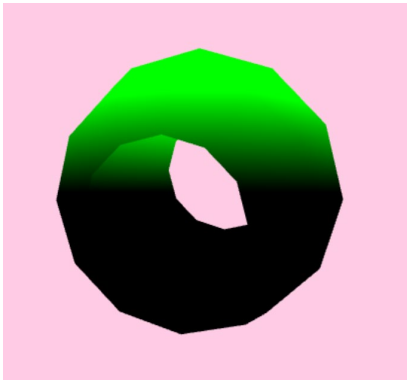
A continuación el código de la función que dibuja la esfera:

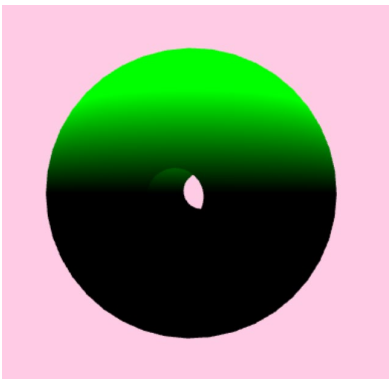
```
function glRenderSphereIBO(n) {

   glPushMatrix();
        mat4.scale(modelMatrix, modelMatrix, [0.7,0.7,0.7]);
        gl.uniformMatrix4fv(modelMatrixLoc, false, modelMatrix);

        // compute vertices
        coords = new Float32Array(6 * n * n);
        step = Math.PI / n;
        R = 1;
        k = 0;
        for (i = 1; i < n; i++) {
          tita = -Math.PI / 2 + i * step;
          for (j = 0; j < 2 * n; j++) {
            alpha = j * step;
            coords[k++] = R * Math.cos(tita) * Math.cos(alpha);
            coords[k++] = R * Math.cos(tita) * Math.sin(alpha);
            coords[k++] = R * Math.sin(tita);
          }
        }

        // compute faces
        arrayIFaces = new Uint16Array((4 * n + 2) * n);
        k = 0;
        for (i = 0; i < n - 2; i++) {
          for (j = 0; j < 2 * n; j++) {
            arrayIFaces[k++] = 2 * n * (i + 1) + j;
            arrayIFaces[k++] = 2 * n * i + j;
          }
          arrayIFaces[k++] = 2 * n * (i + 1);
          arrayIFaces[k++] = 2 * n * i;
        }

        // pass data to GPU
        gl.bindBuffer(gl.ARRAY_BUFFER, vertex_buffer);
        gl.bufferData(gl.ARRAY_BUFFER, coords, gl.STATIC_DRAW);
```

```
        gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, arrayIFaces,
            gl.STATIC_DRAW);

        // draw meshes
        for (i=0; i<n-2; i++)
           gl.drawElements(gl.TRIANGLE_STRIP, 4*n+2,
                  gl.UNSIGNED_SHORT, 2*i*(4*n+2));

    glPopMatrix();
    delete coords;
    delete arrayIFaces;
}
```

Y ahora sustitutimos la llamada a glRenderCubeIBO por glRenderSphereIBO(n). En el código actual la esfera tiene dos agujeros en los polos, para hacer el código más sencillo, ya que esa parte hay que programarla un poco diferente. Esto además tiene 1 ventaja en esta práctica: los agujeros de la esfera me permiten verla como una aceituna deshuesada mientras roto con el ratón (si fuera una esfera completa sería totamente simétrica y no se vería rotar).

Si usamos n=6 se vería bastante pixelado el contorno, y el agujero enorme. Mientras que si usamos n=20, el resultado es más redondito (a costa de pintar muchos más triángulos).





Como trabajo opcional podría ser terminar de cerrar la esfera.

Iluminación difusa usando los vectores normales

Viendo mensajes de error en los shaders

A medida que empezamos a crecer el código de los shaders, es bueno preguntar por el error al compilar los shaders. Esto puede hacerse de forma rápida añadiendo esta código justo después de compilar cada shader:

```
if (!gl.getShaderParameter(vertShader, gl.COMPILE_STATUS)) {
      console.error("vertShader: " + gl.getShaderInfoLog(vertShader));
      return null;
}
```

De esta forma, pulsando F12 en el Chrome podremos ver la línea y el mensaje de error.

Usando el vector normal

Ahora mismo solo estamos iluminando por la coordenada 'y'. Pero esto no es lo correcto. Lo ideal es calcular el color en función del ángulo entre el vector normal y el vector que viene de la luz.

En primer lugar vamos a pasarle al vertex shader el vector normal a cada vértice para que el shader haga el cálculo del ángulo.

```
in vec3 aVertexNormals;
```

Lo siguiente es crear una variable global normalsLoc, donde meter la ubicación del vector de normales en el vertex shader:

```
var normalsLoc;
…
normalsLoc = gl.getAttribLocation(shaderProgram,
       "aVertexNormals");
```

A continuación debemos crear un buffer donde almacenar las normales:

```
var normal_buffer;
…
normal_buffer = gl.createBuffer();
```

Y al final del método init apuntamos los atributos al buffer:

```
gl.bindBuffer(gl.ARRAY_BUFFER, normal_buffer);
gl.vertexAttribPointer(normalsLoc, 3, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(normalsLoc);
```

Lo siguiente es crear el array de normales dentro del método glRenderSphere. La buena noticia es que, para una esfera centrada en el origen, **el vector normal en cada punto coincide con las coordenadas de dicho punto!** Así que vamos a utilizar el mismo array.

```
normals = coords;
```

OJO: Nótese que dentro del código de glRenderSphereIBO(n) tenemos que hacer primero glbind del buffer de vértices para meter los vértices, y luego glbind del buffer de las normales para meter los vectores normales.

El código completo para pasar los datos a la GPU dentro del método glRenderSphereIBO() quedaría así:

```
// pass data to GPU
gl.bindBuffer(gl.ARRAY_BUFFER, vertex_buffer);
gl.bufferData(gl.ARRAY_BUFFER, coords, gl.STATIC_DRAW);
gl.bindBuffer(gl.ARRAY_BUFFER, normal_buffer);
gl.bufferData(gl.ARRAY_BUFFER, normals, gl.STATIC_DRAW);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, arrayIFaces, gl.STATIC_DRAW);
```

Y por último, hay que cambiar el código del vertex shader, en donde nos va a llegar la variable aVertexNormals para cada vértice por separado. Así que habrá que calcular el coseno de ángulo entre la dirección a la luz y el vector normal (nos la da el producto escalar), y esto

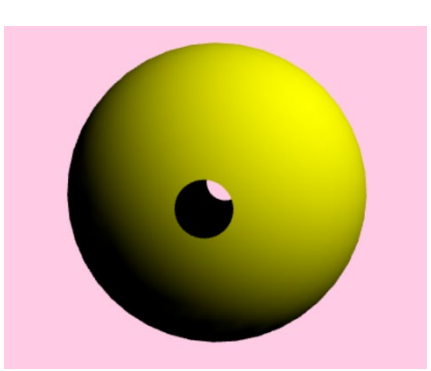
representará un valor entre 1 (de frente a la luz → más iluminación) y 0 (perpendicular → no le llega luz).

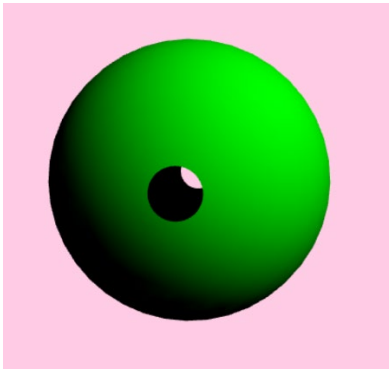
Realmente el código del vertex shader lo vamos a refactorizar un poco. Quedaría así:

```
void main(void) {
  // saving vertex after transformations BEFORE PERSPECTIVE
  vec4 vertex = uModelMatrix * vec4(aCoordinates, 1.0);
  gl_Position = uViewMatrix * vertex;

  // computing vectors
  vec4 light = vec4(10,10,10,1);
  vec3 L = normalize(light.xyz - vertex.xyz);
  vec3 N = normalize(vec3(uModelMatrix * vec4(aVertexNormals, 0.0)));

  // computing diffuse component
  vec3 diffuseMaterial = vec3(0,1,0);
  float diffuse = max(dot(N, L), 0.0);
  vec4 Idif = vec4(diffuse*diffuseMaterial,1);

  // computing ambient component
  vec4 Iamb = vec4(0,0,0.4,1);

  // compute final vertex color
  vVertexColor = Iamb + Idif;
}
```

Como hemos puesto la posición de la luz en (10,10,10), eso quiere decir que estará por delante, arriba a la derecha de la bola.

También estamos usando un RGB a fuego para el material difuso (ahora mismo es verde), de forma que el producto N*L afecta a todo el color (prueba a cambiarlo en el código para ver cómo cambia)





Fíjense que la luz está una posición fija de la escena, y cuando rotamos la esfera no le afecta. Pero si quisiéramos que la luz estuviera anclada a la esfera, entonces simplemente deberíamos multiplicar la misma uModelMatrix que multiplica a los vértices, también a la posición de la luz. Es decir,

```
vec4 light = uModelMatrix * vec4(10,10,10,1);
```

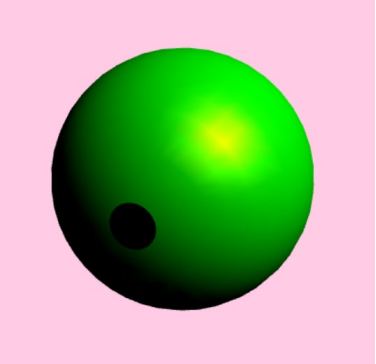
Añadiendo luz especular

Añadiendo luz especular

Para añadir luz especular necesitamos otro material, y añadir el cálculo en el vertex shader, según las matemáticas de las diapos:

```
// compute specular component
vec4 Ispec = vec4(0,0,0,1);
float NL = dot(N,L);
if (NL>0.0) {
    vec3 R = 2.0*N*NL-L;
     vec3 V = normalize(-vertex.xyz);
    float shininess = 10.0;
    vec3 specularMaterial = vec3(1,0,0);
    float specular = pow(max(dot(R, V), 0.0), shininess);
    Ispec = vec4(specular * specularMaterial, 1);
}

// compute final vertex color
vVertexColor = Iamb + Idif + Ispec;
vVertexColor = min(vVertexColor, vec4(1,1,1,1));
```

Ahora ya podemos ver el reflejo especular, y controlarlo también con el color del material y con el valor del shininess. De hecho, una posible ampliación sería añadir controles con gui.dat para varias estos parámetros interactivamente y ver el resultado en tiempo real.



Sombreado de Phong en el fragment shader

Pasando el cálculo al fragment shader

Para pasar a Phong, el color final lo va a calcular el fragment shader. Y en el vertex shader sólo vamos a calcular 3 cosas (que son particulares a cada vértices):

- El vector normal
- El vector que apunta al ojo
- El vector que apunta a la luz

Una vez calculados, estos tres vectores los pasamos como varying al fragment shader, y luego en el shader calculamos el resto. Para empezar quitaríamos el out vec3 vVertexcolor, y añadiríamos los siguientes tres varyings:

```
out vec3 vNormal;
```

```
out vec3 vEyeVector;
out vec3 vLightDirection;
```

Y luego en el main, después de calcular la gl_Position, calcularíamos estos tres vectores de la siguiente manera:

```
// compute normal vector
vNormal = vec3(uModelMatrix * vec4(aVertexNormals, 0.0));

// compute vector from vertex to viewer
vEyeVector = -vertex.xyz;

// compute vector from vertex to light
vec4 light = vec4(10,10,10,1);
// vec4 light = uModelMatrix * vec4(10,10,10,1);
vLightDirection = light.xyz - vertex.xyz;
```

La línea comentada en el cálculo del vector light es para si queremos que la luz esté fija en la escena o anclada a la esfera, como antes. Si la descomentamos, la luz quedara anclada.

Fíjemonos que los tres vectores varyings los hemos definido como vec3 y no vec4, porque aún hay que operar con ellos. Y que no los normalizamos, ya que el fragment shader lo que va a hacer es interpolar un vector para cada pixel, y puede que no estén normalizados.

En el fragment shader, eliminaríamos el in vec4 vVertexColor que nos venía desde el vertex shader, y recogemos los tres nuevos vectores de entrada:

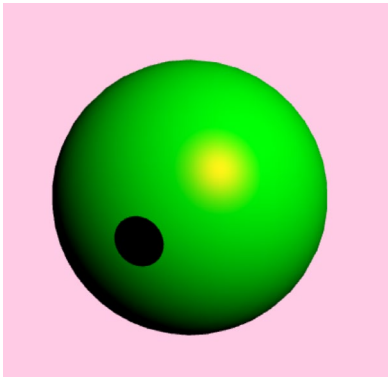
```
in vec3 vNormal;
in vec3 vEyeVector;
in vec3 vLightDirection;
```

Y luego hacer el cálculo de la ecuación de iluminación, con los tres vectores interpolados que nos llegarán para cada pixel:

```
// computing diffuse component
vec3 N = normalize(vNormal);
vec3 L = normalize(vLightDirection);
vec3 diffuseMaterial = vec3(0,1,0);
float diffuse = max(dot(N, L), 0.0);
vec4 Idif = vec4(diffuse*diffuseMaterial,1);

// compute specular component
float NL = dot(N,L);
vec4 Ispec = vec4(0,0,0,1);
if (NL>0.0) {
     vec3 R = 2.0*N*NL-L;
     float shininess = 10.0;
     vec3 specularMaterial = vec3(1,0,0);
     vec3 V = normalize(vEyeVector);
     float specular = pow(max(dot(R, V), 0.0), shininess);
     Ispec = vec4(specular * specularMaterial, 1);
```

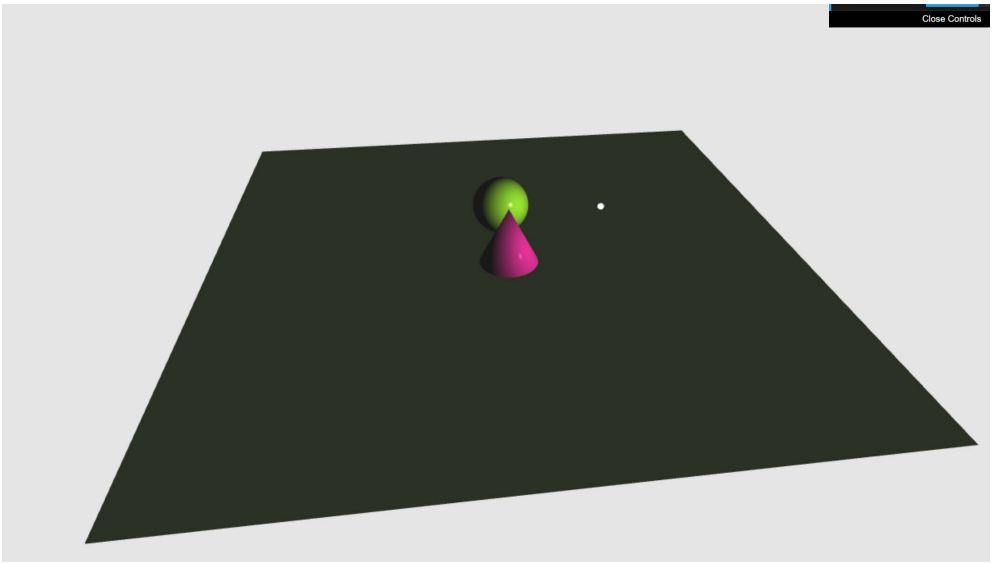
```
}

// computing ambient component
vec4 Iamb = vec4(0,0,0.4,1);

// calculamos color final
fragColor = Iamb + Idif + Ispec;
fragColor = min(fragColor, vec4(1,1,1,1));
```

Ahora se ve mucho mejor que con Gouraud



Posibles ampliaciones

- Añadir controles de gui.dat para modificar los parámetros (posición luz, luz anclada a objeto o fija, color luz, color material, etc)
- Meter iluminación en la escena de la práctica anterior (recordar añadir normales)
- Añadir más luces
- Animar las luces (si queremos mostrar la posición de las luces de alguna manera en la escena, tengan en cuenta que necesitamos pintar algo de geometría para que se vean. Lo más fácil es dibujar una esfera muy pequeñita, pero pasarle material ambiente blanco o amarillo, y material difuso y especular negro. Así siempre se verá amarillo o blanco brillante en cualquier posición)



Añadiendo un color dialog a la gui

Para añadir a la interfaz el poder elegir el color, se añade un parámetro a la variable controls de tipo hexadecimal:

```
this.difColor = "#00FF00";
```

Y luego lo añadimos a la gui haciendo:

```
gui.addColor(controls, 'difColor');
```

Luego hacemos lo de siempre:

- crear una variable difColorLoc,
- llamar a gl.getUniformLocation en el init,
- y luego antes de pintar llamar a gl.uniform pasándole el valor actual

```
var difColorLoc;
…
difColorLoc = gl.getUniformLocation(shaderProgram, "uDifColor");
…
gl.uniform3fv(difColorLoc, hexToRgb(controls.difColor));
```

NOTA: la función hexToRgb es necesaria para convertir la string hexadecimal con los tres valores RGB a un vector float de tres valores entre 0 y 1

```
function hexToRgb(hex) {
      // Elimina el signo "#" si está presente
       hex = hex.replace(/^#/, '');

      // Divide el valor hexadecimal en sus componentes RGB
      const r = parseInt(hex.substring(0, 2), 16) / 255;
      const g = parseInt(hex.substring(2, 4), 16) / 255;
      const b = parseInt(hex.substring(4, 6), 16) / 255;

       // Devuelve un objeto con los valores RGB normalizados
      return [r, g, b];
}
```