Práctica 2: Transformaciones 2D, vectores y matrices

Profesor: Pedro Xavier Contla Romero
Ayudante: Melissa Méndez Servín
Ayudante de laboratorio: Joel Espinosa Longi

Fecha de entrega: 7 de marzo de 2020

Transformaciones 2D

Las transformaciones geométricas más básicas que tenemos, son la traslación, la rotación y el escalamiento.

Vamos a considerar a nuestros objetos geométricos, como elementos que viven en un espacio de dos dimensiones los cuales puede ser descritos a partir de un conjunto de vértices que los conforman. Cada uno de los vértices (o puntos) está determinado por sus coordenadas (x, y).

Un escalamiento representa un cambio en las distancias entre los vértices de un objeto geométrico. El cambio de las distancias está determinado por un movimiento en la posición de los vértices, establecido por un factor de escalamiento. Entonces, para calcular las nuevas coordenadas (x', y') del vértice (x, y) escalado, se realiza lo siguiente:

$$x' = x * sx$$
$$y' = y * sy$$

Una rotación representa un giro (generalmente alrededor del origen) de un objeto geométrico. Como es un giro, las rotaciones involucran el uso de funciones trigonométricas. Para calcular las nuevas coordenadas (x', y') del vértice (x, y) rotado en un ángulo θ, se realiza lo siguiente:

$$x' = x \cdot cos(\theta) - y \cdot sen(\theta)$$
$$y' = x \cdot sen(\theta) + y \cdot cos(\theta)$$

Una traslación representa un cambio en la posición de un objeto geométrico. Como es un cambio de posición, una traslación puede ser descrita como la adición o sustracción de un factor numérico para el desplazamiento en cada componente de los vértices. Entonces, para calcular las nuevas coordenadas (x', y') del vértice (x, y) trasladado, se realiza lo siguiente:

$$x' = x + tx$$
$$y' = y + ty$$

Todas estas transformaciones pueden ser descritas por medio de una matriz, donde los elementos de las matrices están determinadas por los coeficientes por los cuales se multiplica cada componente de los vértices.

Un escalamiento está determinado por la siguiente matriz:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{bmatrix} sx & 0 \\ 0 & sy \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Una rotación por:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{bmatrix} cos(\theta) & -sen(\theta) \\ sen(\theta) & cos(\theta) \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Y una traslación por:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} tx \\ ty \end{pmatrix}$$

Si observamos la matriz de traslación, los factores de desplazamiento (tx, ty) que corresponden la información sobre la traslación, se encuentra fuera de la matriz, esto se debe a que la dimensión de la matriz no es suficiente para representar esta información. Esto nos plantea la necesidad de describir una traslación utilizando una matriz de 3×3, en lugar de la matriz de 2×2 que se utilizó, dando la siguiente matriz:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Entonces, podemos especificar las tres transformaciones geométricas básicas en dos dimensiones utilizando matrices de 3×3, lo que nos proporciona un marco homogéneo de representación. Solo es necesario agregar una tercera componente a las coordenadas de los vértices, para poder operar con estas matrices.

Este mismo razonamiento es lo que se utiliza para tres dimensiones, como se verá más adelante durante el curso.

Entonces, utilizando lo visto en la clase van a desarrollar las clases: `Vector3` y `Matrix3`, cada una de ellas definidas en su propio archivo javascript: `Vector3.js` y `Matrix3.js`, respectivamente. Las clases serán utilizadas por medio de los módulos de JavaScript (ecmascript 6) por lo que es necesario la utilización de `export`, por ejemplo la clase Vector3 se declara de la siguiente manera:

```
export default class Vector3 {
// aquí va el código de la clase Vector3

}
```

Y desde otros archivos se utiliza un `import` de la siguiente manera:

```
import Vector3 from "./Vector3.js";
```

Vector3

La clase `Vector3` representa vectores de tres componentes, x, y y z. Aunque serán utilizados para representar vértices de objetos en dos dimensiones.

El constructor recibe 3 parámetros numéricos y crea un objeto que representa un vector de tres componentes (`constructor(x, y, z)`), en caso de no recibir valores en los argumentos, devuelve el vector $(0, 0, 0)$.

```
/**
 * @param {Vector3} u
 * @param {Vector3} v
 * @return {Vector3}
 */
static add(u, v)
```

`add` es una función que devuelve la suma de sus argumentos.

```
/**
 * @return {Vector3}
 */
clone()
```

`clone` es una función que devuelve un objeto el cual contiene los mismos valores que el objeto desde el cual se invocó la función.

```
/**
 * @param {Vector3} u
 * @param {Vector3} v
 * @return {Vector3}
 */
static cross(u, v)
```
`cross` es una función que devuelve el producto cruz de sus argumentos.

```
/**
* @param {Vector3} u
* @param {Vector3} v
* @return {Number}
*/
static distance(u, v)
```
`distance` es una función que devuelve la distancia euclidiana que hay entre sus argumentos.

```
/**
* @param {Vector3} u
* @param {Vector3} v
* @return {Number}
*/
static dot(u, v)
```
`dot` es una función que devuelve el producto punto de sus argumentos.

```
/**
 * @param {Vector3} u
 * @param {Vector3} v
 * @return {Boolean}
 */
static equals(u, v)
```
`equals` es una función que devuelve verdadero en caso de que sus argumentos sean aproximadamente iguales (con una $\varepsilon = \mathbf{0.000001}$), y falso en caso contrario.

```
/**
 * @param {Vector3} u
 * @param {Vector3} v
 * @return {Boolean}
 */
static exactEquals(u, v)
```
`exactEquals` es una función que devuelve verdadero en caso de que sus argumentos sean exactamente iguales y falso en caso contrario.

```
/**
 * @return {Vector3}
 */
normalize()
```
`normalize` es una función que devuelve el vector resultado de la normalización del vector que invoca la función.

```
/**
 * @param {Number} x
 * @param {Number} y
 * @param {Number} z
 */
set(x, y, z)
```
`set` es una función que asigna nuevos valores a los componentes del vector con que se llama.

```
/**
 * @param {Vector3} u
 * @param {Vector3} v
 * @return {Number}
 */
static squaredDistance(u, v)
```
`squaredDistance` es una función que devuelve la distancia euclidiana al cuadrado que hay entre sus argumentos.

```
/**
 */
zero()
```
`zero` es una función que asigna cero a cada componente del vector que invoca la función.

Matrix3

La clase `Matrix3` representa matrices de 3×3. Y se utilizará para la representación y construcción de transformaciones en dos dimensiones.

El constructor recibe 9 parámetros numéricos, `Matrix3(a00, a01, a02, a10, a11, a12, a20, a21, a22)` y construye la siguiente matriz:

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}$$

En caso de no recibir valores en los argumentos, devuelve la matriz identidad:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
/**
 * @param {Matrix3} m1
 * @param {Matrix3} m2
 * @return {Matrix3}
 */
static add(m1, m2)
```

`add` es una función que devuelve la suma dos matrices.

```
/**
 * @return {Matrix3}
 */
adjoint()
```

`adjoint` es una función que devuelve la matriz adjunta (o matriz de cofactores), de la matriz con que se invoca la función.

```
/**
 * @return {Matrix3}
 */
clone()
```

`clone` es una función que devuelve un objeto el cual contiene los mismos valores que el objeto desde el cual se invocó la función.

```
/**
 * @return {Number}
 */
determinant()
```
`determinant` es una función que devuelve el determinante de la matriz.

```
/**
 * @param {Matrix3} m1
 * @param {Matrix3} m2
 * @return {Boolean}
 */
static equals(m1, m2)
```
`equals` es una función que devuelve verdadero en caso de que sus argumentos sean aproximadamente iguales (con una $\varepsilon = \mathbf{0.000001}$) y falso en caso contrario.

```
/**
 * @param {Matrix3} m1
 * @param {Matrix3} m2
 * @return {Boolean}
 */
static exactEquals(m1, m2)
```
`exactEquals` es una función que devuelve verdadero en caso de que sus argumentos sean exactamente iguales, y falso en caso contrario.

```
/**
 */
identity()
```
`identity` es una función que asigna los valores de la matriz identidad a la matriz desde donde se invocó la función.

```
/**
 * @return {Matrix3}
 */
invert()
```
`invert` es una función que devuelve la matriz inversa de la matriz con la que se invocó la función.

```
/**
 * @param {Matrix3} m1
 * @param {Matrix3} m2
 * @return {Matrix3}
 */
static multiply(m1, m2)
```
`multiply` es una función que devuelve la multiplicación de dos matrices.

```
/**
 * @param {Matrix3} m1
 * @param {Number} c
 * @return {Matrix3}
 */
static multiplyScalar(m1, c)
```

`multiplyScalar` es una función que devuelve una matriz que es el resultado de multiplicar cada componente por un escalar.

```
/**
 * @param {Vector3} v
 * @return {Vector3}
 */
multiplyVector(v)
```

`multiplyVector` es una función que devuelve el vector resultado de multiplicar el vector `v` por la matriz con que se llama la función. Esta función es la que nos va a permitir realizar las transformaciones.

```
/**
 * @param {Number} theta
 * @return {Matrix3}
 */
static rotate(theta)
```

`rotate` es una función que devuelve una matriz de 3×3 que representa una transformación de rotación en `theta` radianes.

```
/**
 * @param {Number} sx
 * @param {Number} sy
 * @return {Matrix3}
 */
static scale(sx, sy)
```

`scale` es una función que devuelve una matriz de 3×3 que representa una transformación de escalamiento, con el factor `sx` como escalamiento en x y `sy` como escalamiento en y.

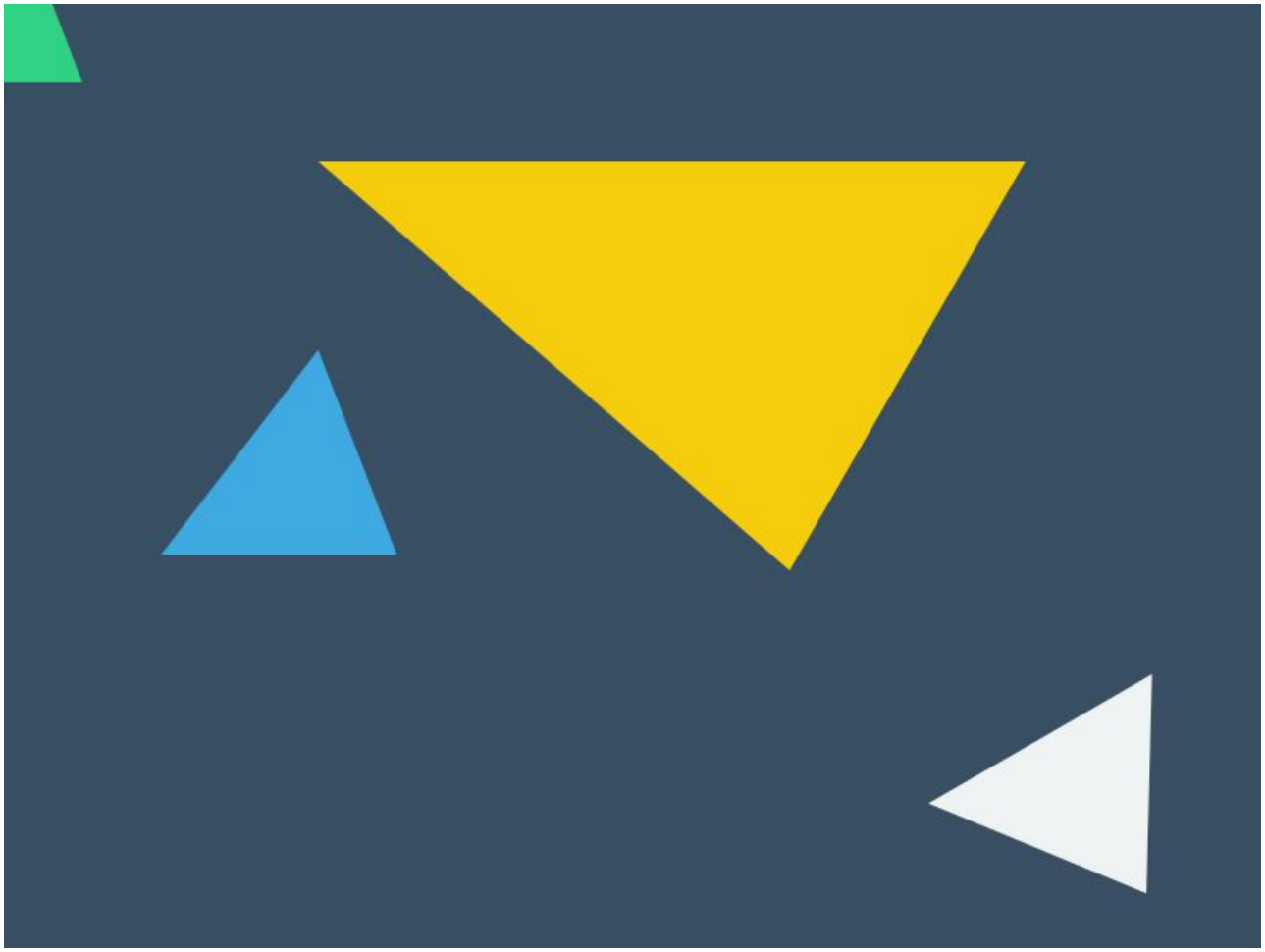
```
/**
 * @param {Number} a00
 * @param {Number} a01
 * @param {Number} a02
 * @param {Number} a10
 * @param {Number} a11
 * @param {Number} a12
 * @param {Number} a20
 * @param {Number} a21
 * @param {Number} a22
 */
set(a00, a01, a02, a10, a11, a12, a20, a21, a22)
```
`set` es una función que asigna nuevos valores a los componentes de la matriz con que se llama.

```
/**
 * @param {Matrix3} m1
 * @param {Matrix3} m2
 * @return {Matrix3}
 */
static subtract(m1, m2)
```
`subtract` es una función que sustrae componente a componente la matriz `m2` de la matriz `m1`.

```
/**
 * @param {Number} tx
 * @param {Number} ty
 * @return {Matrix3}
 */
static translate(tx, ty)
```
`translate` es una función que devuelve una matriz de 3×3 que representa una transformación de traslación, con `tx` como la traslación en x y `ty` como la traslación en y.

```
/**
 * @return {Matrix3}
 */
transpose()
```
`transpose` es una función que devuelve la matriz transpuesta de la matriz desde donde se invocó la función.

Junto con el archivo pdf de la práctica, se adjuntan los archivos **`index.html`** y **`Main.js`** los cuales utilizan los archivos `Vector3.js` y `Matrix3.js` para su funcionamiento. Por lo que parte de su práctica consiste en la correcta implementación de las clases para que los archivos se ejecuten y muestren la siguiente imagen:



Notas de entrega

- Recuerden que para que funcionen los módulos de JavaScript deben utilizar un servidor local. Como les comente en el laboratorio mi recomendación es que utilicen nodejs y http-server.
- La práctica se realizará y entregará de manera individual por medio del ClassRoom del curso.
- Todos los archivos de su práctica deben estar contenidos en un archivo zip, con su número de cuenta como nombre de archivo.
- Podrán entregar la práctica después de la fecha establecida para la entrega (7 de marzo de 2020), pero por cada día de retraso se penalizará con un punto menos. Por este motivo la fecha de entrega es inamovible.
- El código debe estar documentado, en caso contrario se penalizará la calificación.