Tutorial de Vista y Cámara

Sumario

Ver previamente tutoriales básicos para más información sobre creación básica de objetos, manejando el reloj, jerarquía de fotogramas y animaciones.

Este tutorial muestra como usar múltiples vistas con múltiples cámaras.
Cuatro vistas son creadas aquí.

La esquina superior izquierda con una vista(`Viewport1`) y la esquina inferior derecha con una vista(`Viewport4`) comparten la misma cámara.
Para archivar esto, necesitamos usar el mismo nombre en el fichero de configuración para la cámara.

Además, cuando manipulando esta cámara usando los botones izquierdo y derecho del ratón para hacerla girar, las flechas para moverla y dejar presionado control y shift izquierdos para ampliarla, las dos ventanas asociadas con esta cámara se verán afectadas.

La vista(`Viewport2`) superior derecha está basada en otra cámara (`Camera2`) que truncada(EN) es más estrecha que la primera, resultando en una pantalla el doble de grande. No puedes afectar la vista en tiempo de ejecución en este tutorial.

La última vista (`Viewport3`) esta basada en otra cámara(`Camera3`) quien tiene la misma configuración que la primera.

Esta vista mostrará lo que originalmente tenga en `Viewport1` & `Viewport4` antes de modificar su cámara.
Tu puedes también interactuar directamente con las propiedades de la primera vista, usando las teclas WASD para moverla y Q & E para redimensionarla.

PD: Cuando dos vistas se superponen, la antigua (ej. una creada primero que la otra) se mostrará encima.

Por último, tenemos una caja que no se mueve del todo, y un pequeño soldado cuya posición en el mundo será determinado por la actual posición del ratón en la pantalla.
En otras palabras, no importa en cual vista esté tu ratón, y no importa cómo la cámara para esta vista se fija, el soldado siempre tiene sus pies en la misma posición que el puntero del ratón en la pantalla (siempre y cuando sea en una vista).

Vistas y objectos son creados con tamaños y colores aleatorios usando el carácter '~' en el fichero de configuración.

PD: Las cámaras almacenan su posición/zoom/rotación en una estructura orxFRAME, permitiendo esto ser parte de la jerarquía orxFRAME vista en el tutorial de fotogramas.
Como resultado, el objeto de auto-seguimiento se puede lograr mediante el ajuste del objeto como padre de la cámara.
En la otra mano, teniendo una cámara como padre de un objeto se asegurará de que el objeto se mostrará siempre en el mismo lugar en la correspondiente ventana[1].

Detalles

Como es usual, comenzaremos por cargar nuestro fichero de configuración, obteniendo el reloj principal y registrando nuestra función `Update` a él y, por último, creando nuestro objeto principal. Por favor, referirse al tutorial anterior para más detalles.

Sin embargo crearemos cuatro vistas esta vez. Nada realmente nuevo, por lo que solo necesitaríamos escribir este código.

```
pstViewport = orxViewport_CreateFromConfig("Viewport1");
orxViewport_CreateFromConfig("Viewport2");
orxViewport_CreateFromConfig("Viewport3");
orxViewport_CreateFromConfig("Viewport4");
```

Como puedes ver solo mantenemos una referencia a una vista creada. Lo hacemos porque queremos interactuar con ella más adelante, pero no tocaremos las otras tres.

Vamos a ir directamente a nuestro código de actualización `Update`.
En primer lugar ajustaremos a nuestro soldado a la posición del ratón. Ya hemos visto tal cosa en el tutorial de fotogramas.
Aquí haremos exactamente los mismo y veremos como trabaja perfectamente con múltiples vistas. Cuando el ratón no esta sobre la vista, `orxNULL` se devuelve en lugar de los valores del puntero en las coordenadas globales.

```
orxVECTOR vPos;

if(orxRender_GetWorldPosition(orxMouse_GetPosition(&vPos), &vPos) !=
orxNULL)
{
  orxVECTOR vSoldierPos;

  orxObject_GetWorldPosition(pstSoldier, &vSoldierPos);
  vPos.fZ = vSoldierPos.fZ;

  orxObject_SetPosition(pstSoldier, &vPos);
}
```

Antes de interactuar directamente con la vista, juguemos un poco sus cámaras asociadas.
Podemos, por ejemplo, moverla, rotarla o hacer zoom.

Comenzemos por obtener nuestra primera cámara de la vista.

```
pstCamera = orxViewport_GetCamera(pstViewport);
```

OK, eso es fácil. Intentemos rotarla [2)].

```
if(orxInput_IsActive("CameraRotateLeft"))
{
  orxCamera_SetRotation(pstCamera, orxCamera_GetRotation(pstCamera) +
```

```
orx2F(-4.0f) * _pstClockInfo->fDT);
}
```

De nuevo, veremos que nuestra rotación no será afectada por el FPS y puede ser alargado en el tiempo como mismo usamos el reloj DT.
Todavía pudieramos utilizar la configuración del sistema para obtener la velocidad de rotación en vez de hacerlo por código! 😀

Hagamos zoom, ahora.

```
if(orxInput_IsActive("CameraZoomIn"))
{
  orxCamera_SetZoom(pstCamera, orxCamera_GetZoom(pstCamera) * orx2F(1.02f));
}
```

Como este código no utiliza ninguna información del reloj, el se mantendrá afectado por la frecuencia del reloj y por las imágenes por segundo.

Por último, movamos nuestra cámara.

```
orxCamera_GetPosition(pstCamera, &vPos);

if(orxInput_IsActive("CameraRight"))
{
  vPos.fX += orx2F(500) * _pstClockInfo->fDT;
}

orxCamera_SetPosition(pstCamera, &vPos);
```

Ahora hemos terminado de jugar con la cámara.
Como veremos un poco más adelante en este tutorial, la misma cámara está enlazada a dos vistas diferentes.
De este modo ambas serán afectadas cuando jugamos con ellas.

En cuanto a las interacciones directas con las vistas, podemos alterar su tamaño u posición, por ejemplo.
Podemos hacer esto de esta manera, por ejemplo.

```
orxFLOAT fWidth, fHeight, fX, fY;

orxViewport_GetRelativeSize(pstViewport, &fWidth, &fHeight);

if(orxInput_IsActive("ViewportScaleUp"))
{
  fWidth *= orx2F(1.02f);
  fHeight*= orx2F(1.02f);
}

orxViewport_SetRelativeSize(pstViewport, fWidth, fHeight);
```

```
orxViewport_GetPosition(pstViewport, &fX, &fY);

if(orxInput_IsActive("ViewportRight"))
{
  fX += orx2F(500) * _pstClockInfo->fDT;
}

orxViewport_SetPosition(pstViewport, fX, fY);
```

Nada realmente sorprendente como puedes ver.

Echemos ahora un vistazo al lado de los datos de nuestras vistas.

```
[Viewport1]
Camera            = Camera1
RelativeSize      = (0.5, 0.5, 0.0)
RelativePosition  = top left
BackgroundColor   = (0, 100, 0) ~ (0, 255, 0)

[Viewport2]
Camera            = Camera2
RelativeSize      = @Viewport1
RelativePosition  = top right
BackgroundColor   = (100, 0, 0) ~ (255, 0, 0)

[Viewport3]
Camera            = Camera3
RelativeSize      = @Viewport1
RelativePosition  = bottom left
BackgroundColor   = (0, 0, 100) ~ (0, 0, 255)

[Viewport4]
Camera            = @Viewport1
RelativeSize      = @Viewport1
RelativePosition  = bottom right
BackgroundColor   = (255, 255, 0)#(0, 255, 255)#(255, 0, 255)
```

Como podemos ver, nada sorprendente aquí tampoco.
Tenemos 3 cámaras para 4 vistas, donde usamos la `Camera1` para ambas `Viewport1` y `Viewport4`.
Podemos notar que nuestras vistas comienzan con un tamaño relativo de `(0.5, 0.5, 0.0)`. Esto significa que cada vistas usará la mitad del tamaño horizontal y vertical de la pantalla (la coordinada Z es ignorada).
En otras palabras, cada vista cubre exactamente un cuarto de nuestra pantalla, cualquier tamaño que escojamos para esta última, pantalla completa o no.

Como puedes notar, solo damos un valor explicito para el tamaño relativo `RelativeSize` para nuestra vista `Viewport1`.
Todas las otras vistas heredan el tamaño relativo de la primera vista(`Viewport1` `RelativeSize`) cuando escribimos `@Viewport1`.
Esto quiere decir que este valor será el mismo tomado desde la primero vista(`Viewport1`) con la misma llave(`RelativeSize`).

Haremos exactamente de la misma manera para la cámara de la cuarta vista(`Viewport4`'s `Camera`) usando `@Viewport1`.

Entonces necesitamos ubicarlas en pantalla para prevenir que todas se muestren unas encima de las otras. Para hacer esto, usamos la propiedad posición relativa `RelativePosition` que puede ser tomada de un valor literal [3] o un vector de la misma manera que lo hicimos para tamaño relativo(`RelativeSize`.).

Por último, las primeras tres vistas usan diferentes sombras para sus colores de fondo(`BackgroundColor`).
Por ejemplo,

```
BackgroundColor = (200, 0, 0) ~ (255, 0, 0)
```

significa que la vista usará una sombra roja aleatoria [4].

Si queremos un color más preciso de fondo pero manteniendo la aleatoriedad, podemos usar una lista como en

```
BackgroundColor = (255, 255, 0)#(0, 255, 255)#(255, 0, 255)
```

Esto nos brinda tres posibilidades de cuatro colores aleatorios, amarillo, cyan y magenta.

Finalmente, echémosles un vistazo a nuestras cámaras.

```
[Camera1]
FrustumWidth  = @Display.ScreenWidth
FrustumHeight = @Display.ScreenHeight
FrustumFar    = 1.0
FrustumNear   = 0.0
Position      = (0.0, 0.0, -1.0)

[Camera2]
FrustumWidth  = 400.0
FrustumHeight = 300.0
FrustumFar    = 1.0
FrustumNear   = 0.0
Position      = (0.0, 0.0, -1.0)

[Camera3@Camera1]
```

Básicamente definimos sus truncados frustum(EN) (ej. la parte del mundo será *vista* por la cámara y rendereada en la vista).
PD: Como estamos usando cámaras 2D, la forma del truncado es un cuboide rectángular.

Note que la tercera cámara(`Camera3`) hereda de la primera cámara(`Camera1`) pero no anula ninguna propiedad: ellos tienen exactamente la misma propiedad.
PD: Cuando la herencia es usada para toda la sección, es escrita de esta manera: [MySection@ParentSection].
Por qué usamos dos cámaras diferentes entonces?. Solo como para tener dos entidades físicas: cuando alteramos la propiedad de la primera cámara(`Camera1`) en nuestro código, la tercera

cámara(`Camera3`) se mantendrá sin cambios.

Podemos notar que el alto por ancho del truncado de la primera cámara(`Camera1`'s `FrustumWidth` y `FrustumHeight`) heredan de la configuración de la pantalla.
PD: Cuando la herencia es usada por un valor, es escrita como esto `MyKey = @ParentSection.ParentKey`.
La llave padre puede ser omitida si esta es la misma que nuestra llave: `SameKey = @ParentSection`.

Por último note que nuestra segunda cámara(`Camera2`) tiene un menor truncado.
Esto significa que la segunda cámara(`Camera2`) verá una porción más pequeña del mundo. Por lo tanto la correspondiente vista se mostrará como si estuviese `zoomeada`! 😃

Recursos

1)
muy útil para hacer HUD & UI, por ejemplo
2)
solo una parte del código se muestra, falta aún la lógica
3)
compuesto por palabras arriba, abajo, centro, derecha e izquierda; `top`, `bottom`, `center`, `right` y `left`
4)
el caracter '~' es usado como un operador aleatorio entre dos valores numéricos

From:
https://www.orx-project.org/wiki/ - **Orx Learning**

Permanent link:
https://www.orx-project.org/wiki/es/orx/tutorials/viewport?rev=1330620860

Last update: **2025/09/30 17:26 (14 months ago)**