Mundos tridimensionales

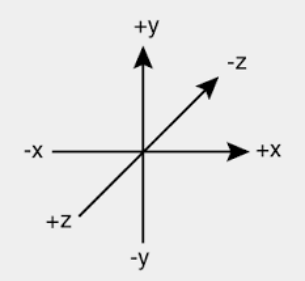
Introducción

En este post voy a introducir algunos conceptos básicos sobre los mundos tridimensionales. Vamos a enfocarlo des del punto de vista de los motores gráficos para juegos.
Los principales temas que vamos a tratar son:

- Coordenadas
- Espacio local (local space) y mundos (world space)
- Vectores
- Camaras
    - Projection mode (3D vs 2D)
- Plygons, edges, vertices and meshes
- Materials, textures and shader
- Rigidbody dynamics
    - Detección de colisiones

Coordenadas

En los mundos 3D se definen 3 coordenadas:



- Z-axis -> profundidad
- Y-axis -> altura
- X-axis -> anchura

Se representan de la siguiente manera (x,y,zy):

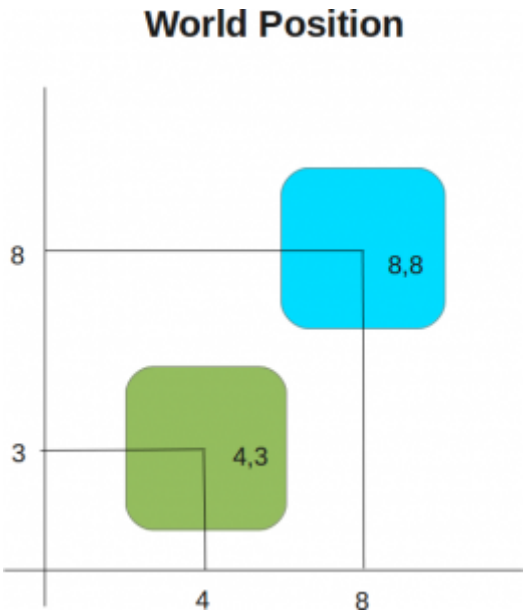
Local space (espacio local) y world space (mundo)

Es muy importante tener claras las diferencias entre local space y world space.

World space

En cualquier entorno 3D el world space será infinito por lo cual para poder referenciar la posición de los objetos deberemos marcar un punto llamado ‘origin’ o ‘world zero’. Este punto será la representación de la posición (0,0,0) .

Como podemos ver en esta imagen los dos objetos tienen sus respectivas posiciones en verso el punto (0,0), esto es igualmente aplicable a escenarios 3D.

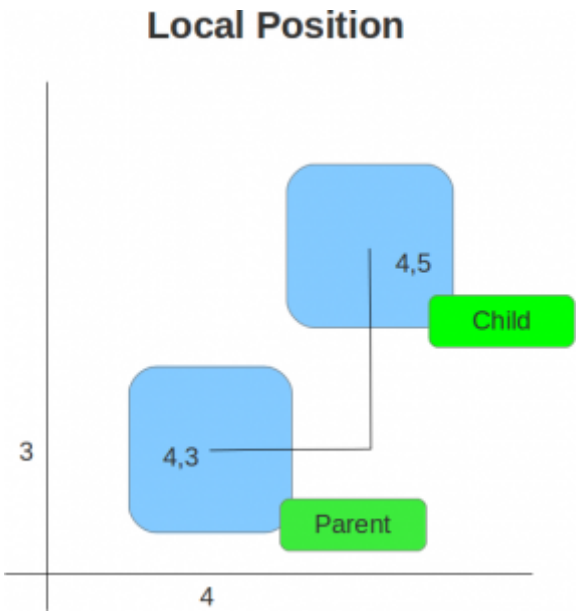


Local space

Los objetos en un escenario 3D están posicionados en relación al world zero. Pero, para hacer las cosas un poco más fáciles usamos local space (o también llamado object space) para definir la posición de un objeto respecto de otro. Esta relación se conoce como parent-child. Local space asume que cada objeto tiene su posición (0,0,0), desde este punto, normalmente el centro del objeto, se expandirá su volumen. La

relación parent-child nos permitirá comparar las posiciones de diferentes objetos entre ellos a través de las relaciones.

Como podemos ver en esta imagen el segundo objeto (child) con posición (4,5) esta situado respecto al objecto (parent) con posición (4,3), esto es igualmente aplicable a escenarios 3D.



Posicionamiento y diseño de assets

Es importante tener en cuenta que si diseñamos assets con alguna herramienta de modelado (por ejemplo blender) tenemos que asegurarnos que la posición en la que creamos el modelo sea la (0,0,0).

Vectores

Los vectores simplemente son lineas dibujada en un escenario 3D con una longitud y dirección. Se pueden mover en el world space sin que estos sufran modificaciones. Son muy útiles en el diseño de juegos 3D, nos permiten:

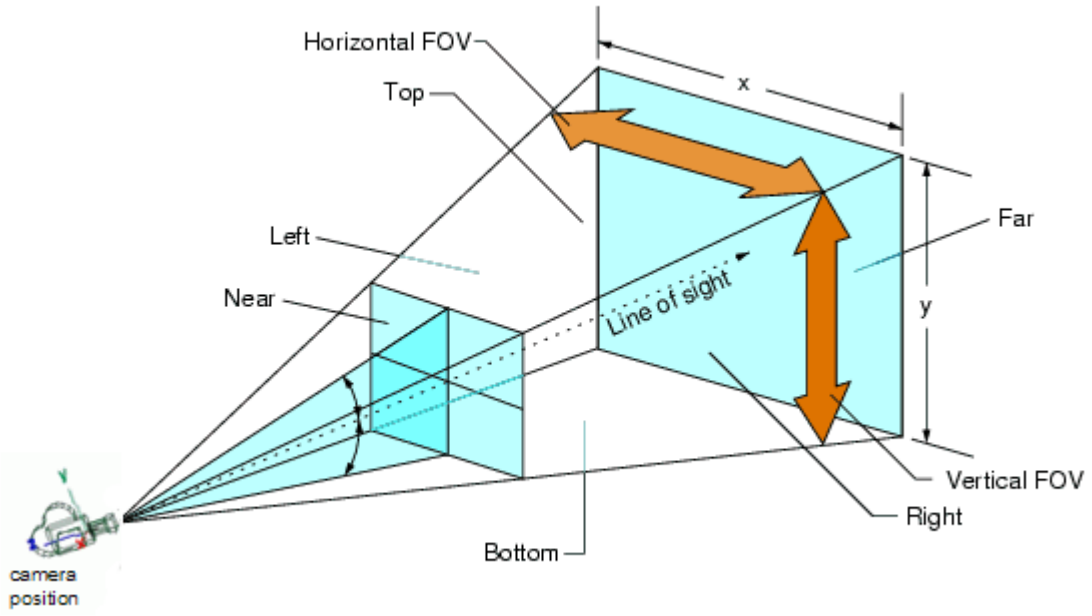
- Calcular distancias entre objetos
- Ángulos de posición entre ellos
- La dirección de estos

Cámaras

Las cámaras son esenciales en mundos 3D, ya que actúan como el punto de visión. Pueden ser posicionadas en cualquier punto del mundo, animadas, enlazadas a un objeto u objetos que forman parte de un escenario. Varias cámaras pueden existir en una escena particular, pero se asume que una siempre será la «main camera» (cámara principal) que renderizará lo que el usuario puede visualizar.

Projection mode (3D vs 2D)

El Projection mode (modo de proyección) de estados cámara se representa en 3D (perspectiva) o 2D (ortográfica). Por lo general, las cámaras están en el modo perspectiva de proyección, y como tal, tiene un campo de visión en forma de pirámide (FOV).



Se puede utilizar en una cámara principal rectangular para juegos 2D o como cámara secundaria en juegos 3D que sirve para hacer Heads Up Display (HUD) elementos como un mapa o una barra de energía. En los motores de juego los efectos como iluminación, desenfoques de movimiento y otros se aplican a la cámara para ayudar a crear una experiencia de juego más realista al ojo humano. Los juegos 3D más modernos utilizan

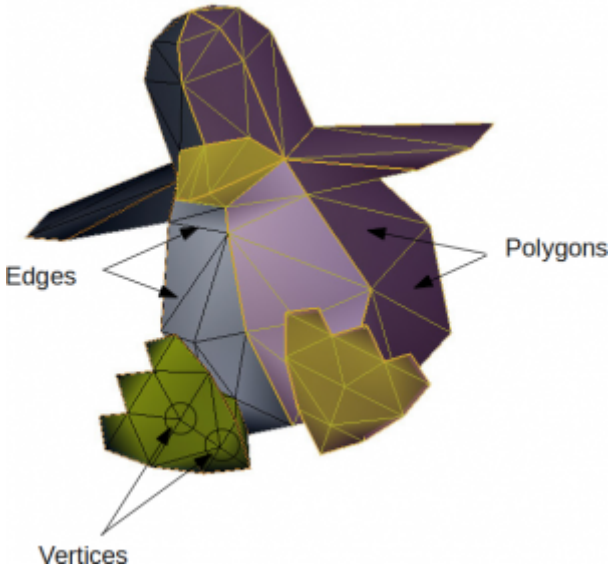
varias cámaras para mostrar las partes del mundo.

Plygons, edges, vertices and meshes

Las formas 3D estan compuestas de pequeñas formas 2D:

- Polygons: Formas 2D (por ejemplo en Unity son triángulos)
- Edges: Los triángulos están formados por edges (bordes) conectados entre si.
- Vertices: Punto en el que convergen los edges.
- Meshes: son formas complejas creadas a partir de la interconexión de muchos plygons. Por ejemplo el tux que tenemos bajo estas lineas.

Tux Mesh



Al conocer estos puntos y ploygons, los motores de juego son capaces de hacer cálculos sobre los puntos de impacto, conocidos como colisiones. Se utiliza la detección de colisiones complejas con Mesh Colliders (colisionadores de malla), por ejemplo en juegos de shooters para detectar la ubicación exacta en la que una bala impacta sobre un objeto. Los meshes pueden tener otros usos, por ejemplo se pueden utilizar para especificar una forma del objeto menos detallado, pero más o menos la misma forma. Esto puede ayudar a mejorar el rendimiento del motor de física evitando que este

compruebe un mesh muy detallado.

Materials, textures and shader

- Materials: Son un concepto común en los entornos 3D, ya que establecen la apariencia visual de un modelo 3D (mesh). Desde colores básicos a reflejos en imágenes que representan superficies, los materiales lo manejan todo.
- Textura: una o más imágenes que se pueden aplicar al material para mejorar la apariencia de este.
- Shaders: un material trabaja con un shader, que no es más que un script encargado de estilizar el renderizado.

Cuando se crean texturas con programas de diseño como Photoshop o GIMP, hay que tener en cuenta la resolución. Grandes texturas aseguran un mayor detalle pero es más costoso renderizarlas.

Rigidbody physics

Cuando se trabaja con motores de juego, los motores de física proporcionan la posibilidad de que los objetos en los juegos simulen respuestas similares a las del mundo real.
En los motores de juego, por defecto un objeto no debe verse afectado por la física, porque esto requiere una gran cantidad de potencia de procesamiento, y porque simplemente no hay necesidad de hacerlo. Por ejemplo, en un juego de de carreras en 3D tiene sentido
que los coches esten bajo la influencia del motor de física, pero no la pista o alrededores (árboles, paredes…) que se mantengan estáticos durante la partida.
Los motores de física para juegos utilizan el sistema Rigidbody dynamics para crear movimientos realistas. Esto significa que en lugar de tener objetos estáticos en un entorno 3D, estos pueden tener propiedades tales como la masa, gravedad, velocidad y fricción. A medida que la potencia del

hardware y software incrementa, Rigidbody physics es cada vez más usado, ya que ofrece la posibilidad simular entornos 3D más variados y realistas.

Detección de colisiones

La detección de colisiones es la forma en la que analizamos nuestro mundo 3D. Al aplicar un Collider component a un objeto, estamos poniendo una red invisible a su alrededor. Esta red, por lo general, imita su forma y es la encargada de informar de cualquier colisión con otros Colliders, haciendo que el motor de juego de responder en consecuencia.

Observaciones
Bien, este ha sido el primer post. Se han introducido los conocimientos básicos para poder trabajar en un entrono 3D. En el siguiente post haremos una aproximación al motor Unity.

Ruben.

Tutorial Threejs – Parte I , Introducción

Hola a todos.

Vamos a comenzar una serie de posts dedicados a Threejs , que espero que sea de vuestro agrado.

¿Que es Threejs?

Threejs es motor 3D ligero para javascript .

Funciona muy bien bajo webGL , aprovechando la aceleración de nuestra tarjeta gráfica , dando unos resultados espectaculares.

Podemos encontrar unos ejemplos de lo que se puede hacer en la página de ejemplos oficial de Threejs.

Ésta serie de artículos pretende explicar el funcionamiento de Threejs , que hacer , como empezar , como crear escenas , importar objetos , etc , etc.

Empezaremos explicando un poco que es necesario para empezar:

Un navegador con soporte webGL , actualmente los mejores resultados los da Google Chrome , aunque se puede utilizar Mozilla Firefox , aunque es más pesado y por lo tanto un poco más lento , no olvidemos que Chrome utiliza el motor webkit que utilizan también muchos smartphones , mientras que Mozilla utiliza el motor Gecko , basado en java .

Conocimientos de programación , javascript , y html5 .

Nociones de 3D , para no perderse , o mucha paciencia para absorver conceptos nuevos.

Descargar la librería Threejs con los ejemplos y herramientas que iremos utilizando y explicando.

Que podemos hacer !

Con Threejs podemos hacer casi cualquier cosa que podemos hacer con un motor 3D , vamos a explicarlo para que sirva también como tutorial de iniciación al 3D.

Básicamente consiste en crear una escena , con objetos dentro que visualizaremos en la pantalla , en nuestro caso en un canvas de html5 , a través de cámaras e iluminados por luces .

La escena

Es la representación del mundo virtual tridimensional , donde

se irán colocando los demás elementos , es algo así como un contenedor del espacio en 3D , que empieza estando vacío .

Este espacio está vacío , no tiene nada , iremos colocando por turno los objetos que deseamos visualizar , basándonos en que la escena se pude medir , con eje de coordenadas virtual a partir del cual iremos posicionando los objetos en las 3 dimensiones básicas .

Gracias a esta referencia dentro de la escena podemos rotar los objetos , desplazarlos , etc , etc.

La escena tiene un fondo que se puede personalizar , que sería un poco así lo que sería el horizonte , se pude dejar un color plano , o una imagen , depende de lo que se quiera hacer.

La cámara

La cámara nos permite visualizar la escena desde un punto de vista concreto .
Se comporta como un objeto más , se puede rotar , desplazar , pero además tiene algunas propiedades más , propias de una cámara .

Podemos tener varias cámaras y utilizar una u otra según convenga.

Las luces

Las luces son objetos que irradian luz y nos permiten visualizar los objetos gracias a la reflexión de la luz en ellos.

Hay varios tipos de luz , de punto (Point), solar (sun), foco (Spot) , en otras aplicaciones podemos encontrar más tipos de luz.

La intensidad y el color de las luces son configurables y nos permiten una multitud de posibilidades y pueden provocar sombras sobre los objetos de la escena.

Cada objeto reacciona diferente a la luz según sus valores para la luz de difusión (color reflejado) , especular (color refractado) y emisión (color emitido por ejemplo una bombilla) , variando según el color e intensidad.

Para una iluminación correcta es necesario el uso de normales , que son el cálculo de la perpendicular de cada cara visible del objeto .

Para empezar , la mayoría de aplicaciones comienzan con un escena que contiene una cámara y una luz , en muchos casos incluye un objeto en modo de ejemplo que suele ser un cubo o una esfera .

Los objetos

Los objetos son una representación de arrays de vértices que forman líneas y caras.
Jeje que fácil no ,

Los vértices son puntos que ocupan unas coordenadas .
La unión de 2 vértices forma líneas.
La unión de varias líneas forman planos que se suelen llamar caras.

Las caras se suelen formar por 3 o 4 líneas que le dan forma , y obviamente una cara/plano de 4 líneas se puede representar como 2 caras/planos de 3 líneas , así que lo más normal es encontrarnos con modelos de objetos formados por arrays de vértices que representan estos triángulos .

En la mayoría de aplicaciones para modelar se especifica al exportar si se quieren crear triángulos (3 líneas) en aquellas caras que son romboides (4 líneas) , para su uso posterior.

Este array de vértices que comentamos no es más que la forma del objeto , hay que posicionar el objeto en la escena .

Los objetos están hechos con uno o varios materiales a los que hay que asignarle valores según tenga que reaccionar a la luz

un color de difusión , otro de especular y el color de emisión si emite luz.

Cada material puede tener una textura , que puede ser una imagen , hasta en ocasiones un vídeo.

Las texturas dan el toque de calidad a los objetos , siendo unos de los temas más importantes del desarrollo en 3d ,por ejemplo podemos aplicar transparencias gracias a las texturas , o de un plano básico aplicarle una textura para que parezca una casa.

Los objetos pueden ser de formas básicas como un cubo o una esfera , o puede ser una figura compleja , pero en el caso de ser una figura compleja mejor crearla con un editor externo (no con código me refiero) , e importarlo como un modelo.

Los modelos

Los modelos 3D son objetos pre-diseñados que se utilizan para incorporar en la escena. Pueden incluir sus propias texturas , y estar formados por uno o varios objetos , a su vez con sus propias texturas.

Los modelos se comportan como una unidad , y se ubican a partir de un punto que es su centro de coordenadas , que se suele ubicar abajo , pero puede variar según el creador.

Podemos diseñar objetos en aplicaciones como Blender , con la comodidad que ello aporta , y exportarlos para el uso en nuestras aplicaciones , Threejs nos aporta herramientas para convertir objetos en formato obj en objetos en formato compatible para Threejs en json.

Los modelos pueden tener animaciones programadas preparadas para nuestro uso , por ejemplo ficheros md2 con acciones como saltar , correr , cargar , etc .

Threejs nos aporta una herramienta para crear modelos en formato Threejs , muy útil y práctica.

Resumiendo un poco , ya sabemos que tenemos que crear una escena , añadirle una cámara , una luz , y los objetos y modelos que queramos.

Bueno hasta aquí la introducción , en el próximo artículo pasaremos a la acción creando la primera escena .

Saludos ,
Scuraki

Tutorial Arduino parte 4

Hola a todos .

Este es el cuarto tutorial sobre Arduino , en este tutorial vamos a continuar el tutorial 3 , creando una aplicación 3d para Android , que encienda la bombilla de forma remota .

Creamos un proyecto Android nuevo en eclipse con Android SDK.

Continuando como teníamos en el tutorial 3 , en la parte del servidor php , el fichero que recibía una variable GET y encendía o apagaba la bombilla si la variable es high o low.

En este primer código podemos ver como desde el proyecto Android podemos realizar la conexión con el servidor PHP para que envíe la señal a la placa Arduino.

El fichero PHP ha de estar en modo servicio , con las lineas //die… descomentadas , para que nos devuelva una respuesta.

```
private void lighting()
{
ConnectivityManager cm = (ConnectivityManager) getSystemService(Context.CONNECTIVITY_SERVICE);
```

```
if (cm != null) {
boolean connected=false;

NetworkInfo[] info = cm.getAllNetworkInfo();
if (info != null) {
for (int i = 0; i < info.length; i++) { if (info[i].getState()
== NetworkInfo.State.CONNECTED) { connected = true; } } }
if(connected) { String
destiny="http://192.168.0.131/arduino/service.php?check=on";
String missatxe="Signal CHECK sended. Click again to switch
ON"; cambiarTextoBoton("Switch ON"); switch(estado) { case 0:
destiny ="http://192.168.0.131/arduino/service.php?hight=on";
missatxe="Signal ON sended. Click again to switch OFF";
cambiarTextoBoton("Switch OFF"); break; case 1: destiny
="http://192.168.0.131/arduino/service.php?low=on";
missatxe="Signal OFF sended. Click again to switch ON";
cambiarTextoBoton("Switch ON"); break; } escribir(missatxe);
InputStream is = null; //initialize String result = ""; try{
HttpClient httpclient = new DefaultHttpClient(); HttpPost
httppost = new HttpPost(destiny); HttpResponse response =
httpclient.execute(httppost); HttpEntity entity =
response.getEntity(); is = entity.getContent(); //convert
response to string try{ BufferedReader reader = new
BufferedReader(new InputStreamReader(is,"UTF-8"),8);
StringBuilder sb = new StringBuilder(); String line = null;
while ((line = reader.readLine()) != null) { sb.append(line +
"\n"); } is.close(); result=sb.toString(); ArrayList
message=this.parse( result);
String status=message.get(0).toString();
String signal=message.get(1).toString();
String mensage=message.get(2).toString();
if(signal.compareTo("1")==0)
{
checkOK=1;
estado=0;
}
else
```

```
{
if(signal.compareTo("2")==0)
{
checkOK=2;
estado=1;
}
else
{
if(signal.compareTo("-1")==0)
{
checkOK=-1;
estado=-1;
}
}

}
}catch(Exception e){
Log.e("log_tag", "Error converting result "+e.toString());
escribir("Error converting result "+e.toString());
}
}catch(Exception e){
Log.e("log_tag", "Error in http connection "+e.toString());
escribir("Error in http connection "+e.toString());
}
}
}

}
```

Bien , ya podemos comunicarnos con el servidor , pero bueno un poco rancia la aplicación , no?

¿Falta un botón , un poco de color o algo no?

Puesto a investigar un poco pués encontré una librería que permitía crear aplicaciones OPENGL de manera un poco más sencilla de lo normal , con un lenguage más para un principiante como yo que ir directamente a OPENGL de Android ,

que es un poco más rudo para alguien que no lo usa muy a menudo.

Esa librería es Min3D.

Min3D nos permite crear escenas 3D e importar objetos 3DS o OBJ , así como texturas y objetos básicos como esferas y cubos.

Tras descargar min3d de su repositorio , incluimos la carpeta al proyecto y podemos ver y probar los ejemplos.

Partiendo del ejemplo ExampleLoadObjFileMultiple.java , donde podemos ver como cargar un objeto desde un fichero .OBJ .

En este fichero podemos ver que hereda de la clase extends RendererActivity , y lo aplicaremos a la clase que estemos utilizando .

El método initScene() es el encargado de crear la escena , en este método es donde colocaremos todos los objetos de la escena , cargaremos todos los ficheros necesarios , .OBJ , .PNG , y crearemos las esferas y rectángulos que necesitemos.

Este método es llamado al iniciar la aplicación y cada vez que el dispositivo entra en modo PAUSE , o se bloquea la pantalla.

Como en toda aplicación 3D , además de una escena necesitamos una cámara y luces para iluminar nuestros objetos.

En este método asignaremos un color de fondo , o una textura , situaremos las luces , los objetos y situaremos y enfocaremos la cámara , para obtener la perspectiva adecuada.

El método updateScene() se ejecuta en cada frame , sería el equivalente al típico evento draw() de repintado de la pantalla en aplicaciones 2D , osea que se ejecuta cada vez que se pinta la pantalla.

Éste método es el encargado de las instrucciones de las

animaciones , por ejemplo si queremos rotar un objeto , en este método calculamos el valor nuevo de la rotación y se le asigna.

Para cargar los ficheros .OBJ en la aplicación Android es necesario modificar los nombres de los archivos.

Los ficheros .OBJ suelen ir acompañados de un fichero con el mismo nombre con extensión .mtl con la definición de los materiales de los grupos de los objetos que hayan definidos en el fichero .OBJ , además de las imágenes de las texturas.

Osea que el fichero .OBJ tiene objetos , valores de vértices y objetos y el fichero .mtl los materiales.

Las imágenes de las texturas las colocaremos en la carpeta res/drawable… .

Los ficheros .OBJ y .MTL los renombramos sustituyendo el punto por un «_» osea un guión bajo , para evitar los conocidos conflictos de ficheros con el mismo nombre y diferente extensión que tiene la programación con Android.

Para este tutorial hemos creado una farola , con una esfera que hace de bombilla , y una esfera con una textura con transparencia .PNG , que hará el efecto de que la farola está encendida.

También se ha creado una especie de bullofa que emite unas esfera con transparencia , emulando el envío de ondas , dando a entender que se está comunicando con el servidor , una esfera nos indicará según la textura que tenga el texto ON de color verde , o el texto OFF de color roja o de color ambar si no hay conexión con el servidor ,a además hay otra bola que nos indicará según su textura , si estamos a más de cierta distancia de la bombilla , si hay cobertura GPS .

Bueno abrimos blender y creamos una lámpara y una bullofa (algo que de el efecto de ser un emisor de ondas) , y las

exportamos en formato wavefront .OBJ .

Si no queremos utilizar blender o otro software para crear objetos 3D , los podemos descargar de alguna página , por ejemplo ésta , que ofrezcan objetos 3D en formato .OBJ .

Podemos ver como asignar el color de fondo:

```
scene.backgroundColor().setAll(0xfff2d533);
```

Asignar una textura a un rectángulo:

```
Bitmap b = Utils.makeBitmapFromResourceId(this, R.drawable.scuraki);
float w = 20f;
float h = w * (float)b.getHeight() / (float)b.getWidth();
Rectangle suelo = new Rectangle(w, h, 1,1, new Color4());
suelo.doubleSidedEnabled(true); // ... so that the back of the plane is visible
suelo.normalsEnabled(false);
scene.addChild(suelo);

Shared.textureManager().addTextureId(b, "scuraki", false);
suelo.textures().addById("scuraki");
```

Asignar una textura a una esfera:

```
b = Utils.makeBitmapFromResourceId(R.drawable.bolaroja);
Shared.textureManager().addTextureId(b, "bolaroja", false);
bolarojaTexture = new TextureVo("bolaroja");
esfera.textures().addReplace(bolarojaTexture);
```

Iluminar la escena:

```
luzobj = new Light();
luzobj.ambient.setAll(new Color4 (128,128,128,128));
luzobj.diffuse.setAll(new Color4 (64,64,164, 128));
luzobj.emissive.setAll(new Color4 (0,0,0,255));
luzobj.specular.setAll(new Color4 (0,0,0,255));
luzobj.type(LightType.POSITIONAL);
scene.lights().add(luzobj);
luzobj.position.setAll(0.65f, -0.85f, 3.5f);
```

Añadir un objeto .OBJ a la escena:

```
parser2 = Parser.createParser(Parser.Type.OBJ, getResources(),
"adictosalainformatica.min3DAdictos:raw/fanal_obj", true);

parser2.parse();

fanal = parser2.getParsedObject();
fanal.scale().y = 0.25f;
fanal.scale().z = 0.25f;
fanal.scale().x = 0.25f;
fanal.shadeModel(ShadeModel.SMOOTH);
fanal.vertexColorsEnabled(true);
fanal.normalsEnabled(true);
fanal.colorMaterialEnabled(false);

scene.addChild(fanal);
fanal.position().x=0.0f;
fanal.position().y=1.6f;
fanal.position().z=-5f;
fanal.rotation().x=25f;
```

Controlar la rotación de los objetos en el método updateScene:

```
bombilla.rotation().y=yrot;
bombilla.rotation().x=xrot;
receptor.rotation().y=yrot;
receptor.rotation().x=xrot;

xrot += xspeed;
yrot += yspeed;
_count++;
```

Para dar un toque de color la bullofa cambia el valor de escalado cada cierto tiempo , y se crean unas esferas con transparencia , que viajan desde la bullofa (que está cerca de la cámara) hasta la lámpara (que está al fondo de la escena), emulando el envio de ondas.

Un array de esferas ameniza la pantalla , desplazandose .

La clase Burbuja es la encargada de controlar estas esferas:

```
package adictosalainformatica.min3DAdictos;

import android.graphics.Bitmap;
import min3d.Shared;
import min3d.Utils;
import min3d.core.Scene;
import min3d.objectPrimitives.Sphere;
import min3d.vos.Color4;
import min3d.vos.TextureVo;
public class Burbuja {
private float velocidad_x=.000f;
private float velocidad_y=0.02f;
private float velocidad_z=0.04f;
private float posicion_x=0.45f;
private float posicion_y=-0.25f;
private float posicion_z=0.3f;
private float variacion_x=0.45f;
private float variacion_y=-0.25f;
private float variacion_z=0.6f;
private float limite_x=4.01f;
private float limite_y=4.01f;
private float limite_z=4.8f;
private Color4 color=null;
private Sphere esfera =null;
long _index;
private float contador=0;
public Burbuja(long index)
{
this._index=index;
}
public void make(Scene scene )
{
if(esfera!=null)
{
esfera.clear();
}
```

```
esfera=null;
TextureVo textura=null;
esfera = new Sphere(1.5f, 20,20);
esfera.scale().x = esfera.scale().y = esfera.scale().z = .1f;
esfera.position().setAll(posicion_x, posicion_y, posicion_z);
Bitmap                          b                          =
Utils.makeBitmapFromResourceId(R.drawable.tscuraki);
Shared.textureManager().addTextureId(b, "burbuja0" + _index,
false);
b.recycle();
textura = new TextureVo("burbuja0" + _index);
esfera.textures().addReplace(textura);
esfera.colorMaterialEnabled(false);
esfera.vertexColorsEnabled(false);
esfera.lightingEnabled();
scene.addChild(esfera);

//Log.v(Min3d.TAG, "ReCrea Burbuja=" + _index);

}

public int mover()
{
int moviendo=1;
posicion_x=variacion_x + ( contador * velocidad_x);
posicion_y=variacion_y + ( contador * velocidad_y);
posicion_z=variacion_z - ( contador * velocidad_z);
esfera.position().setAll(posicion_x, posicion_y, posicion_z);
if((posicion_x>limite_x)||(posicion_y>limite_y)||(posicion_z *
-1 > limite_z ))
{
esfera.clear();

moviendo=0;
}
//Log.v(Min3d.TAG, "Mueve Burbuja=" + _index + " posiciónx="+
posicion_x + " posicióny=" + posicion_y + " posiciónz=" +
posicion_z);
```

```
esfera.rotation().x=contador * -1.3f ;
esfera.rotation().y=contador * -1.3f;
esfera.rotation().z=contador * -1.3f;
esfera.scale().x = esfera.scale().y = esfera.scale().z = .1f +
(contador * 0.001f);
contador++;
return moviendo;

}

public float getLimite_x() {
return limite_x;
}
public void setLimite_x(float limite_x) {
this.limite_x = limite_x;
}
public float getLimite_y() {
return limite_y;
}
public void setLimite_y(float limite_y) {
this.limite_y = limite_y;
}
public float getLimite_z() {
return limite_z;
}
public void setLimite_z(float limite_z) {
this.limite_z = limite_z;
}
public Color4 getColor() {
return color;
}
public void setColor(Color4 color) {
this.color = color;
}
public float getVelocidad_x() {
return velocidad_x;
}
```

```
public void setVelocidad_x(float velocidad_x) {
this.velocidad_x = velocidad_x;
}
public float getVelocidad_y() {
return velocidad_y;
}
public void setVelocidad_y(float velocidad_y) {
this.velocidad_y = velocidad_y;
}
public float getVelocidad_z() {
return velocidad_z;
}
public void setVelocidad_z(float velocidad_z) {
this.velocidad_z = velocidad_z;
}
public float getPosicion_x() {
return posicion_x;
}
public void setPosicion_x(float posicion_x) {
this.posicion_x = posicion_x;
}
public float getPosicion_y() {
return posicion_y;
}
public void setPosicion_y(float posicion_y) {
this.posicion_y = posicion_y;
}
public float getPosicion_z() {
return posicion_z;
}
public void setPosicion_z(float posicion_z) {
this.posicion_z = posicion_z;
}
public float getVariacion_x() {
return variacion_x;
}
public void setVariacion_x(float variacion_x) {
```

```
this.variacion_x = variacion_x;
}
public float getVariacion_y() {
return variacion_y;
}
public void setVariacion_y(float variacion_y) {
this.variacion_y = variacion_y;
}
public float getVariacion_z() {
return variacion_z;
}
public void setVariacion_z(float variacion_z) {
this.variacion_z = variacion_z;
}
public Sphere getEsfera() {
return esfera;
}
public void setEsfera(Sphere esfera) {
this.esfera = esfera;
}
public long getIndex() {
return _index;
}
public void setIndex(long _index) {
this._index = _index;
}

}
```

Finalmente controlamos la distancia a la bombilla gracias al sensor GPS , activándolo si es necesario.

Podemos decidir activar la bombilla a cierta distancia , útil para luces de garages , por ejemplo que se encienda cuando falta 2 kilómetros para llegar con el coche.

En nuestro caso tenemos la esfera que nos indica la distancia , modificamos la textura para que nos muestre FAR si está

lejos y NEAR si está cerca.

```
private void loadJipiEs()
{
// Acquire a reference to the system Location Manager
this.locationManager = (LocationManager)
this.getSystemService(Context.LOCATION_SERVICE);

if
(!this.locationManager.isProviderEnabled(LocationManager.GPS_P
ROVIDER))
{
createGpsDisabledAlert();
}
else
{
// List all providers:
List providers = this.locationManager.getAllProviders();

Criteria criteria = new Criteria();
criteria.setAccuracy(Criteria.ACCURACY_FINE);

this.bestProvider =
this.locationManager.getBestProvider(criteria, false);

Location mylocation =
this.locationManager.getLastKnownLocation(this.bestProvider);

if(mylocation!=null)
{
this.getLocation(mylocation);
}
else
{
//this.afegir("Posición inicial vacía.");
}
}
```

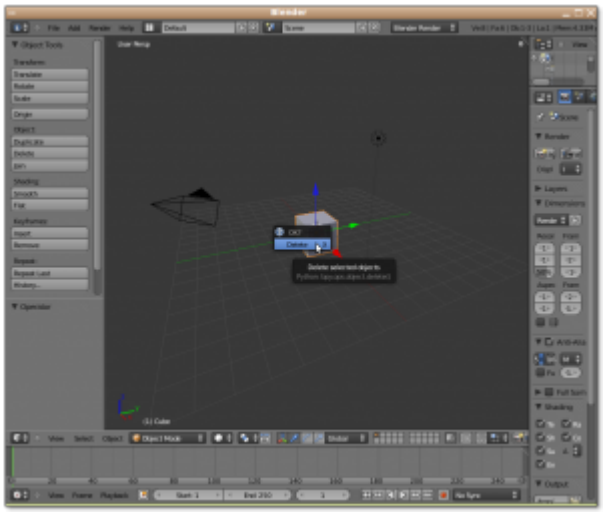
```
}
private void getLocation(Location location)
{
if(location!=null)
{

boolean hasAltitude=false;
boolean hasAccuracy=false;
boolean hasBearing=false;
lat=location.getLatitude();
lon=location.getLongitude();
alt=location.getAltitude();

hasAltitude=location.hasAltitude();
hasAccuracy=location.hasAccuracy();
hasBearing=location.hasBearing();

distance=location.distanceTo(coords);
if(distance<5000) { distanceState="near"; } else {
distanceState="far"; } } else { distanceState="Unknown"; }
//Toast.makeText(getApplicationContext(), "Distance is : " +
distance, Toast.LENGTH_LONG).show(); /*runOnUiThread(new
Runnable() { public void run() {
Toast.makeText(getApplicationContext(), "Distance is : " +
distance, Toast.LENGTH_LONG).show();
missatger.setText("Distance is : " + distance); } });*/
escribir("Distance is : " + distance); } private void
createGpsDisabledAlert(){ AlertDialog.Builder builder = new
AlertDialog.Builder(this); builder.setMessage("Your GPS is
disabled! Would you like to enable it?") .setCancelable(false)
.setPositiveButton("Enable GPS", new
DialogInterface.OnClickListener(){ public void
onClick(DialogInterface dialog, int id){ showGpsOptions(); }
}); builder.setNegativeButton("Do nothing", new
DialogInterface.OnClickListener(){ public void
onClick(DialogInterface dialog, int id){ dialog.cancel(); }
}); AlertDialog alert = builder.create(); alert.show(); }
private void showGpsOptions(){ Intent gpsOptionsIntent = new
```

Intent(android.provider.Settings.ACTION_LOCATION_SOURCE_SETTINGS); startActivity(gpsOptionsIntent); } }

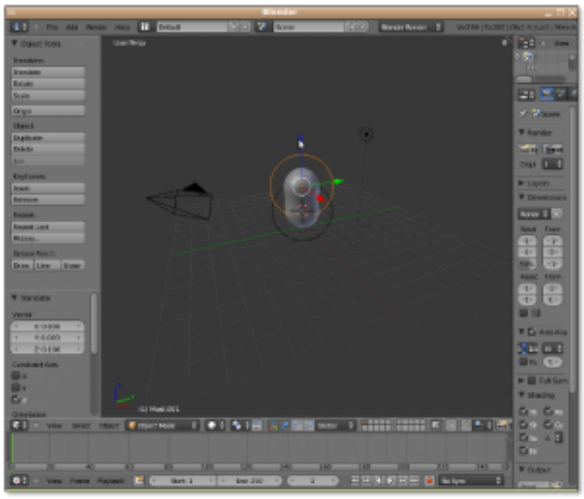
En nuestro caso la luz la hemos activado a través de un botón externo a la escena , esta señal modifica la textura de la esfera que nos indica el estado de la bombilla , según sea la respuesta del servidor , ON , OFF o sin conexión , si la bombila está ON la esfera que emula la bombilla encendida , que está en la lámpara , se hace visible rodeando una esfera más pequeña que hace de núcleo de la bombilla.

Pelo en blender Primer tutorial

Primer tutorial de blender Antes de nada Blender es un programa de modelado , animación , etc en 3D. Blender es **gratuito** , **opensource** . Para descargar Blender , visitad la página de descargas de Blender. En este tutorial se utilizará la versión **2.5** Beta para Linux. Empezaremos con una introducción a las teclas y al entorno. Al iniciar Blender nos sale por defecto un cubo , una luz y una cámara. El objeto seleccionado es el cubo , que está iluminado. Si pulsamos las teclas Ctrl + Alt y movemos la rueda del ratón veremos que la vista gira alrededor del cubo . Si pulsamos Shift y la rueda del ratón se mueve la vista en sentido vertical . Si pulsamos Ctrl y la rueda del ratón se mueve la vista en sentido horizontal. Si pulsamos con botón derecho en un objeto , se



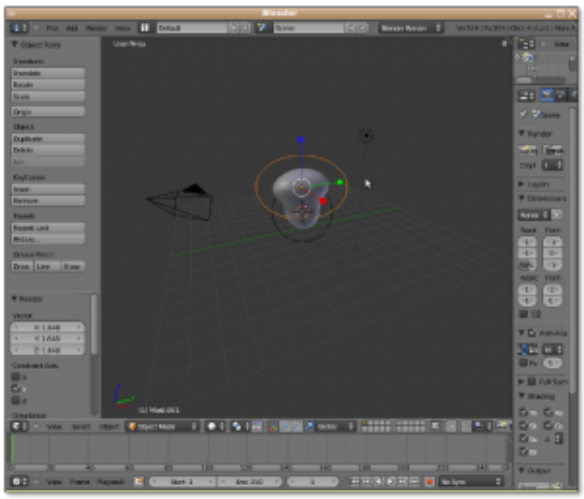
selecciona el objeto pulsado. En este enlace: http://blender.gulo.org/ encontrareis tutoriales para iniciarse a Blender , versión 2.4 y en este la versión 2.5. **Bueno vamos con el tutorial**. -Creamos un nuevo documento , en File/New. -Eliminamos el cubo pulsando la tecla Supr. -Añadimos un Metaball , en Add/Metaball/Metaball , este será el primero de varios. -Añadimos otro Metaball , este no se vé , desplazamos el objeto seleccionado pulsando las flechas de movimiento , hacia arriba , y observamos que los 2 están unidos por una misteriosa fuerza , que si lo alejamos demasiado se separan. Éste es el motivo de utilizar Metaball , en vez de esferas normales.



Uso de Metaball

-Hacemos más grande (escalar) el Metaball seleccionado , dando forma a nuestro objeto. -Así añadiremos los Metaball necesarios para tener la figura que deseamos , en este caso

una cabeza , uno para cada oreja , los ojos , la boca.

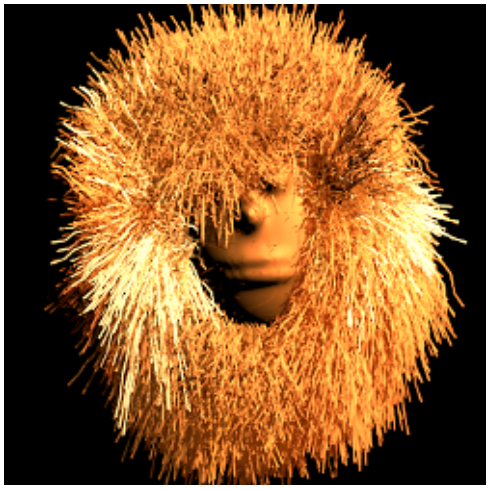


Escalando

-Una vez tengamos una figura (básica) que nos convenza , transformaremos el metaball en un Mesh (Masa), para convertirlo en un objeto con propiedades . Pulsamos Alt + c , y seleccionamos la Opción Mesh from Curve/Meta/… -Ya tenemos un objeto Mesh , con todas sus propiedades , ya podemos asignarle un Material , esculpir la figura , editar los vértices , etc . Ahora vamos a crear 2 materiales , uno será el de la cara y otro será el del pelo. El Material puede ser transparente , tiene 2 colores difuse (color) y especular (color al reflejar), puede reflejar la luz (Mirror) ,etc. -Para crear Pelo vamos a la opción de Partículas , añadimos una , de tipo Hair. Esto nos genera pelo en todas las partes del objeto , para evitar esto , y que solo salga por la parte que nos interesa , osea arriba y detras de la cara, tendremos que crear un grupo de vértices .Vamos a edit mode (o pulsamos Tab) , pulsamos la tecla a, para seleccionar o deseleccionar todos , y pulsando la tecla b nos permite seleccionar creando un rectángulo. Una vez seleccionado pulsamos Ctrl + g para crear un nuevo grupo. En la pestaña de partículas vamos a la opción vertexgroups y en el primero , seleccionamos el grupo creado antes (si es el primero se llama group , sino group0001 , etc).

Uso de Particle Mode

-Ya tenemos el pelo en la zona que queremos.Ahora le asignamos el material 2 ,que es el segundo de la lista de materiales , para que no sea del mismo color que la cara .



Uso de las propiedades
de partículas

-Una vez tenemos el pelo del color que tengamos puede que nos interese peinar el pelo, si peinar no me he equivocado , entramo en el modo Particle Mode , a la izquierda en el menú de Brush (brocha) , por defecto arriba está seleccionado none , seleccionamos comb para poder modificar y vamos peinando nuestra figura. -Más o menos tenemos nuestra figura creada , ahora falta renderizar , ¿eso que es? , pués es obtener imágenes de cada paso (frame) , a partir de lo que ve la cámara. En nuestro caso tenemos solo 1 imagen , ya explicaré

como crear animaciones ,sencillas. Si pulsamos Ctrl + Izq de los cursores de teclado o Ctrl + Dcha , podemos cambiar la perspectiva de blender , hay una perspectiva de animación , otra de juegos , y normalmente hay una de cámara. También podemos ver lo que ve la cámara (recordad que es un objeto y la podemos mover , rotar e incluso la podemos animar) pulsando Numpad0 , o en el menú view/Camera. Si seleccionamos la perspectiva de camera podemos ajustar el objeto que hemos creado al rectángulo que ve la cámara , moviendo con las flechas de desplazamiento . Podemos ver el resultado en el menú Render/Render Image , o pulsando F12.

Pelo coloreado y peinado

-Por defecto tenemos una luz , las luces se pueden copiar , modificar el color de la luz , mover , rotar. Como cualquier otro objeto , podemos animar la luz , etc . Para este ejemplo he utilizado la luz de tipo Sun (Sol) , pero podeis elegir la que querais , en el panel Light , estando la luz seleccionada. Por último , un enlace a como crear pelo artístico con **curvas** y la herramienta de **esculpir** : http://simpatiaporblender.blogspot.com/2010/01/interesante-forma-para-modela-cabellos.html No está de más explicar que se puede combinar el pelo creado con el sistema de partículas con pelo creado con el sistema de curvas ,como explica el tutorial anterior , puede dar resultados muy buenos , por ejemplo para

crear una pulsera. y hasta aquí nuestro primer **tutorial** de **blender** .