Interacción
Detección de Colisiones

Sistemas Gráficos

Grado en Ingeniería Informática

Curso 2019/2020

1. Introducción

Una parte fundamental de cualquier aplicación es la interacción con el usuario. Aunque la interacción persona-ordenador es un tema para una asignatura completa, en este tema abordaremos los aspectos de la interacción con el usuario directamente relacionados con la particularidad de los gráficos por ordenador.

Además de la interacción típica de un usuario con la aplicación a través de menús, botones o en general, la interfaz de usuario que se haya diseñado para la aplicación, en una escena gráfica aparecen unas particularidades como pueden ser:

- La selección de un objeto de la escena, lo que se conoce como Picking.
- La selección de una posición en la escena, es decir, sus coordenadas (x,y,z).
- La modificación de un objeto particular como es la cámara, que afecta directamente a lo que se está visualizando.

Hay que tener en cuenta que una vez que se ha visualizado un frame, lo que el usuario tiene en pantalla es una imagen, una matriz rectangulas de píxeles de colores. Que ha salido del resultado de procesar un grafo de escena, un modelo más complejo y con más información,

pero que en definitiva lo que hay en pantalla es una imagen. Y que cuando un usuario hace clic con el ratón para seleccionar lo que el usuario ve como una figura, en realidad, solo ha seleccionado un pixel, que además es 2D.

Tenemos que poder saber, a partir de esa posición 2D en la imagen, que se corresponde con una determinada figura en el modelo, en el grafo de escena.

Además, la figura que se ha seleccionado es muy probable que pertenezca a un objeto mayor. Por ejemplo, si hacemos clic sobre la mano de un personaje, ¿en realidad quería elegir una mano para moverla? ¿o quería elegir el brazo entero y ha hecho clic en una parte del brazo como es la mano? ¿o tal vez quería seleccionar el personaje completo?

Y cuando se ha seleccionado la figura, ¿qué se desa hacer con ella? ¿moverla, girarla?

O si el usuario selecciona un suelo, ¿está seleccionando el suelo como un todo? tal vez lo que le interese sea una posición concreta en ese suelo (coordenada 3D) porque se va a colocar ahí alguna cosa.

Como veis, *entrar* con el ratón 2D en la escena 3D a través de la imagen 2D que se está visualizando y saber lo que quiere hacer el usuario no es un asunto trivial.

Por otro lado, los objetos también pueden interactuar unos con otros. Si tenemos un juego de billar, el taco interacciona con las bolas golpeándolas. Las bolas colisionan unas con otras y hace que se modifique su trayectoria y su velocidad. Cuando las bolas llegan a una tronera, caen porque ha dejado de existir la base sobre la que iban rodando.

Hablar de interacción en la escena supone hacerlo en dos ámbitos, usuario-escena y objeto-objeto en la misma escena.

Además, en una escena puede llegar a haber un número considerable de elementos. Las operaciones de picking y de detección de colisiones implican, en última instancia, hacer búsquedas de objetos en la escena. Teniendo en cuenta la eficiencia, habría que realizar algún tipo de indexación entre los objetos de la escena que permita acelerar dichas búsquedas.

En Prado se ha dejado un ejemplo que permite:

- Manejar una grúa de obra con la GUI.
- Añadir unas cajas en el suelo con el ratón.
- Seleccionar cajas para moverlas o girarlas.
- Si en el movimiento de una caja colisiona con otra, se detecta esa colisión y la caja que se movía se apila encima de la caja con la que ha colisionado.

- Se pueden enganchar cajas con la grua y soltarlas en otro lado.

El ejemplo, figura 1, incluye varias técnicas de las que se abordan en este tema, para que lo tengáis como referencia y veais una implementación de algunos de los conceptos que se explican en el tema a un nivel más teórico.

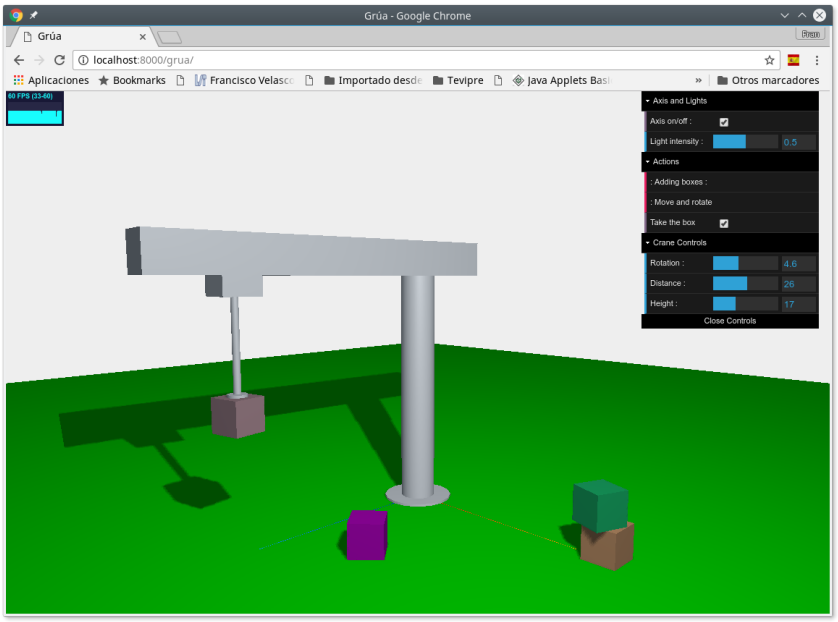


Figura 1: Esta sencilla aplicación implementa diversas técnicas de interacción.

2. Entrada/salida de información por parte del usuario

2.1. Interacción con el ratón en la escena

El ratón tiene varias características que hacen que su procesamiento requiera un poco de atención.

- Se usa para todo.
- Tiene pocos botones.
- Es 2D, y se quiere interactuar con una escena 3D.
- Es un dispositivo orientado a eventos.

Empezando por el final, cuando se mueve el ratón se genera un evento, si se pulsa un botón, se genera un evento, si se suelta se genera otro. Y eso ocurre cuando el usuario decide hacer un clic, no cuando la aplicación se lo solicita.

Es habitual, en una interacción tradicional mediante teclado, que la aplicación se detenga y le pida al usuario que escriba un dato. Mientras tanto la aplicación no hace otra cosa hasta que el usuario escribe lo que sea y pulsa `enter`.

Con una interacción orientada a eventos, la aplicación está funcionando y en paralelo hay unos *listeners* que están esperando a que se produzca su evento, y cuando eso ocurre, si tiene asociado una función, se ejecuta dicha función.

Por un lado, hay que decidir que eventos deseamos atender y asociarles métodos a dichos eventos.

Pero también decíamos que el ratón se usa para todo y tiene pocos botones. Por tanto, si el usuario hace un clic, puede ser para multitud de funcionalidades en la aplicación. El evento del clic solo tiene asociado un método, por lo tanto se está llamando a un solo método, el cual tiene que determinar qué es lo que quiere hacer el usuario, en qué contexto está haciendo ese clic, para realizar las llamadas a los métodos apropiados y dar respuesta a ese requerimiento concreto del usuario.

Se tiene que definir un conjunto de estados que van a indicar qué se está haciendo en la aplicación en cada momento.

Así, **un método asociado a un evento del ratón, de manera general, implementará el siguiente algoritmo**:

1. Consultar en que estado se encuentra actualmente la aplicación.
2. Consultar la información que está asociada a ese evento. Por ejemplo, si se ha hecho un clic, habrá que consultar con qué botón se ha hecho, y si estaba pulsada alguna tecla modificadora: Ctrl, Shift, Alt, o una combinación de ellas.
3. Con todo eso, determinar cuál es el procesamiento que se está solicitadon y realizar las llamadas apropiadas a los métodos que correspondan. Añadir una figura, mover la cámara, apagar una luz, etc.

Eventos del ratón que pueden escucharse

Los eventos del ratón que pueden escucharse son, como os podéis imaginar, pulsar un botón, soltarlo (que generan dos eventos distintos), moverlo, y hacer girar la rueda.

Sus nombres concretos son:

- mousedown
- mouseup
- mousemove
- wheel

Como veis, no tenemos el evento arrastrar el ratón. Si queremos programar algo asociado al arrastre del ratón, hay que programarlo en el evento de movimiento y comprobar si el botón está pulsado.

Para cada evento que quiera capturarse y procesarse hay que añadir un listener en el main.

```
window.addEventListener ("mousemove", (event) => objeto.metodo (event));
```

El primer parámetro, entrecomillado, indica el evento que se desea escuchar, y el segundo parámetro indica la función que se llamará cuando se produzca el evento. En el ejemplo, al querer llamar a un método de un objeto y no a una función, lo que se hace es establecer un enlace entre una función anónima con un parámetro, `(event)` con el método concreto del objeto concreto que será llamado, con los mismos parámetros, `objeto.metodo(event)`.

En el cuerpo de ese método, que recibe el objeto `event` como parámetro, se pueden consultar los atributos del objeto `event` para tomar decisiones. Algunos de esos atributos son:

- `clientX`: La coordenada X del ratón
- `clientY`: La coordenada Y del ratón, ambas relativas a la esquina superior izquierda de la ventana, y en coordenadas de dispositivo. Es decir, la unidad de ese sistema de referencia es el píxel.
- `which`: El botón concreto que se ha pulsado. Otro entero.
  - 0: Ninguno.
  - 1: El izquierdo.
  - 2: El central.
  - 3: El derecho.

- `ctrlKey`: Un booleano que indica si se tenía pulsada la tecla modificadora `Ctrl` cuando se produjo el evento.
- `altKey`: Un booleano. Lo mismo con respecto a la tecla modificadora `Alt`.
- `shiftKey`: Un booleano. Idem con respecto a la tecla modificadora `Shift`.

Por ejemplo, se desea que el clic del ratón, con `Ctrl` pulsado haga un movimiento de cámara, y cuando no se tenga `Ctrl` pulsado haga otras cosas. La implementación sería así.

```
onMouseDown (event) {
  if (event.ctrlKey) {
    // Se habilitan los movimientos de cámara
    scene.getCameraControls().enabled = true;
    // No se hace otra cosa
  } else {
    scene.getCameraControls().enabled = false;
    // Se desabilitan los movimientos de cámara
    // Se realiza otro procesamiento
    . . .
  }
}
```

Estados de la aplicación

Conocer qué se está haciendo en cada momento para que los métodos asociados a los eventos del ratón sepan cómo tienen que interpretar esos clics y movimientos es fácil, basta con tener definidas unas constantes que representen los distintos estados en los que se puede encontrar una aplicación y almacenar en un atributo de la clase principal el estado concreto que corresponda en cada momento.

Cuando desde la aplicación, con alguna elección de un menú o con cualquier modo que haya de indicar órdenes, se indica que se va a hacer algo concreto, se cambia el estado de la aplicación y así, cuando desde un método asociado a un evento del ratón se consulte dicho estado se sabrá qué se está haciendo y por tanto cómo debe interpretarse dicho evento y su información asociada.

Veamos un ejemplo:

```
// Los estados se pueden definir como constantes numéricas
// Si la clase principal de la aplicación es  TheScene
// Cuando se cierre la clase, y antes del main, se definirían esas constantes así
TheScene.NO_ACTION = 0;
TheScene.ADDING_BOXES = 1;
TheScene.MOVING_BOXES = 2;

// En el constructor de la clase principal,  TheScene en este ejemplo,
//    se inicializaría el atributo que indicará el estado actual de la aplicación
this.applicationMode = TheScene.NO_ACTION;

// En cualquier método asociado a un evento del ratón se puede consultar
//    el estado actual de la aplicación para tomar decisiones
onMouseDown (event) {
  switch (this.applicationMode) {
    case TheScene.ADDING_BOXES :
      // procesamiento para el evento  mouseDown  y el estado  ADDING_BOXES

      . . .
```

¿En qué momento se cambió `this.aplicationMode` de `TheScene.NO_ACTION` a `TheScene.ADDING_BOXES`? Tal vez cuando el usuario indicó en la GUI que quería añadir cajas en la escena. De modo que los clic que haga a partir de ahora deben interpretarse como sitios donde se desea añadir una caja.

¿Y cuándo salimos de ese modo? Pues depende de cómo se haya diseñado esa operación. Si se ha establecido que solo se añade una caja, cuando se acabe la operación se vuelve a cambiar el estado de la aplicación. O si se ha establecido que se añaden cajas hasta que se diga de algún modo que se ha finalizado, pues habrá que especificar cuál es ese modo de indicar el fin. Tal vez le corresponda a un clic con el botón derecho del ratón, o tal vez pulsando la tecla escape, o cualquiera de las dos acciones.

En definitiva, **el manejo de los estados de la aplicación se convierte en un aspecto muy a tener en cuenta en el diseño de la aplicación**.

2.2. Órdenes mediante teclado

El dar órdenes mediante teclado, en el sentido de lo que conocemos como atajos de teclado, no difiere mucho de lo que hemos comentado con relación a los eventos del ratón, puesto que el teclado también genera eventos y su definición y procesamiento es similar a los eventos del ratón.

Se añaden los listeners que quieran capturarse y se enlazan con los métodos que deban ejecutarse cuando ocurran tales eventos.

Los eventos de teclado son:

- `keydown`: Se produce cuando se pulsa una tecla, cualquiera.
- `keyup`: Se produce cuando se suelta una tecla, cualquiera.
- `keypress`: Se produce cuando se pulsa y se suelta una tecla. Un solo evento al soltar. Este evento no lo producen las teclas modificadoras.

El atributo del objeto `event` en el que leer qué tecla es la que ha producido el evento varía de unos navegadores a otros; pero con la siguiente línea de código se puede tener en la variable `tecla` la tecla concreta que ha producido el evento.

```
var tecla = event.which || event.keyCode
```

En `tecla` lo que tenemos es un entero que identifica a la tecla pulsada. En `https://www.w3schools.com/charsets/ref_html_utf8.asp` tenéis la lista completa de códigos.

Aunque, para saber más cómodamente si se ha pulsado una tecla con un caracter imprimible se puede usar le método de clase `String.fromCharCode (tecla)` que devuelve un `String` que se puede comparar con una cadena con una letra. Por ejemplo:

```
if (String.fromCharCode (tecla) == "A") . . .
```

Mensajes en pantalla

Cuando se quieren mostrar mensajes al usuario, se puede tener una zona en pantalla dedicada para ello, y desde la aplicación escribir ahí.

En el `html` se podría poner algo así:

```
<div style="position:absolute; left:100px; top:10px" id="Messages">
</div>
```

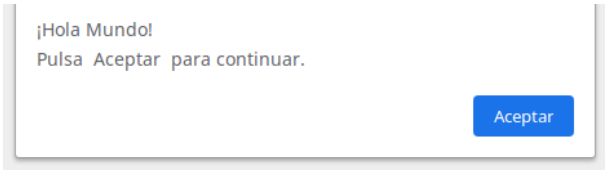
Y en la aplicación, en javascript, podemos tener un método para mostrar mensajes en esa zona.

```
setMessage (str) {
  document.getElementById ("Messages").innerHTML = "<h2>"+str+"</h2>";
}
```

Como veis, lo que se envía a esa área que se ha identificado con el nombre “Messages” es directamente código HTML. Con toda la flexibilidad que eso supone en cuanto a formato.

Si lo que se quiere es mostrar la típica ventana pop-up con un botón “Aceptar” y que todo se detenga hasta que se acepte, tenemos una forma básica de hacerlo con el método `alert` del objeto `window`.

```
window.alert("Hola Mundo!\nPulsa Aceptar para continuar.");
```



3. Selección de objetos: Picking

Hay una interacción del usuario con la aplicación que, desde la llegada del ratón, es fundamental: es que el usuario señale directamente el elemento con el que interactúa, ver figura 2.

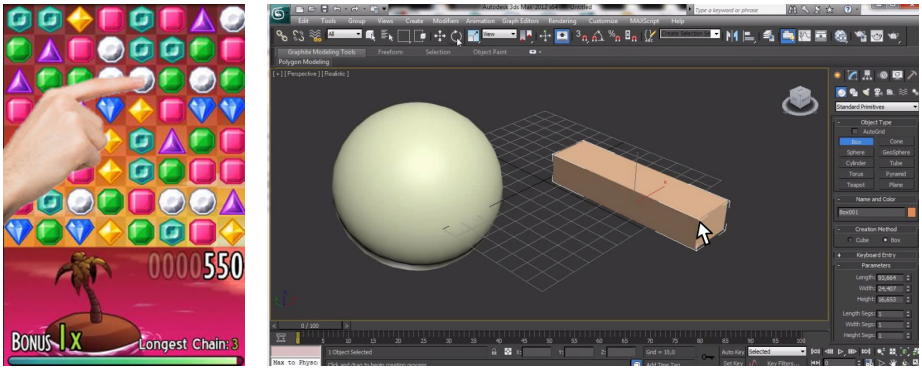


Figura 2: La selección directa de objetos, ya sea en 2D o en 3D, con el dedo o con otro dispositivo es una operación muy habitual.

El dispositivo que se use no supone ninguna diferencia ya que tanto si es el dedo como si es un ratón o un lápiz todos producen lo mismo, una posición 2D en la ventana de la aplicación. Lo que si requiere un tratamiento diferente es si lo que se está seleccionando es un elemento de una escena también en 2D, como en la imagen del juego de móvil, o es un elemento de una escena 3D, como en la imagen del modelador de sólidos.

En ambos casos hay que realizar un cambio de sistema de referencia, la entrada suele ser una posición entera 2D, centrada en el píxel concreto que se ha seleccionado. Y el objeto

seleccionado está en el sistema de coordenadas del mundo de la escena gráfica, que puede también ser 2D, donde hay una relación 1 a 1, a un punto en la pantalla le corresponde un punto en la escena; pero que puede ser 3D, donde la relación es 1 a infinito, a un punto en la pantalla le corresponden infinitos puntos en la escena.

Afortunadamente el proceso de determinar qué infinitos puntos de la escena le corresponden al punto en pantalla clicado por el usuario no es difícil. Véase la figura 3.

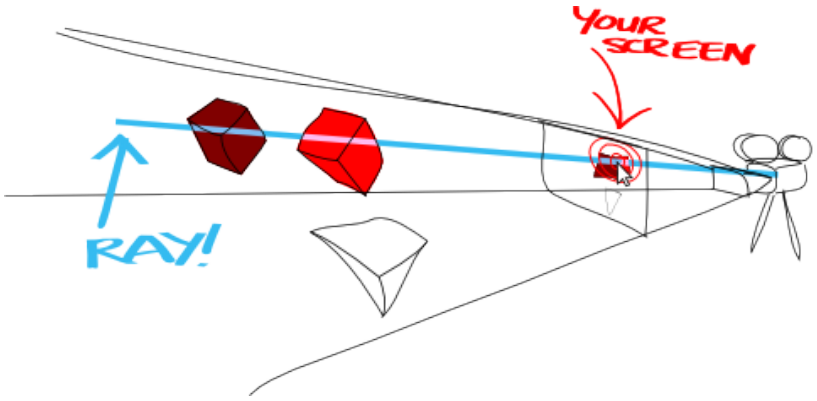


Figura 3: Tanto la proyección de la imagen como la seleccción de figuras se basa en lo mismo, los proyectores que llegan o parten del centro de proyección.

Sabemos que la imagen que ve el usuario de la escena es una proyección de la misma según una cámara, normalmente una proyección cónica. Es decir, la imagen se ha formado colocando una *imagen virgen*, sin color, entre la escena y la cámara. Entonces, desde cada punto de la escena se ha lanzado un rayo hacia la cámara, todos dirigidos al mismo punto, *el centro de proyección* y ahí donde atraviesa la *imagen virgen* impresiona un píxel con un color (como en una fotografía). Cuando se han procesado todos los rayos y todos los píxeles han quedado coloreados, se tiene la imagen de la escena.

Pues cuando se trata de determinar lo que está seleccionan un usuario cuando hace clic en su pantalla, solo hay que hacer el proceso a la inversa. Se lanza un rayo que parta del centro de proyección de la cámara y que pase por el píxel seleccionado. Ese rayo (formado por infinitos puntos) entra en la escena y aquellos elementos con lo que se cruce el rayo son los candidatos a ser la elección del usuario.

Los candidatos estarán a distinta distancia, unos más cerca del observador y otros más lejos. El candidato que finalmente se considera que es la elección del usuario es el que esté más cerca de la cámara. Que por otro lado, es muy posible que sea el único que se vea ya que lo que está más cerca oculta la visión de los objetos que están en esa línea visual pero más lejos.

Dicho algoritmo, en código Three, es el mostrado en código 1.

```
1 onDocumentMouseDown (event) {
2   var mouse = new THREE.Vector2 ();
3   mouse.x = (event.clientX / window.innerWidth) * 2 - 1;
4   mouse.y = 1 - 2 * (event.clientY / window.innerHeight);
5
6   var raycaster = new THREE.Raycaster ();
7   raycaster.setFromCamera (mouse, this.camera);
8
9   var pickedObjects = raycaster.intersectObjects (this.pickableObjets, true);
10
11  if (pickedObjects.length > 0) {
12    var selectedObject = pickedObjects[0].object;
13    var selectedPoint =  new THREE.Vector3 (pickedObjects[0].point);
14    . . .
15  }
16 }
```

Código 1: Gracias a la clase Raycaster, la implementación del picking necesita de pocas líneas.

Expliquemos el código. Se trata del método asociado al evento `"mousedown"`. En las líneas 2 a 4 lo primero que se hace es extraer las coordenadas 2D del ratón y trasformarlas a otro sistema de coordenadas, tambien 2D pero distinto, que es el que asume la clase Raycaster, que va a procesar esa posición. Ver figura 4.

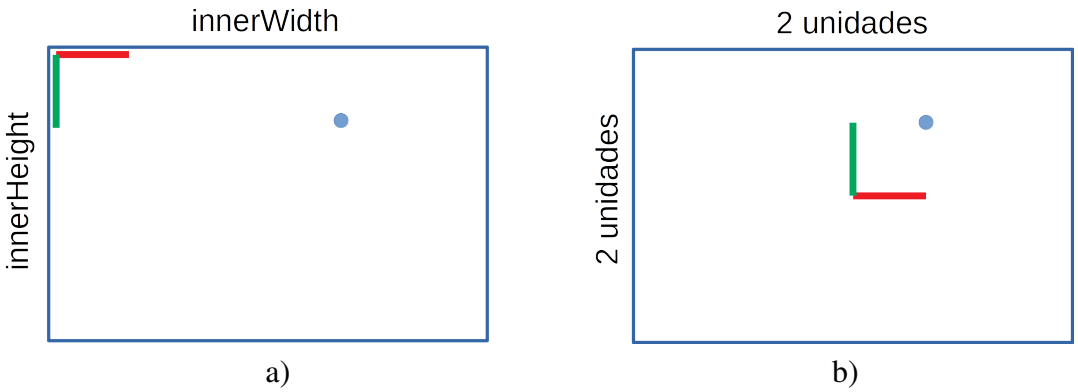


Figura 4: El sistema de referencia 2D que asume la clase Raycaster es distinto al que ha usado el ratón para indicar su posición.

Ambas imágenes, a y b, representan la imagen donde se muestra la escena, el punto azul representa el lugar donde el usuario ha hecho clic. La imagen a) muestra el sistema de coordenadas de dispositivo que con respecto al cual el puntero del ratón toma sus coordenadas. Es un sistema de referencia de unidades enteras, con el origen en la esquina superior izquierda y que como valor máximo en cada dimensión es el tamaño en píxeles de la ventana. Esos máximos los obtenemos de los atributos `innerWidth` e `innerHeight` del objeto `window` que nos proporciona el navegador como variable global.

Supongamos que esas dimensiones, en el ejemplo, son 600 x 400 píxeles. Por tanto, el punto azul tendría las coordenadas (400,100).

La imagen b) muestra la misma ventana pero con un sistema de referencia en unidades reales que asume que, el ancho de la ventan siempre es 2, el alto de la ventana siempre es 2, con el origen en el centro y, otro cambio, las Y positivas van hacia arriba. Si os fijáis, en la imagen b) las Y positivas del sistema de referencia van hacia abajo. En el sistema de referencia de la imagen b) toda la ventana está entre las coordenadas $(-1,-1)$ y $(1,1)$.

¿Qué coordenadas le corresponden al punto azul en este sistema de referencia? Veámoslo en la línea 3 del código 1. Si la x del ratón, `event.clientX`, puede variar entre 0 e *innerWidth*, si lo dividimos por *innerWidth* ya tenemos un valor que puede variar entre 0 y 1. Si ahora lo multiplicamos por 2, el valor podrá variar entre 0 y 2. Si le restamos 1, el valor podrá variar entre -1 y 1. Ya lo tenemos, la x que almacenanos el el punto 2D `mouse` que estamos calculando será 0,33; la que le corresponde según las condiciones del sistema de referencia de la imagen b).

En la línea 4 del código, se hace un cálculo parecido, la diferencia está en el hecho de que el sentido del eje Y del sistema de referencia ha cambiado. Por eso lleva una multiplicación extra por -1.

La posición del punto azul en coordenadas de dispositivo normalizadas es $(0{,}33 \quad , \quad 0{,}50)$.

Líneas 6 y 7. Solo se instancia el rayo que vamos a usar para buscar los objetos candidatos a ser seleccionados. Se crea el rayo y se configura teniendo en cuenta la cámara y la posición en coordenadas de dispositivo normalizadas donde se ha clicado.

La línea 9 es una llamada a un método de la clase Raycaster que busca que objetos son intersecados por el rayo. ¿Qué significan sus parámetros?

`this.pickableObjects` es una lista de nodos del grafo de escena. Esta operación de búsqueda de figuras que intersecan con un rayo es una operación costosa y no tendría sentido buscar intersecciones con figuras que no son seleccionables en ese momento. Por ejemplo, imaginaos un juego en el que el personaje está en una habitación y debe coger herramientas que puedan serle útiles para salir de la habitación o para otras fases del juego. Lo razonable sería que en la lista de figuras seleccionables solo estuvieran las herramientas que haya en ese momento en la habitación, y no meter ni muebles, ni decoración, ni otros personajes. Así, el algoritmo de intersección rayo-figura no pierde tiempo con figuras que no son seleccionables en ese momento. Esa lista de figuras seleccionables sería una lista dinámica donde vamos añadiendo o sacando elementos según sea necesario en cada momento.

El parámetro `true` indica que hay que buscar intersecciones en los nodos que hay en la lista y en sus descendientes, directos e indirectos. Este parámetro será `true` la mayoría de las veces. Como sabéis, una figura en el grafo de escena está representada por el nodo raiz del subgrafo que contiene todos los `Mesh` y `Objetc3D` que la definen. Los únicos elementos

que pueden intersecar con el rayo son los que tienen geometría. Por lo tanto, si un objeto seleccionable está representado por un subgrafo con varios nodos `Mesh`, metiendo su raiz en la lista y poniendo este parámetro a `true`, el algoritmo va a buscar intersecciones con todos los `Mesh` que directa o indirectamente cuelgan de esa raíz.

Lo que devuelve el método `intersectObjects` es una lista con los `Mesh` cuya geometría ha intersecado con el rayo. La lista está ordenada desde el más cercano al más lejano. En realidad la lista no es de objetos de la clase `Mesh` sino de otro tipo de objetos que contienen más información, no solo un `Mesh`.

Si la lista no está vacía, línea 11, sabemos que el `Mesh` intersecado más cercano estará en la posición 0 de la lista y bien podemos referenciar dicho `Mesh` a través del atributo `object` (línea 12) o bien podemos obtener las coordenadas exactas (en coordenadas del mundo) donde se produjo esa intersección del rayo con la geometría mediante el atributo `point` (linea 13), o ambas cosas.

Por ejemplo, si el usuario hace clic en una mesa para que un personaje deje ahí un libro que lleva, con el atributo `object` sabremos cual es el nodo `Mesh` concreto de esa mesa, y con el atributo `point` tenemos el punto exacto donde dejar el libro.

Accediendo a la figura global

Sin embargo, lo normal es que queramos acceder a la figura global, no a uno de tantos `Mesh` que componen un objeto global, como se ve en el ejemplo de la figura 5.

Figura 5: Un personaje, como muchos objetos, está formado por numerosas geometrías.

Imaginemos que en una escena tenemos diferentes personajes, y cada personaje que seleccione el usuario se va a poner a caminar.

El procedimiento sería fácil: previamente la aplicación ha entrado en el modo *caminar personajes*, cada vez que el usuario hace clic, el evento es capturado y procesado por el método asociado a dicho evento, en dicho método, tras comprobar que la aplicación está en el modo *caminar personajes* realiza un picking y obtiene el `Mesh` que ha clicado el usuario, y a dicho objeto se le envía el mensaje `caminar`. ¿Qué obtenemos? Un error del tipo `La clase THREE.Mesh no tiene un método denominado caminar.`

Se asume que en esta aplicación cada personaje es una instancia de una clase que tiene un método `caminar`.

Una instancia de dicha clase en realidad referencia al nodo raíz del subgrafo que representa al personaje y que descendiendo directa o indirectamente de dicho nodo raíz estarán varios nodos `Mesh`, y cualquiera de ellos ha podido ser seleccionado. En cada nodo `Mesh` se debe establecer una referencia a la raiz del subgrafo para poder referenciar dicho objeto y poder llamar a los métodos de la clase.

Eso se consigue con el atributo `userData` que tiene la clase `Object3D` y por herencia también lo tiene la clase `Mesh`. Cuando se esté construyendo el subgrafo de un personaje, se debe hacer que en cada `Mesh` que instanciemos, su atributo `userData` referencie a la raiz.

```
class DarthVader extends THREE.Object3D {
  constructor() {
    . . .
    this.cabeza = new THREE.Mesh ( . . . );
    this.cabeza.userData = this;
    . . .
  }
  caminar() {
    . . .
  }
  . . .
}
```

Ahora sí, cuando se esté procesando un picking y obtengamos un `Mesh` cualquiera de un personaje, sí podremos enviarle el mensaje `caminar` accediendo a la raiz del subgrafo de ese personaje a través de su atributo `userData`.

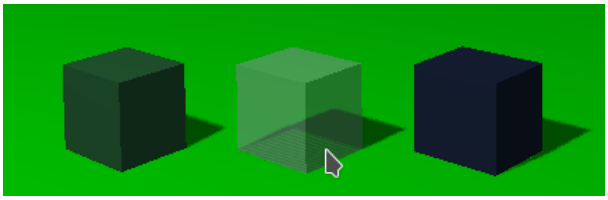
```
var meshClicado = pickedObjects[0].object;
meshClicado.userData.caminar();
```

Feedback

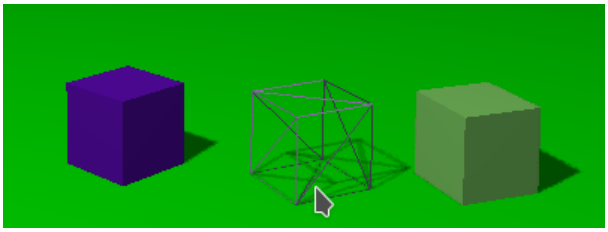
La interacción con el usuario no es unidireccional, desde el usuario hacia la aplicación, debe ser bidireccional. Por tanto, cada vez que el usuario realice la selección de un elemento de

la escena, la aplicación debe informarle de alguna forma que dicha selección se ha producido y debe mostrarle el objeto concreto que ha sido seleccionado.

La manera habitual es cambiar de alguna forma el aspecto de dicho elemento para que se distinga y el usuario confirme su entrada. Por ejemplo, se puede poner el objeto parcialmente transparente, o en modo alambre, como en la figura 6.



`opacity = 0.5 y transparent = true`



`wireframe = true`

Figura 6: Ejemplos de feedback: Transparencia y modo alambre. En ambos casos, solo hay que hacer un pequeño cambio en los atributos del material de la figura.

4. Detección de colisiones

Otro modo de interacción que nos encontramos en una aplicación es dentro de la propia escena, cuando unos objetos colisionan con otros. Se trata de una operación costosa. Pensad que habría que comprobar si la geometría de un objeto interseca con la geometría de los demás objetos; y eso, para los objetos que se están moviendo, habría que hacerlo en todo momento.

Para acelerar dicho proceso la detección de colisiones se realiza en dos fases:

Fase gruesa : Se implementa un algoritmo rápido que permite descartar los elementos que se sabe, al 100 %, que no colisionan.

Fase fina : Para los que exista alguna posibilidad de que colisionen, se determina la colision usando un algoritmo que puede llegar a ser 100 % preciso, aunque más lento.

No obstante, la precisión en el test de colisión viene determinada por los requerimientos de la aplicación. En ocasiones, una rápida detección de fase gruesa ya es suficiente para considerar que ha habido colisión.

La detección en la fase gruesa se basa en:

1. Tener la escena indexada espacialmente.
   La busqueda de los objetos de la escena con los que pueda colisionar un objeto móvil se acelera mucho si sabemos dónde están los objetos y podemos descartar rápidamente todos aquellos objetos que no se encuentran en la misma zona por donde se va posicionando el objeto móvil.
2. Usar cajas o esferas englobantes.
   Una vez seleccionados los objetos que están en la misma zona del objeto móvil, el test de colisión no se realiza con la geometría real de los objetos en sí, sino que se usan unas geometrías auxiliares, cajas o esferas, que envuelven **completamente** al objeto, figura 7.

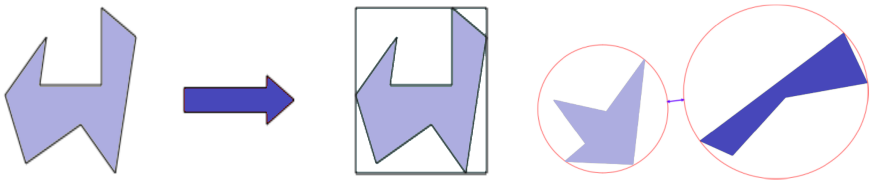


Figura 7: La caja o esfera englobante cubre por completo a la geometría original pero de la manera más ajustada posible.

El motivo es que es muy rápido determinar si dos eferas o dos cajas están intersecando. Y si dos esferas no intersecan, podemos asegurar que los objetos que envuelven tampoco van a intersecar, figura 8.

Os propongo como ejercicio que diseñéis sendos algoritmos para calcular la caja y la esfera englobante de un objeto cualquiera a partir de su lista de vértices.

4.1. Indexación espacial de la escena

En cuanto a la indexación espacial de la escena, se basa en realizar una descomposición recursiva del espacio de manera que queda dividido en sectores pudiendo determinar de una manera muy rápida qué elementos no están en el mismo sector, o bien del objeto móvil contra el cual se están buscando colisiones, o bien del rayo que se ha lanzado si se está haciendo un picking.

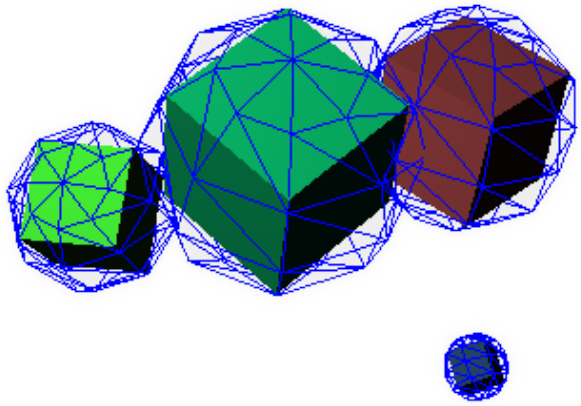


Figura 8: Si no hay intersección entre las esferas que envuelven a la geometría real, estas tampoco van a intersecar.

Se pueden emplear dos estrategias para la subdivisión: primando el que en cada subdivisión el número de elementos en cada parte esté equilibrado (Kd-Tree) o primando el que en cada subdivisión el volumen de cada parte esté equilibrado (Octree).

En lo que sigue, por facilidad a la hora de hacer y entender las ilustraciones, se mostrarán las ilustraciones en 2D. La explicación es totalmente válida para el 3D ya que en este contexto, la tercera dimensión no añade ninguna dificultad extra.

Kd-Trees

Según este criterio de subdivisión, se parte del volumen completo como único sector y en cada subdivisión el sector a subdividir de divide en 2 de manera que el número de elementos en cada sector resultante sea igual, o con una diferencia mínima. En cada uno de los sectores resultantes, si el número de elementos supera un mínimo establecido a priori, se vuelve a subdividir, así hasta que en todos los sectores del último nivel, el número de elementos no supere ese mínimo.

El plano de subdivisión (3D) o la línea de subdivisión (2D) cumple con las siguientes condiciones:

- Siempre está alineado con los ejes.
- El eje de alineación va alternando en cada nivel. Si se empezó dividiendo según el eje X, en el siguiente nivel se dividirá según el eje Y, en el siguiente según el eje Z, en el siguiente otra vez por el eje X y así sucesivamente.

Veamos un ejemplo en 2D. Los elementos de la escena a indexar son los números que

se muestran en la figura 9.a). Y se establece que el número de elementos máximo permitidos en un sector es 2. Habrá que subdividir un sector siempre que tenga más de 2 elementos.

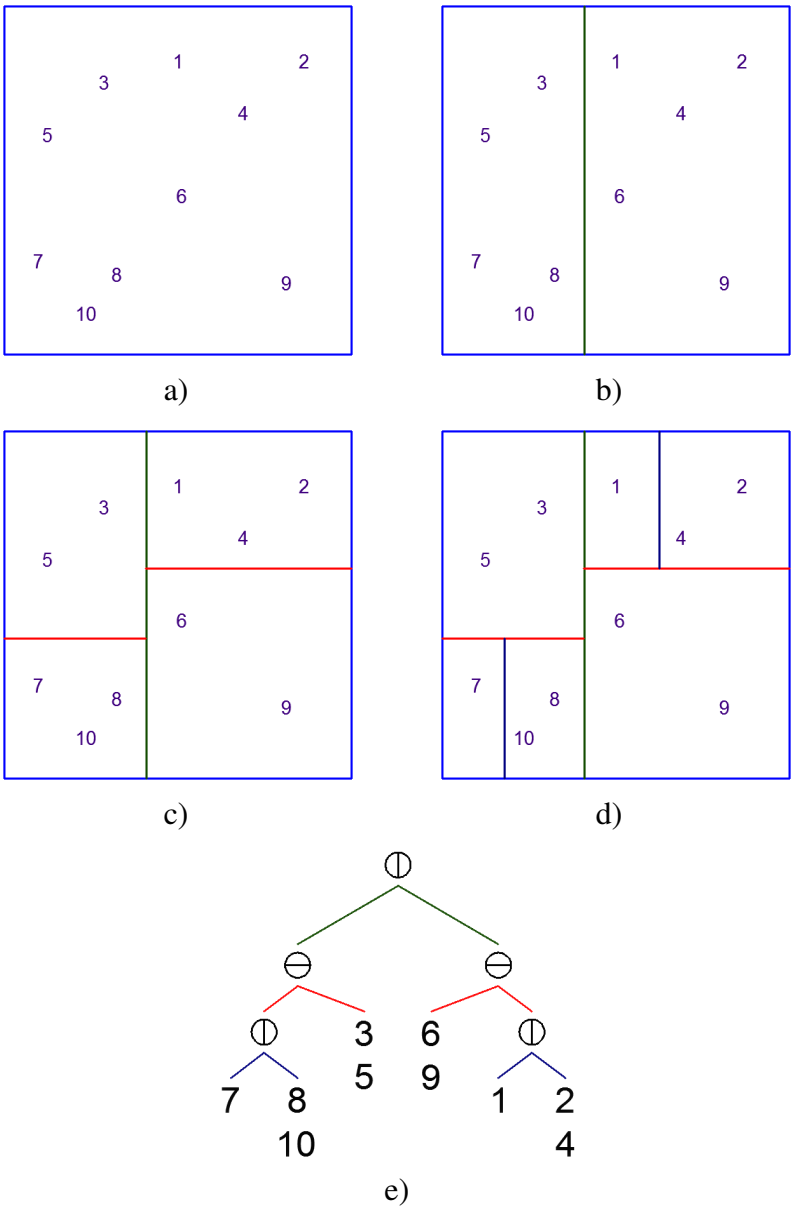


Figura 9: Proceso de subdivision según un Kd-tree 2D.

Se decide empezar con una subdivisión vertical, línea verde, entonces en el siguiente nivel tendrá que ser horizontal y en el siguiente nivel vertical otra vez y así de manera alternativa.

La subdivisión del primer nivel, línea verde, se posiciona de manera que a ambos lados quede el mísmo número de elementos (o con una diferencia máxima de 1, si son impares). Nos

da como resultado la figura 9.b). En el árbol, figura 9.e) obtenemos la raíz, con ese icono que representa una división vertical.

Evaluamos los dos sectores que nos han quedado en la figura 9.b) y todo el que tenga más de 2 elementos lo volvemos a dividir, ahora por líneas horizontales, las rojas, posicionando cada línea roja de manera que a sus dos lados quede el mismo número de elementos, salvo diferencia de 1 si son impares. Nos queda la figura 9.c) y en el árbol aparece el segundo nivel, con esos iconos que representan uns división horizontal.

Vemos, que las dos líneas rojas no están alineadas. No es obligatoria esa alineación. El requisito es que el número de elementos esté equilibrado.

Volvemos a evaluar los nuevos sectores y vemos que dos de ellos, superior izquierda e inferior derecha ya tienen solo dos elementos. Estos sectores no hay que subdividirlos y ello resulta en hojas en el árbol, las hojas $3,5$ y $6,9$.

Los otros sectores si superan el número máximo de elementos y sí hay que volver a subdividirlos. En ese nivel vuelve a tocar la subdivisión vertical (siempre alternando entre niveles), las líneas azules, posicionándolas de manera que a ambos lados de las líneas azules el número de elementos esté lo más equilibrado posible. Dando como resultado la figura 9.d) y el árbol ya finalizado como se ve en la figura 9.e), que no es más que un índice basado en un árbol binario, que como sabéis, permite búsquedas orden $O(\log n)$.

Cuando se busquen candidatos a colisionar con un elemento cualquiera x, solo hay que comprobar si ese elemento x está a la izquierda o a la derecha de la línea verde para descender en el árbol por la rama correcta y descartar la otra. Entonces, se comprueba si x está arriba o abajo de la línea roja que corresponda, y así sucesivamente de manera que con muy pocas comprobaciones rápidas se ha descartado a la inmensa mayoría de los elementos de la escena.

Octrees

Con este criterio de subdivisión, cada vez que en un sector se supera el número máximo de elementos permitidos, se subdivide en 4 cuadrantes (2D) u 8 octantes (3D) exactamente iguales entre sí en cuanto al área (2D) o volumen que ocupan (3D), y de manera recursiva se vuelven a subdividir los sectores que superen el número máximo de elementos permitidos. Figura 10. Resultando un árbol cuaternario (2D) u octal (3D).

Indexación espacial en Three

En Three se puede realizar indexación expacial mediante octrees, para ello se usa la biblioteca Octree.js, proporcionada junto con la práctica 1, y las Mesh a indexar tienen que haberse construido en base a geometrías CPU (las que no tienen la palabra Buffer en el nombre de su clase).

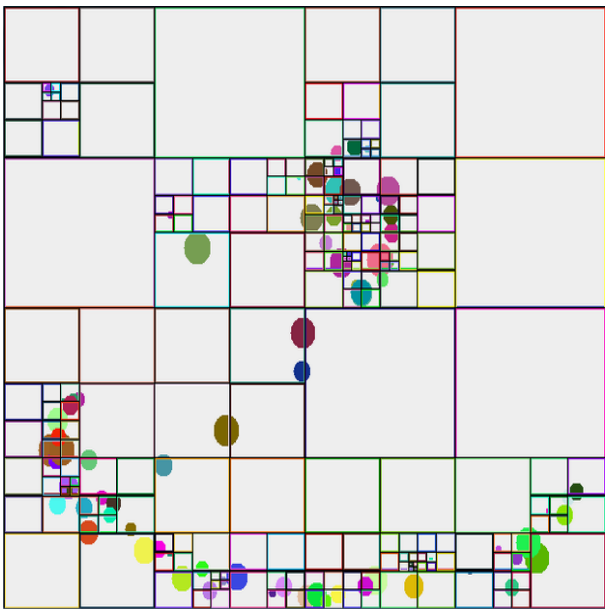


Figura 10: En cada subdivisión los sectores resultantes tienen la misma área entre sí.

Su uso es tan sencillo como:

1. Construir el árbol, vacío
2. Añadirle los elementos que se quieren indexar
3. Realizar las búsquedas cuando proceda
   - Cuando se haga picking, por ray casting.
   - Cuando se haga detección de colisiones, por cercanía a una posición.

La construcción del árbol no es más que instanciar la clase.

```
this.octree = new THREE.Octree ({
  undeferred: false,
    // Si undeferred es true, los objetos añadidos al árbol
    //   se insertan inmediatamente. Si es  false,
    //   se insertan cuando se haga  octree.update()
  depthMax: Infinity,
    // Se puede establecer una profundidad máxima
  objectsThreshold: 4,
    // Numero de objetos para subdividir un nodo
  overlapPct: 0.2
    // Porcentaje de solapamiento entre nodos
    //   Facilita la gestión de los objetos que están
    //   en la frontera de un nodo
  });
```

Poner `undeferred` a `false` mejora el rendimiento. Al poner un número máximo de objetos por sector, la profundidad máxima se puede dejar en infinito, nunca va a llegar a ser infinita. Los sectores pueden hacerse un poco más grandes de lo que le correspondería por una subdivisión estricta del espacio. Al ser ligeramente más grandes, hay un cierto solapamiento entre sectores contiguos. Ese solapamiento entre sectores facilita las búsquedas de las figuras que se encuentren justo en la frontera entre 2 sectores.

Una vez que el árbol está construido, solo hay que añadirle los `Mesh` que se deseen indexar. Se le puede indicar que considere las caras de la geometría, lo cual mejora las búsquedas.

```
this.octree.add ( this.unMesh, {useFaces:true});
```

Hay también que actualizar el octree en un método `update`. **Importante:** hay que actualizar el octree **después** de haber renderizado la escena.

```
update() {
  . . .
  this.renderer.render(this, this.getCamera());
  this.octree.update();
}
```

Cuando se haga un picking, se obtiene el conjunto de candidatos según el octree, y después se buscan cuales son intersecados por el rayo. Nótese como el metodo de búsqueda de objetos que están intersecados por el rayo (`intersectOctreeObjects`), es diferente al usado en la línea 9 del código 1, cuando se realizaba la búsqueda sin la mediación de un octree.

```
// Se parte de un objeto THREE.Raycaster
//   construido según la cámara actual y la posición del ratón
//   (consultar la sección Picking en estos apuntes)

// Se obtiene el conjunto de candidatos para el picking
var octreeObjects = this.octree.search (
  raycaster.ray.origin,
  raycaster.ray.far,
  true,
  raycaster.ray.direction
);

// Se busca cuales de esos candidatos son atravesados por el rayo
var intersections = raycaster.intersectOctreeObjects (octreeObjects);

// Si hay alguno, el más cercano está en  intersections[0]
```

El vídeo que se da junto con los ejercicios de la práctica 1 denominado `octree-rayCasting.mp4` muestra un ejemplo del resultado de usar un octree para mejorar el picking. En la escena hay 250 esferas y se va haciendo picking mientra se mueve el ratón. Sin octree, la busqueda de esferas que intersecan con el ray se realiza entre las 250 esferas. Cuando se usa el octree, la busqueda se realiza solo con unas 15 ó 20 esferas. Menos del 10 % del total.

Cuando se quiera buscar objetos con los que uno concreto podría colisionar, se usaría el octree para obtener una lista de candidatos.

```
// Suponer que el objeto que se está moviendo es
//   objetoMovil  y tiene un radio aproximado de  1 unidad

// Se obtiene el conjunto de candidatos para la búsqueda de colisiones
octreeObjects = octree.search (objetoMovil.position, 1, true);

// Se busca la posible colisión del objeto móvil
//   solamente con los objetos de la lista
```

El vídeo que se da junto con los ejercicios de la práctica 1 denominado `octree-colisiones.mp4` muestra un ejemplo del resultado de usar un octree para mejorar la búsqueda objetos que podrían colisionar según una fase gruesa que solo considera la indexación espacial. En la escena hay 250 esferas estáticas y una móvil, cuando se usa el octree, las esferas que son consideradas como posibles por la indexación son unas 5 ó 7 como máximo. Aproximadamente un 3 % del total.