UNIVERSIDAD NACIONAL DEL LITORAL
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS

Tecnicatura en diseño y programación de videojuegos

UNL VIRTUAL

Prototipado rápido

Unidad 3
Movimiento, *inputs* y colisiones

Docente
Ariel Cid

CONTENIDO

1. Movimiento

La Unidad 2 finalizó con la descripción de cómo una entidad podía mostrarse en pantalla con una imagen que la representara. En esta Unidad se continúa con el desarrollo del código de ese prototipo, complejizando su funcionamiento para aprender ciertos conceptos claves.

El primero de los temas a abordar es el movimiento de las entidades en pantalla. Cada entidad posee dos atributos que indican la posición de la misma (*x* e *y*). Modificando estos atributos a lo largo del tiempo se puede desplazar cada entidad, pero hay que tener en cuenta lo que se mencionó en la unidad anterior sobre los *timesteps*: el desplazamiento respeta la fórmula de física que se adecúe a las necesidades -por ejemplo la del desplazamiento con movimiento rectilíneo uniforme- pero debe considerar que el factor temporal puede medirse en *frames* (*timestep* fijo) o en segundos (*timestep* variable).

En el primer caso, se actualizará la posición de la entidad Player en su método *update*, usando un valor arbitrario (se podría definir como constante en otro lugar pero se omite para mantener el foco en este tema).

```
override public function update():void
{
        x += 5;
        super.update();
}
```

Ese código se ejecutará una vez por *frame*, por lo que se obtendrá un desplazamiento igual a esa velocidad por la cantidad de FPS definida para el juego (5*30 = 150 pixels por segundo).

En el caso de que se use un *timestep* variable, como se había optado por usar en la unidad anterior, el código será similar aunque con la inclusión del tiempo transcurrido entre cuadro y cuadro. Se usará para esto la clase FP, que contiene un parámetro llamado *elapsed*, que refleja siempre ese intervalo de tiempo.

```
override public function update():void
{
        x += 5 * FP.elapsed;
        super.update();
}
```

Este método dependerá de la velocidad de procesamiento de la máquina, pero con seguridad se debería aumentar el valor de la velocidad debido a que FP.elapsed tiende a ser de magnitud muy pequeña.

2. Input handling

Habiendo visto a grandes rasgos la lógica del movimiento en *timestep* variable, queda por ver cómo este puede responder a la interacción con el jugador. Se usan dentro de *update* las clases Input y Key para verificar la presión de diferentes teclas.

```
override public function update():void
{
        if(Input.check(Key.RIGHT))
        this.x += 100 * FP.elapsed;

        super.update();
}
```

El método *check* devolverá *true* si la tecla se encuentra presionada en ese *frame*. Para distintos comportamientos hay otros métodos dentro de Input; por ejemplo *pressed* devolverá *true* si la tecla se presionó en ese *frame* (o sea, sólo en el cuadro en que la tecla baja). Hay que contemplar también que el jugador debe "chocar" contra los límites de la pantalla para no salirse de la misma (ya que todavía no manipulamos la cámara). El código completo para las distintas entradas del jugador quedaría de esta manera:

```
override public function update():void
{
        if (Input.check(Key.RIGHT))
        {
                if ((this.x + playerImage.width) < FP.screen.width)
        x += 200 * FP.elapsed;
                else
                        this.x = FP.screen.width - playerImage.width;

        }

        if (Input.check(Key.LEFT))
        {
                if (this.x > 0)
                        x -= 200 * FP.elapsed;
                else
                        this.x = 0;
        }

        if (Input.check(Key.DOWN))
        {
                if ((this.y + playerImage.height) < FP.screen.height)
                        y += 200 * FP.elapsed;
                else
                        this.y = FP.screen.height - playerImage.height;
        }

        if (Input.check(Key.UP))
        {
                if (this.y > 0)
                        y -= 200 * FP.elapsed;
                else
                        this.y = 0;
        }
        super.update();
}
```

3. Colisiones

El último apartado de esta unidad se destina a repasar algunas maneras básicas de detectar y manejar colisiones entre entidades.

En primera instancia se describen los métodos más sencillos, las colisiones por cajas y por tipos. Las entidades tienen un método llamado *setHitbox* que permite *setear* de forma sencilla el área que colisionará con otras entidades; cuando el código pide chequear si hay colisión de este tipo, FlashPunk verifica si hay al menos dos entidades cuyas cajas o *hitboxes* interseccionan. Este tipo de chequeos es algo impreciso (porque las cajas siempre serán rectángulos), pero mucho mejor en rendimiento que el segundo método que se muestra luego (por máscaras, donde se chequeará pixel por pixel si dos entidades colisionan).

Colisiones por cajas

Para tener una entidad contra la cual colisionar, se crea la clase Enemy, que hereda de Entity (al cual se le ha agregado previamente un comportamiento básico para poder visualizarlo rápidamente).

```
public class Enemy extends Entity
{
        [Embed(source="enemy.png")]
        private const ENEMY_IMG:Class;
        private var enemyImg:Image;

        public function Enemy(px:Number = 0, py:Number = 0)
        {
                enemyImg = new Image(ENEMY_IMG);
                enemyImg.color = 0xDD0000;

                setHitbox(enemyImg.width, enemyImg.height);
                type = "Enemy";

                super(px, py, enemyImg);
        }

        override public function update():void
        {
                y += 200 * FP.elapsed;

                if (y > FP.screen.height)
                        world.remove(this);
        }
}
```

El *GameWorld* generará entonces enemigos en posiciones aleatorias.

```
override public function update():void
{
        if (classCount(Enemy) < 10)
{
                for (var i:uint = 0; i < 10; i++)
                        this.add(new Enemy(
FP.rand(FP.screen.width - player.getWidth()),
-FP.rand(400) - player.getHeight()
));
        }
        super.update();
}
```

El Player deberá también definir un *hitbox* y un *type* en su constructor

```
setHitbox(16, 16);
type = "Player";
```

En este punto, las entidades ya se encontrarán en condiciones de chequear si colisionan. Dentro del *update* de *Enemy* se incluirán las siguientes líneas con tal fin:

```
if (this.collide("Player", this.x, this.y))
FP.world = new GameWorld();
```

Si bien es algo brusco, el código anterior reinicia el juego al crear nuevamente el mundo cuando el jugador colisiona con un enemigo, a esto lo hace para mostrar el funcionamiento de las colisiones simples y las consecuencias de que dicha colisión se realice.

Colisiones por máscaras

Las colisiones por máscaras funcionan de manera muy similar, aunque los resultados son radicalmente diferentes. Para esto, se reemplazan las líneas de *Player* y *Enemy* que llaman al método *setHitbox* por otra línea que crea una nueva *Pixelmask*.

```
mask = new Pixelmask(PLAYER_IMG);
```

Y lo mismo en *Enemy*:

```
mask = new Pixelmask(ENEMY_IMG);
```

De esta manera, al ejecutar el juego se podrá ver que la colisión ahora será de pixel contra pixel, en vez de caja contra caja. Si bien este modo es mucho más preciso (y ayuda a ciertos géneros específicos de juego, como los *shoot 'em up* o *plataformers*), consume más recursos en cada chequeo de colisión.