CLASE 3 IG: Transformaciones

Práctica 3

En teoría ya habremos dado todo el tema de las matrices de transformación homogéneas.

Partimos de esta plantilla básica en Glitch, con algunas de las cosas que ya hicimos las anteriores semanas.

Nueva función para dibujar cuadrado

Olvidemos la función de drawRectangle que usamos para el Pong, y creemos una llamada drawSquare, que solo pinta un cuadrado centrado en el origen de tamaño 1:

```
function drawSquare() {
    v = new Float32Array([-0.5,0.5,0.5,0.5,-0.5,-0.5,
                          -0.5,-0.5,0.5,0.5,0.5,-0.5]);
    // Pass the vertex data to the buffer
    gl.bufferData(gl.ARRAY_BUFFER, v,
                  gl.STATIC_DRAW);
    gl.drawArrays(gl.TRIANGLES, 0, 6);
  }
```

Pintamos un cuadrado tamaño unidad, simplemente llamando a esta función

```
drawSquare();
```

Añadir librería dat.gui para diseñar la GUI, y crear dos sliders para trasladar

Importamos la librería

Vamos a usar una librería GUI que es la más habitual, tanto en ejemplos de webgl como de three.js, llamada dat.gui.min.js. Su web es https://github.com/dataarts/dat.gui. Normalmente el fichero se encuentra en la carpeta “build”.

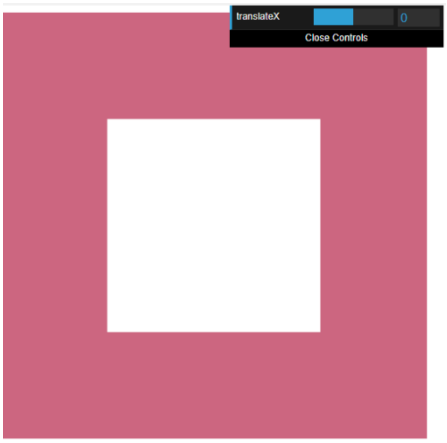
En la parte de head del html hay que importar el script. Si estamos usando web server local, lincamos con la copia del fichero que tengo en serdis. Y si estamos en glitch, tenemos que subir el fichero dat.gui.min.js al proyecto

```
<!-- librería GUI -->
<script type="text/javascript" src="dat.gui.min.js"></script>
```

Añadimos un slider

Al comienzo de nuestro script principal, fuera de las funciones, primero creamos un objeto controls que guarde las variables que queremos interactuar

```
  var settings = {
      translateX: 0.0
   }
```



Y luego creamos el slider

```
 var gui = new dat.GUI();
 gui.add(settings, 'translateX', -1.0,
1.0, 0.01);
```

Trasladar en el shader

Para mover el cuadrado, debemos leer el valor del slider, y pasárselo al vertex shader para que le sume esa cantidad. Para ello añadimos un uniform al vertex shader, y se lo sumamos a gl_position.

```
uniform vec2 uTranslation;
void main(void) {
      gl_Position = vec4(aCoordinates+uTranslation, 0.0, 1.0);
```

Luego declaramos una variable global translation:

```
var translationLoc;
```

En el init asociamos esta variable al uniform, justo donde ya hacíamos lo mismo con otros uniforms previos:

```
translationLoc = gl.getUniformLocation(shaderProgram,
"uTranslation");
```

Finalmente en cada render metemos el valor de la GUI en la variable (por ahora solo metemos la traslación en X), antes de la llamada a dibujar el cuadrado:

```
// translate from GUI
gl.uniform2f(translationLoc, settings.translateX, 0);
```

NOTA: este cambio de trasladar en el shader es importante. Anteriormente en las prácticas anteriores, la suma de la cantidad a trasladar se hacía en GPU, y si tuviera una escena de muchos vértices, sería lento. Ahora la suma se hace en el vertex, con lo cual es rapídisimo porque la GPU lo ejecuta en paralelo.

Añadimos un segundo slider

Podemos hacer lo mismo pero con la y:

- Le damos un valor por defecto en la función controls;
- Creamos un nuevo slider
- Le pasamos el vector con ambos factores de traslación al gl.uniform

Ubicando la posición de los controles

Para que la GUI no nos tape el canvas, podemos pintarlo al final de la página, añadiendo este código después de crear los controles:

```
// Apply CSS styles to locate in the bottom side of the window
var guiContainer = document.querySelector('.dg.main');
guiContainer.style.position = 'fixed';
guiContainer.style.bottom = '0';
guiContainer.style.left = '0';
```

Ahora se nos colocará en la punta de abajo, tanto en escritorio como en el móvil.

Usando matrices de transformación 4x4

Si quisiéramos añadir rotaciones y escalados de esta misma manera, el código del shader sería complicado, porque habría que recibir vectores de traslación, ángulos de rotación, factores de escala, y tener que estar tranformando los vértices a partir de todos estos datos. Sin embargo, usando matrices de transformación, el código del shader queda mucho más simple, porque solo tiene que multiplicar cada vértice por una matriz y listo.

Antes de empezar a mirar las matrices, debemos mirar las diapos de la **clase 3-3: *matrices en WeGLI*.**

NOTA IMPORTANTE: En OpenGL, los puntos no son vectores fila como en las diapos de teoría, sino **vectores columna**. Por tanto, un punto P no es una fila (x,y,z,1) sino una columna. Y por tanto, la multiplicación de matrices no se hace por la derecha, es decir, P' = P * M, sino por la **izquierda** (para que el álgebra de matrices funciones). O sea, P' = M * P.

Además esto implica otro cambio. Las matrices de transformación M no son tal cual las hemos visto en teoría, sino que **son las traspuestas** (si hacemos una multiplicación en papel lo veremos enseguida). Todo esto se puede ver en el documento ***IG3-3 Matrices en WebGL*** que está en el campus virtual.

$$\begin{pmatrix} & M & \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} x_m \\ y_m \\ z_m \\ w_m \end{pmatrix}$$

Usando matrices en el vertex shader

Comenzamos quitando el uniform anterior del shader para la traslación, y lo sustituimos por otro uniform donde le va a llegar una matriz 4x4. Esta nueva matriz **se le multiplicará al vértice en coordenadas homogéneas por la izquierda**.

```
uniform mat4 uModelMatrix;
void main(void) {
     gl_Position = uModelMatrix * vec4(aCoordinates, 0.0, 1.0);
```

Además, debemos crear el puntero a esta variable dentro del shader:

```
var modelMatrixLoc;
modelMatrixLoc = gl.getUniformLocation(shaderProgram,
"uModelMatrix");
```

Usando la librería de matrices para trasladar

Para usar matrices en webgl usaremos la librería glMatrix, que se encuentra en: https://github.com/toji/gl-matrix en la carpeta "dist".

Nos tenemos que descargar el fichero gl-matrix-min.js, y añadirlo a nuestro proyecto glitch. Y a continuación debemos añadir dos líneas al código. La primera es para Importar la librería en el head del html:

```
<script type="text/javascript" src="gl-matrix-min.js"></script>
```

La segunda es para indicar las variables globales que nos ofrece la librería, y poder usarlas directamente en el código. Colocaremos esta línea al principio de nuestro script:

```
const { vec2, vec3, mat3, mat4 } = glMatrix;
```

Una hez hecho esto, la variable global mat4 nos dará acceso a una serie de métodos para poder trabajar con matrices 4x4, y poder usar funciones para trasladar, escalar y rotar. Así que comenzamos creando una matriz vacía en el método render, que inicializaremos a la identidad, y donde iremos añadiendo todas las transformaciones dadas por la GUI en una única matriz.

```
// Set the model Matrix.
modelMatrix = mat4.create();
mat4.identity(modelMatrix);
mat4.translate(modelMatrix, modelMatrix,
        [settings.translateX, settings.translateY, 0]);
gl.uniformMatrix4fv(modelMatrixLoc, false, modelMatrix);
```

Añadimos slider para la rotación

Añadimos un slider para el ángulo a rotar. Como vamos a rotar en 2D por ahora, será una rotación en eje z.

```
var settings = {
   translateX: 0.0,
   translateY: 0.0,
   rotateZ: 0.0
 };

var gui = new dat.GUI();
gui.add(settings, 'translateX', -1.1, 1.1, 0.01);
gui.add(settings, 'translateY', -1.1, 1.1, 0.01);
gui.add(settings, 'rotateZ', -180, 180);
```

Multiplicamos por la matrix de rotación en Z después de trasladar.

```
mat4.rotateZ(modelMatrix, modelMatrix,
```

```
        settings.rotateZ/180*Math.PI);
```

Y finalmente el vertex shader no cambia. No hay que añadir info de ángulos ni nada. La misma matriz de antes lleva implícita toda la concatenación de transformaciones

NOTA: El orden de las matrices es importante!!

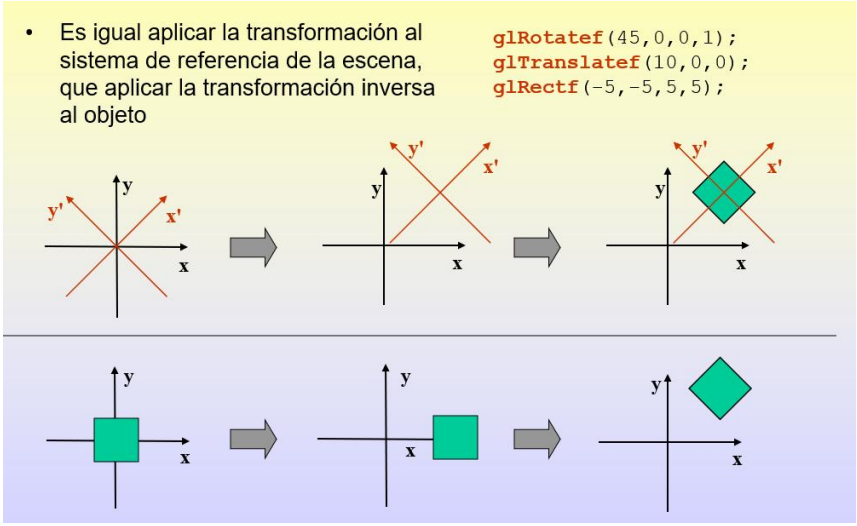
Fijémonos en la instrucción del vertex shader:

```
gl_Position = uModelMatrix * vec4(aCoordinates, 0, 1);
```

La matriz se multiplica por la izquierda, es decir , P' = M * P.

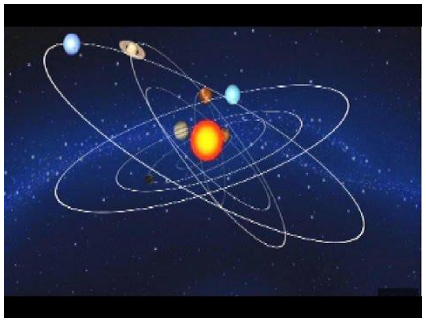
A su vez, la matriz se formó comenzando con la identidad, multiplicándola primero por una traslación, y en segundo lugar por una rotación. Por tanto, M = T * R, y finalmente tenemos que P' = T * R * P. Esto significa que, desde el punto de vista de P, el cual va siendo multiplicado paso a paso por diferentes matrices de transformación simples, primero se rota, y luego se traslada!. Y es por eso que en pantalla, el cuadrado rota alrededor del origen, y luego se traslada.

Si lo hiciéramos al revés…



Comenzando un sistema solar

Vamos entonces a intentar pintar un sistema solar, con el sol en el centro de la escena, y luego los planetas y sus satélites



Pintando el sol

Vamos a renombrar el objeto ball, y lo llamamos “sun”, con su posición central, su tamaño y su color:

```
   var sun = {
       'x':0, 'y':0,
       'width':0.2, 'height':0.2,
       'color':[1,1,0,1],
     }
```

Y finalmente lo pintamos en el render, después de las transformaciones hechas por la GUI, y antes de dibujar el cuadrado.

```
// draw sun
mat4.scale(modelMatrix, modelMatrix, [sun.width, sun.height,
1]);
gl.uniformMatrix4fv(modelMatrixLoc, false, modelMatrix);
gl.uniform4fv(colorLocation, sun.color);
drawSquare();
```

Pintando la tierra

Para pintar la tierra, creamos un objeto earth

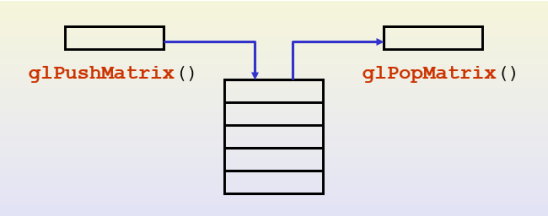
```
var earth = {
       'x':0.6, 'y':0,
       'width':0.1, 'height':0.1,
       'color':[0.2,0.2,1,1],
     }
```

Y pintamos en el render después del sol.

```
// draw earth
mat4.translate(modelMatrix, modelMatrix, [earth.x, earth.y, 0]);
mat4.scale(modelMatrix, modelMatrix, [earth.width, earth.height,
1]);
gl.uniformMatrix4fv(modelMatrixLoc, false, modelMatrix);
gl.uniform4fv(colorLocation, earth.color);
drawSquare();
```

Pero qué es lo que pasa. La tierra se ve muy chiquita y muy pegada, porque le afecta el escalado previo del sol. Si intento arreglarlo **reseteando la identidad** antes de ponerme con las transformaciones de la tierra, entonces pierdo las transformaciones de la GUI

La pila de matrices



Necesitamos una pila para recuperar un estado anterior. Así que nos creamos un array de pila, y nos creamos las funciones glPushMatrix y glPopMatrix

```
var matrixStack = [];

function glPushMatrix() {
       const matrix = mat4.create();
       mat4.copy(matrix, modelMatrix);
       matrixStack.push(matrix);
     }

function glPopMatrix() {
       modelMatrix = matrixStack.pop();
     }
```

Así que ahora el código render queda

```
// draw sun
glPushMatrix();
     mat4.scale(modelMatrix, modelMatrix,
          [sun.width, sun.height, 1]);
     gl.uniformMatrix4fv(modelMatrixLoc, false, modelMatrix);
     gl.uniform4fv(colorLocation, sun.color);
     drawSquare();
glPopMatrix();

// draw earth
glPushMatrix();
     mat4.translate(modelMatrix, modelMatrix,
          [earth.x, earth.y, 0]);
     mat4.scale(modelMatrix, modelMatrix,
          [earth.width, earth.height, 1]);
     gl.uniformMatrix4fv(modelMatrixLoc, false, modelMatrix);
     gl.uniform4fv(colorLocation, earth.color);
     drawSquare();
glPopMatrix();
```

Animando la tierra

Vamos a meterle un campo “angle” a la tierra, y lo iremos incrementando en cada render.

```
// draw earth
earth.angle += 0.01;
mat4.rotateZ(modelMatrix,modelMatrix,earth.angle);
```

OJO que rotamos antes de trasladar. Cambia el orden en el código para que veas lo que pasa

Metiendo la luna

Creamos un objeto luna

```
var moon = {
  'x':0.2, 'y':0,
  'width':0.05, 'height':0.05,
  'color':[1,1,1,1],
  'angle':0
}
```

Teóricamente sería copiar el trozo de render que tenemos para la tierra, exactamente igual pero con la variable luna. Pero alguna cosilla más habrá que cambiar….. (pista: piensa bien las ubicaciones de los push y popmatrix)

Posibles ampliaciones

- Añadir rotación en otros ejes en la GUI para verlo desde otra perspectiva
- Añadir escalado en la GUI para simular zoom de la cámara
- Añadir otros planetas con otros satélites
- Añadir órbitas que no estén en el plano XY (haciendo un rotate previo en X o en Y)
- Dibujar las órbitas de cada planeta y satélite

Para dibujar la curva de las órbitas se aconseja crear una función drawCircle, que pinte un círculo de radio unidad con la primitiva LINE_LOOP (la cual va pintando segmentos rectos entre cada par de vértices consecutivos). Y ya luego llamando a las transformaciones correspondientes de traslación y escalado, dibujamos la órbita donde queramos