# 1 LIBGDX Os graficos

# UNDIDADE 2: Os gráficos

Os gráficos que imos usar no desenvolvemento de xogos 2D van ser texturas asociadas a gráficos jpg ou bmp.

Unha textura é un gráfico bi-dimensional cunha forma rectangular ou cadrada que vai imitar o aspecto dos obxectos. Ven ser a pel exterior dun elemento real. Dito gráfico vai ser empregado para 'cubrir' a superficie dun obxecto virtual.

**Nota:** Como comentamos nun punto anterior o framework tiña soporte para OPEN GL 1.X. Se utilizamos dita versión de open gl, os gráficos tiñan que ter un tamaño potencia de 2 (2,4,8,16,32,64,...) no seu ancho e alto.

**Nota:** Para poder referenciar as texturas dende calquera clase do xogo e debido a que cando saímos da aplicación sen pechar (no caso da versión Android) o contexto gráfico se perde (é dicir, as referencias a todos os recursos gráficos que teñamos xa non valen cando volvemos á aplicación) imos cargar ditas texturas dende unha clase separada na que todas as texturas son de clase (Static).

## 1.1 Sumario



## 1.2 O programa GIMP

Á hora de andar con gráficos, editalos, cambiar a cor de transparencia, converter formatos...podemos utilizar diferentes editores gráficos.

O alumno pode usar calquera que xa teña instalado no seu computador, xa que normalmente en todos imos atopar o que imos facer neste curso.

- O programa que podedes descargar é o **gimp**: http://www.gimp.org.es/

- Neste enlace tedes un videotutorial, pero na rede podedes atopar multitude de artigos sobre o manexo do programa.

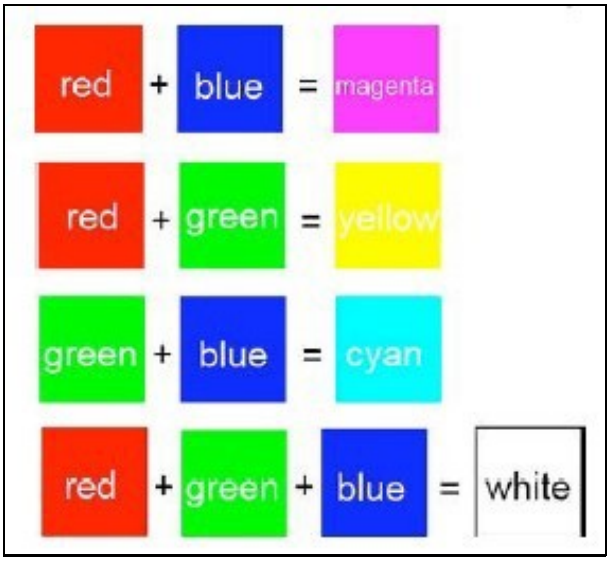
## 1.3 As cores

Cando debuxamos un gráfico, a textura ten información de que cor ten que ter cada pixel que se vai debuxar.

Esta información segue un modelo de cores. Nos imos a centrarnos no modelo RGB (Red Green Blue). Existen outros modelos coma YUV ou CMYK.

Con estas tres cores básicas (vermello, verde e azul) podemos ter calquera cor, mesturándoos na proporción correcta.

Por exemplo:




Información obtida do libro Beginning Android 4 Games

Ditos cores se especifican coa combinación de tres: RED ? GREEN ? BLUE.

Cada cor básico pode expresarse cun número flotante entre 0.0 ? 1.0 (sendo 1 a intensidade máis grande e 0 a máis pequena).

Tamén podemos codificar cada cor cun byte con valor entre 0 a 255.

Dependendo das diferentes opcións podemos ter:

- Codificación RGB de 32 bits: ten 12 bytes por cada pixel (32 bits por cor (ó ser flotante) = 4 bytes x 3 cores=12 bytes). As intensidades poden varias entre 0.0 e 1.0.
- Codificación RGB de 24 bits: ten 3 ou 4 bytes (se ten alfa, visto despois) por cada pixel (8 bits por cor = 1 bytes x 3 cores=3 bytes). As intensidades poden varias entre 0 e 255. A orde dos compoñentes pode ser RGB ou BGR. Se coñece coma RGB888 ou BGR888.
- Codificación RGB de 16 bits: ten 2 bytes por cada pixel. Red e blue teñen unha intensidade que varía entre 0 e 31 (5 bits x 2 = 10 bits) e o verde unha intensidade entre 0 e 63 (6 bits). A orde pode ser RGB ou BGR e se coñece coma RGB565 ou BGR565.

**O normal e que traballemos co formato RGB888.**

## 1.4 Formato das imaxes

Á hora de gardar as imaxes temos multitude de formatos gráficos. Dous dos máis coñecidos son: **JPEG** e **PNG**.

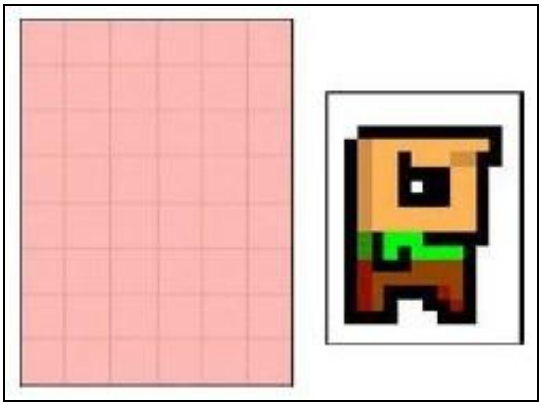
- JPEG é un formato que perde calidade con respecto a imaxe orixinal.
- PNG non perde calidade e é idéntica a imaxe orixinal.

Sempre que utilicemos unha imaxe que non requira transparencias (visto a continuación) nin unha calidade excesiva deberemos elixir jpg xa que ten un gran nivel de compresión.

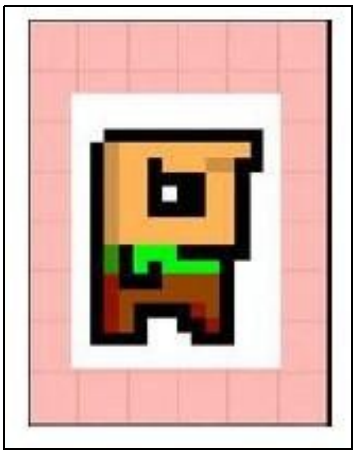
### 1.4.1 A compoñente Alfa: Nivel de transparencia.

A compoñente alfa e un valor que indica o nivel de transparencia dunha imaxe. Normalmente ó superpoñer unha imaxe sobre outra a imaxe superior 'tapa' a inferior. Pero no caso de gráficos como personaxes en movemento, queremos que só se superpoña a imaxe do personaxe.

O problema ven que cando definimos un personaxe este se define cun rectángulo:



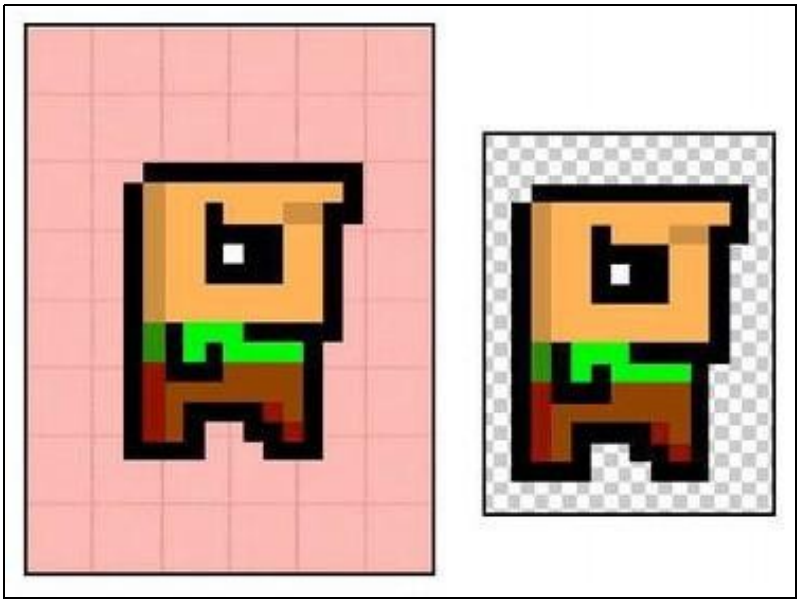
Información obtida do libro 'Beginning Android 4 Games'
O poñer enriba do fondo a imaxe do personaxe pasaría isto:



Imaxes superposta. A cor branca aparece arredor do gráfico.
Grazas o **compoñente alfa** podemos establecer que unha determinada cor sexa transparente (o que se coñece como alfa).

Así, no caso anterior de formato gráfico, podemos falar de RGBA8888 (e as súas variantes). Con isto estamos a dicir que a compoñente alfa está composto por un valor que vai dende 0 (totalmente transparente) a 255 (totalmente opaco).

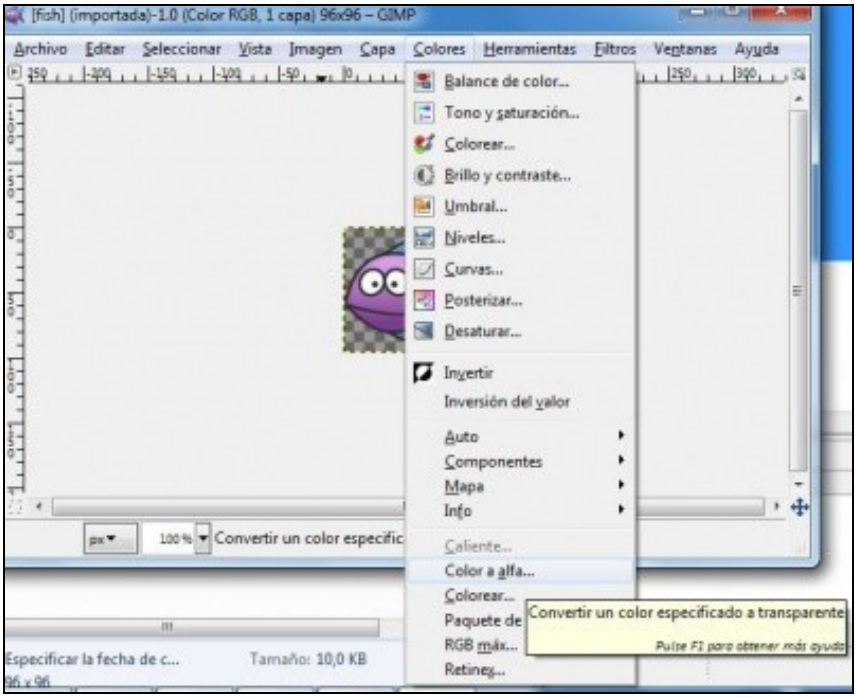
Se facemos que a cor branca sexa transparente teremos isto:



Imaxes superposta. A cor branca ten un nivel de transparencia a 0.
**IMPORTANTE:** Se a imaxe necesita transparencias non podemos gardala en formato JPG xa que non o soporta. Utilizaremos o formato PNG.

Vídeo de como facer un fondo transparente en GIMP: https://www.youtube.com/watch?v=m-7j6bP94fg

Tamén podemos dándolle a cor un nivel de transparencia, como amosa a seguinte imaxe:




Cambiando unha cor a transparente

## 1.5 Carga de gráficos có framework LIBGDX

Para utilizar gráficos nun xogo necesitamos facer:

- Necesitamos unha referencia ó arquivo físico onde se atopa a textura (arquivo png/jpg).
- Crear a textura a partires de dita referencia.
- Usar dito gráfico no xogo.
- Liberar o gráfico da memoria do dispositivo cando xa non se utilice.

### 1.5.1 Referencia ós gráficos

Para ter unha referencia os gráficos imos utilizar a clase FileHandle.

Representa un arquivo, cartafol no sistema de arquivos, Android SD Card ou cartafol Assets de Android.

No enlace anterior existen multitude de métodos para o manexo de arquivos. Neste manual só imos empregalo para referenciar os gráficos que van conformar as texturas.

Dentro da **clase Gdx**:

- Gdx.files.internal: é unha referencia ó **cartafol assets** de Android, onde todos os demais proxectos apuntan.

**Exemplo de código para obter unha referencia a un arquivo gráfico:**

```
FileHandle imageFileHandle = Gdx.files.internal("nomegrafico.jpg");   // Pode ser un png
```

Podemos utilizar rutas relativas ó cartafol assets. Así, si dentro de dito cartafol teño outro chamado graficos poderei facer referencia a un gráfico dentro do cartafol desta forma:

**Exemplo de código para obter unha referencia a un arquivo gráfico nun cartafol en assets:**

```
FileHandle imageFileHandle = Gdx.files.internal("graficos/nomegrafico.jpg");   // Pode ser un png
```

En assets podemos crear unha estrutura de cartafol como o facemos no explorador de arquivos do S.O.

### 1.5.2 Carga das texturas

Para cargar os gráficos necesitamos un obxecto da clase Texture.

Métodos máis importantes:

- **constructor: Texture(FileHandle file)**
  Crea unha instancia da clase Texture.
  - ◊ file: Recibe como parámetro un obxecto da clase FileHandle que ven ser unha referencia ó arquivo físico (bmp/jpg) da textura.
- Método int getHeight()
  Obtén a altura da textura en pixeles.
- Método int getWidth()
  Obtén a anchura da textura en pixeles.

**Exemplo de carga dunha textura:**

```
FileHandle imageFileHandle = Gdx.files.internal("nomegrafico.jpg");   // Pode ser un bmp
Texture textura;
textura = new Texture(imageFileHandle);
```

### 1.5.3 Liberación da memoria das texturas

- ◊ Método dispose()
  Libera a memoria.

**Exemplo de liberación de memoria:**

```
textura.dispose();
```

## 1.6 Localización dos gráficos no noso xogo

### 1.6.1 Localización física

Fisicamente os gráficos van estar gardados no **cartafol assets** do proxecto Android.

Imos crear un cartafol dentro do mesmo de nome 'GRAFICOS' onde gardaremos os gráficos do xogo. Tedes a liberdade de estruturar dito cartafol como queirades, así poderíamos poñer os inimigos noutro cartafol dentro de GRAFICOS (GRAFICOS/INIMIGOS) etc....

Descargade agora esta imaxe e dádelle de nome 'LIBGDX_itin1_alien.png' e gardala no cartafol anterior (assets/GRAFICOS):

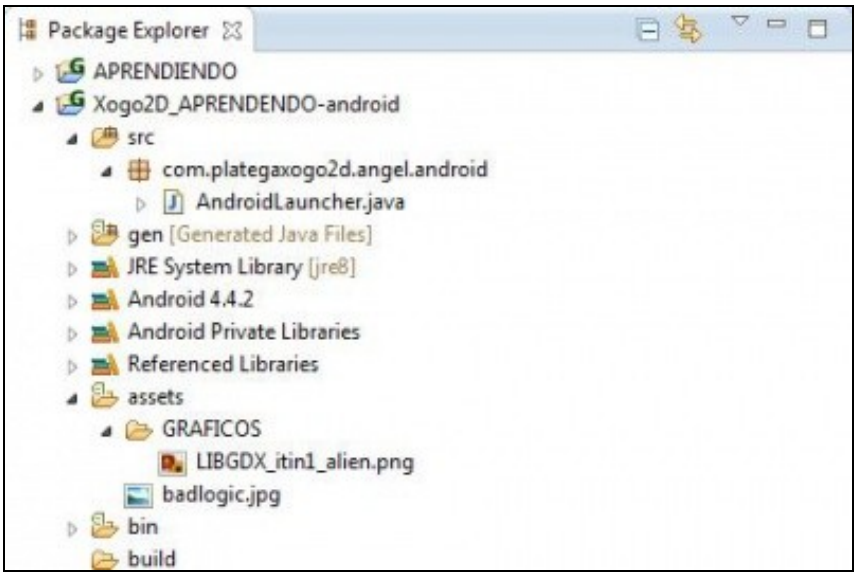
Nota: Gráfico obtido de http://opengameart.org/users/rrodi411

**Nota:** Se descargades a imaxe no cartafol assets doutra das plataformas (por exemplo desktop) é necesario **refrescar** o cartafol assets da plataforma

android para que o teña en conta.

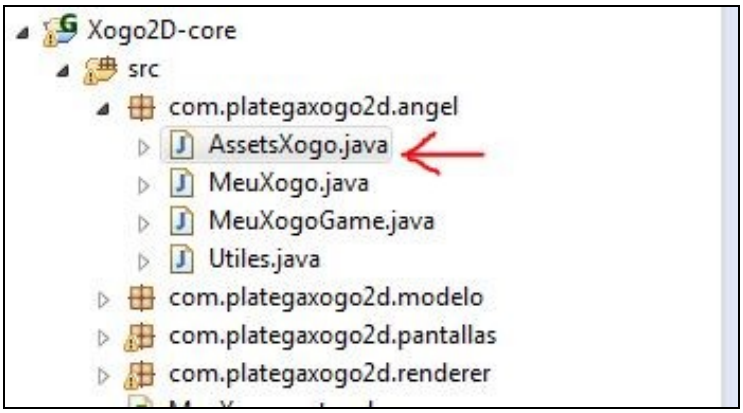
**Nota:** A imaxe ten extensión png por ter transparencias.

Teredes algo parecido a isto:




### 1.6.2 Localización lóxica

Para cargar os gráficos imos utilizar unha clase que imos crear no paquete principal e que chamaremos **AssetsXogo**.




Polo comentado anteriormente da perda do contexto gráfico cando saímos do xogo en Android, imos definir as texturas de clase (static) e imos crear dous métodos de clase: un para cargar as texturas e outro para liberalas.

Dependendo do xogo podemos ter varios arquivos Assets se que por exemplo, podemos cargar os gráficos en función do nivel do xogo, facendo unha división lóxica e creando un arquivo para o protagonista, outro para os inimigos,...

**Exemplo de código para cargar e liberar unha textura**

```
import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.files.FileHandle;
import com.badlogic.gdx.graphics.Texture;

public class AssetsXogo {

public static Texture textureAlien;

/**
        * MÃ©todo encargado de cargar todas as texturas
```

```
        */
public static void cargarTexturas(){
FileHandle imageFileHandle = Gdx.files.internal("GRAFICOS/LIBGDX_itin1_alien.png");
textureAlien = new Texture(imageFileHandle);
}


/**
        * MÃ©todo encargado de liberar todas as texturas
        */
public static void liberarTexturas(){
textureAlien.dispose();
}
}
```

## 1.7 Carga dos gráficos

Agora é necesario chamar a ditos métodos. O lugar pode variar. Podemos liberar e cargar os recursos gráficos a medida que os necesitemos (por exemplo nun xogo con diferentes niveis) ou ben cargar os gráficos ó inicio do xogo e liberalos ó rematar.

Nos imos escoller a segunda opción.

Resumindo:

- No noso exemplo imos chamar a carga dos gráficos ó cargar o xogo e liberaremos os gráficos cando saiamos do xogo.
- Faremos que a referencia a ditos gráficos sexa en forma de variables de clase (static).

**Clase MeuXogoGame Explicación: Chama os métodos para cargar as texturas e liberalas**

```
package com.plategaxogo2d.angel;

import com.badlogic.gdx.Game;
import com.plategaxogo2d.pantallas.PantallaXogo;

public class MeuXogoGame extends Game {

@Override
public void create() {
// TODO Auto-generated method stub

AssetsXogo.cargarTexturas();
setScreen(new PantallaXogo(this));
}

@Override
public void dispose(){
super.dispose();

AssetsXogo.liberarTexturas();
}
}
```

**Exercicio proposto:** Copiar a clase **UD2_2_RendererXogo** (botón dereito, copiar e pegar) e darlle de nome **UD2_3_RendererXogo**. Fai que dita clase visualice o gráfico do alien gardado no cartafol assets/GRAFICOS cun tamaño de 15 unidades. Modificar a clase PantallaXogo para que chame a dita clase.

**Solución:**

**Código da clase UD2_3_RendererXogo:**

Obxectivo: Cargar o gráfico do alien gardado en asssets/GRAFICOS.

```
/**
 *  Explica cales son os métodos que temos para mover a cámara.
 *   Carga un gráfico sinxelo no centro da pantalla
 */

package com.plategaxogo2d.renderer;

import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.InputProcessor;
import com.badlogic.gdx.graphics.GL20;
import com.badlogic.gdx.graphics.OrthographicCamera;
import com.badlogic.gdx.graphics.Texture;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.badlogic.gdx.math.Vector3;
import com.plategaxogo2d.angel.AssetsXogo;
import com.plategaxogo2d.angel.Utiles;
import com.plategaxogo2d.modelo.Mundo;

public class UD2_3_RendererXogo implements InputProcessor {

private SpriteBatch spritebatch;

private OrthographicCamera camara2d;

private Vector3 temporal;

public UD2_3_RendererXogo() {
camara2d = new OrthographicCamera();
spritebatch = new SpriteBatch();

temporal = new Vector3();

Gdx.input.setInputProcessor(this);
}

/**
         * Debuxa todos os elementos graficos da pantalla
         *
         * @param delta
         *            : tempo que pasa entre un frame e o seguinte.
         */
public void render(float delta) {
Gdx.gl.glClearColor(1, 1, 1, 1);
Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

spritebatch.begin();
spritebatch.draw(AssetsXogo.textureAlien, temporal.x,temporal.y, 15, 15);
spritebatch.end();

}

public void resize(int width, int height) {

camara2d.setToOrtho(false, Mundo.TAMANO_MUNDO_ANCHO,
Mundo.TAMANO_MUNDO_ALTO);
camara2d.update();

spritebatch.setProjectionMatrix(camara2d.combined);

}

public void dispose() {

Gdx.input.setInputProcessor(null);
spritebatch.dispose();
}

@Override
public boolean keyDown(int keycode) {
// TODO Auto-generated method stub
return false;
```

```
}

@Override
public boolean keyUp(int keycode) {
// TODO Auto-generated method stub
return false;
}

@Override
public boolean keyTyped(char character) {
// TODO Auto-generated method stub
return false;
}

@Override
public boolean touchDown(int screenX, int screenY, int pointer, int button) {
// TODO Auto-generated method stub

temporal.set(screenX,screenY,0);
camara2d.unproject(temporal);

return false;
}

@Override
public boolean touchUp(int screenX, int screenY, int pointer, int button) {
// TODO Auto-generated method stub
return false;
}

@Override
public boolean touchDragged(int screenX, int screenY, int pointer) {
// TODO Auto-generated method stub
return false;
}

@Override
public boolean mouseMoved(int screenX, int screenY) {
// TODO Auto-generated method stub
return false;
}

@Override
public boolean scrolled(int amount) {
// TODO Auto-generated method stub
return false;
}
}
```

Nota: Borramos a propiedade **grafico** (da clase Texture) xa que agora o cargamos dende AssetsXogo.

- Seguimos có noso proxecto. Modificamos a clase PantallaXogo para que volva a chamar á clase RendererXogo.

---

**TAREFA 2.3 A FACER:** Esta parte está asociada á realización dunha tarefa.

---

## 1.8 Borrando a pantalla

Como comentamos anteriormente nos xogos estamos a borrar e debuxar toda a pantalla continuamente no método render.

Para facelo utilizamos estas dúas ordes de OPEN GL:

```
Gdx.gl.glClearColor(0, 0, 0, 1);
Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
```

A primeira liña: glClearColor especifica as cores que van ser usadas cando se limpa o buffer. Un buffer ven ser unha zona de memoria que vai ser 'levada' á GPU para pintar. Dita cor é utilizada pola segunda liña, asinando ó buffer coas cores especificadas.

- **Formato da función glClearColor**:

Leva catro parámetros: Red, Green, Blue, Alfa.

Xa explicamos estes datos anteriormente.

Sempre poñeremos dito código ó principio do método render:

**Clase RendererXogo**
**Obxectivo:** Limpar a pantalla.

```
public void render(float delta){
Gdx.gl.glClearColor(0, 0, 0, 1);
Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
    }
```

**Nota:** Esta vai ser a cor de fondo do noso xogo. Se dito xogo tivera un fondo gráfico que ocupara toda a pantalla non se vería ningunha cor loxicamente.

## 1.9 Visualizando os gráficos

Máis información neste enlace.

Xa vimos nos anteriores exemplos como amosar os gráficos.

Imos velo un pouco máis en profundidade...

Clase SpriteBatch: permite a visualización dun gráfico na pantalla.

Métodos máis importantes:

- **begin-end**: todo o que se debuxe entre estas dúas liñas será enviada á GPU (unidade de procesamento gráfico) dunha soa vez. Isto fai que o rendemento aumente.
- **setProjectionMatrix(Matrix4 projection)**: espera recibir a matriz combinada de proxección e modelado para debuxar os gráficos en función de ditos valores. Normalmente pasaremos como valor a matriz combinada da cámara (camara.combined).
- **Método draw**: ten moitas variantes (está sobrecargado) e serve para debuxar unha textura.
- **Método dispose**: libera os recursos.
- **Método setColor**: asina un tinte á textura.

Nota: Como xa comentamos anteriormente o debuxo de todas as nosas personaxes vaise facer no método render da clase que deriva de Game. No noso caso, dito método chama a un método render (definido por nos) da clase RendererXogo.

Para a utilización do SpriteBacth temos que seguir estes pasos:

1. Definimos un obxecto da clase SpriteBatch e o instanciamos no método create.
2. No método resize indicámoslle que o seu sistema de coordenadas sexa o definido na cámara. O que temos que ter claro neste momento e que estamos a debuxar puntos nos que se lles aplican operacións con matrices para 'colocalos' no seu sitio. Cando definimos a cámara, indicando un tamaño de viewport, estamos creando unha matriz (matriz de proxección) de tal forma que se aplicamos dita matriz a os puntos este se 'colocan' na posición adecuada. Isto o conseguimos coa seguinte liña: **spritebatch.setProjectionMatrix(camaraortho.combined);** Con esta orde estamos a indicar o spritebatch que lle aplique a todo o que debuxe as transformacións gardadas na cámara ortográfica. **Nota:** Isto temos que facelo sempre despois de chamar ó método update da cámara.

3. No método render() borramos a pantalla en cada fotograma e facemos que todo o que se atope entre o begin e o end sexa enviado á vez a unidade gráfica para a súa visualización. Isto aumenta o rendemento xa que é moito mellor enviar un só gráfico que moitos á GPU (Graphics Process Unit).
4. Dentro do begin-end do spritebacth chamamos ó método draw para debuxar as texturas.
5. Cando rematamos (saímos da pantalla) liberamos os recursos. Chamamos ó **método dispose**.

Empecemos a velo na clase RendererXogo.

Neste caso imos utilizar o método draw que leva 5 parámetros: a textura, pos x, pos y, tam x, tam y.

Nota: Lembrar que estamos a traballar con unidades definidas no viewport de cámara, neste exemplo en 300x500 unidades.

**Clase RendererXogo**
**Obxectivo:** Visualizar o alien utilizando o SpriteBatch.

```
package com.plategaxogo2d.renderer;

import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.graphics.GL20;
import com.badlogic.gdx.graphics.OrthographicCamera;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.plategaxogo2d.angel.AssetsXogo;
import com.plategaxogo2d.modelo.Mundo;

public class RendererXogo {

private OrthographicCamera camara2d;
private SpriteBatch spritebatch;

public RendererXogo() {
camara2d = new OrthographicCamera();
spritebatch = new SpriteBatch();
}

/**
         * Debuxa todos os elementos graficos da pantalla
         *
         * @param delta
         *            : tempo que pasa entre un frame e o seguinte.
         */
public void render(float delta) {
Gdx.gl.glClearColor(0, 0, 0, 1);
Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

spritebatch.begin();
spritebatch.draw(AssetsXogo.textureAlien, 100, 100, 15, 15);
spritebatch.end();

}

public void resize(int width, int height) {

                camara2d.setToOrtho(false, Mundo.TAMANO_MUNDO_ANCHO, Mundo.TAMANO_MUNDO_ALTO);
camara2d.update();

spritebatch.setProjectionMatrix(camara2d.combined);

}

public void dispose() {
spritebatch.dispose();
}
}
```

Nota: A clase SpriteBatch ten varios métodos que podemos usar, coma por exemplo setColor(Color tint) or setColor(r,g,b,a) que debuxa un tinte (tiñe) na textura coa cor indicada e co nivel de transparencia que indiquemos (a =alfa; se é igual a 1 é totalmente opaco e con 0 totalmente transparente).

Así se queremos cambiar a cor do alien podemos facelo:

**Clase RendererXogo**
**Obxectivo:** Visualizar outro alien cun tinte diferente.

```
public void render(float delta) {
Gdx.gl.glClearColor(0, 0, 0, 1);
Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

spritebatch.begin();
spritebatch.setColor(Color.YELLOW);
spritebatch.draw(AssetsXogo.textureAlien, 100, 100, 15, 15);
spritebatch.setColor(Color.BLUE);
spritebatch.draw(AssetsXogo.textureAlien, 200, 100, 15, 15);
spritebatch.end();
}
```

## 1.10 Facendo debuger dos gráficos

Inicialmente, cando se desenvolve o xogo, podes non ter a disposición os gráficos (as texturas). En LIBGDX podemos 'sustituir' ditos gráficos por figuras xeométricas para saber onde se atopan os gráficos ou facer probas.

Tamén pode servir para ver, unha vez temos os gráficos, problemas de detección de choques (intersecciónes) de gráficos.

A forma de ver ditos gráficos é moi parecido ó que fixemos có spritebatch anteriormente explicado.

Clase ShapeRenderer: permite debuxar figuras xeométricas na pantalla.

Métodos máis importantes:

- begin - end: Todo o que se atopa entre estas dúas ordes será enviado á vez á tarxeta gráfica para a súa visualización.
- ♦ public void begin(ShapeRenderer.ShapeType type):
  - ◊ parámetro type: indica o tipo de recheo que vai usar. Os valores posibles son puntos, liñas ou recheo. Quere isto dicir que se indicamos recheo (Shatype.filled) a figura xeométrica que creemos estará rechea. O veremos cun exemplo.
- Múltiples métodos para debuxar diferentes figuras xeométricas, como por exemplo:
  - ♦ circle(float x, float y, float radius): debuxa un círculo.
  - ♦ rect(float x, float y, float width, float height): debuxa un rectángulo.
- setColor(Color color): indica a cor das liñas, puntos ou recheo que se vai utilizar. Dito método está sobrecargado e se pode enviar a cor descomposta no seu compoñentes (rgba). Neste caso se fai uso de constantes xa definidas de cores, como por exemplo Color.BLACK.
- **setProjectionMatrix(Matrix4 projection)**: espera recibir a matriz combinada de proxección e modelado para debuxar os gráficos en función de ditos valores. Normalmente pasaremos como valor a matriz comninada da cámara (camara.combined).

O forma de utilizala é o seguinte:

- Definimos a propiedade e a instanciamos no constructor:

**Clase RendererXogo**
**Obxectivo:** Definimos e instanciamos a clase que vai permitir ver os gráficos en forma de figuras xeométricas.

```
private ShapeRenderer shaperender;
        ..................

public RendererXogo() {

camara2d = new OrthographicCamera();
```

```
spritebatch = new SpriteBatch();
shaperender = new ShapeRenderer();
}
```

- No método resize facemos que o sistema de coordenadas da cámara sexa o mesmo no ShapeRenderer (fixemos o mesmo na clase SpriteBatch).

**Clase RendererXogo**
**Obxectivo:** Axustamos o sistema de coordenadas do shaperenderer.

```
public void resize(int width, int height) {

camara2d.setToOrtho(false, Mundo.TAMANO_MUNDO_ANCHO,
Mundo.TAMANO_MUNDO_ALTO);
camara2d.update();

spritebatch.setProjectionMatrix(camara2d.combined);
shaperender.setProjectionMatrix(camara2d.combined);
}
```

- Creamos un método debugger que será chamado dende o método render e que vai ser o que debuxe os gráficos. Como exemplo imos debuxar a bolboreta Candela.

**Clase RendererXogo**
**Obxectivo:** Creamos o método que vai debuxar. Usamos liñas e a cor será azul.

```
/**
         * Debuxa os gráficos en forma de figuras xeométricas
         */
private void debugger(){

shaperender.begin(ShapeType.Line);
shaperender.setColor(Color.BLUE);
shaperender.rect(100,100,15,15);
shaperender.end();

}
```

- Creamos unha propiedade boolean que vai indicar cando amosar o modo debugger e engadimos no método render á chamado ó método debugger.

  Nota: Normalmente este valor sería enviado cando creamos a instancia da clase PantallaXogo no método create da clase MeuXogoGame.

**Clase RendererXogo**
**Obxectivo:** Chamamos ó método debugger en función da propiedade debugger.

```
public class RendererXogo {

private boolean debugger=true;

        ..................

public void render(float delta) {
Gdx.gl.glClearColor(1, 1, 1, 1);
Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

spritebatch.begin();
debuxarCandela();
debuxarBolboretas();
```

```
        spritebatch.end();

        if (debugger){
        debugger();
        }
        }
                ..................
```

## 1.11 Onde obter gráficos para o noso xogo

Se estades a desenvolver un xogo necesitaredes gráficos (se non o sabedes facer vos) para o mesmo.

Se estades pensando vender o xogo ditos gráficos deberían ter unha licenza para usos comerciais.

Algúns dos sitios onde podedes atopar ditos gráficos son:

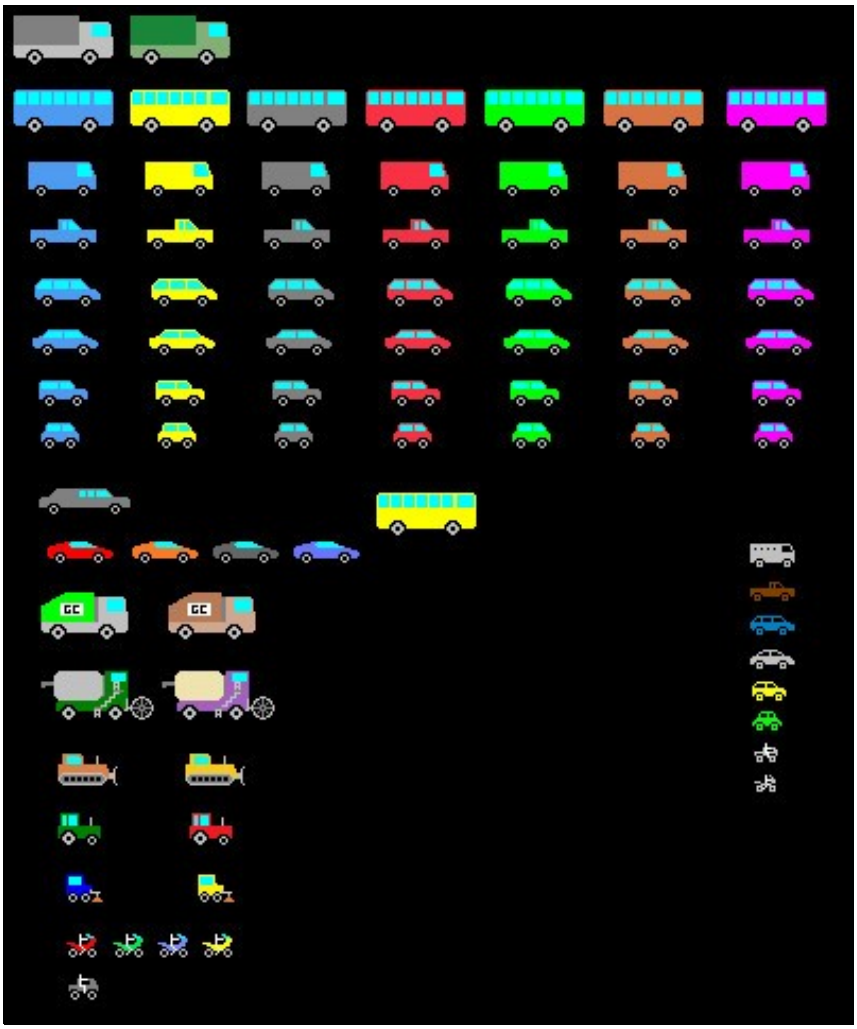
- OPENGAMEART
- ICONARCHIVE

Nos dous sitios podedes buscar por gráficos con licenza para usos comerciais.

## 1.12 Como obter gráficos simples

Seguramente vos sucederá como a min. Cando buscades gráficos para os xogos atopades nas páxinas anteriores moitas posibilidades, pero resulta que tedes moitos gráficos dos atopados que non queredes engadir ó voso proxecto.

Por exemplo:



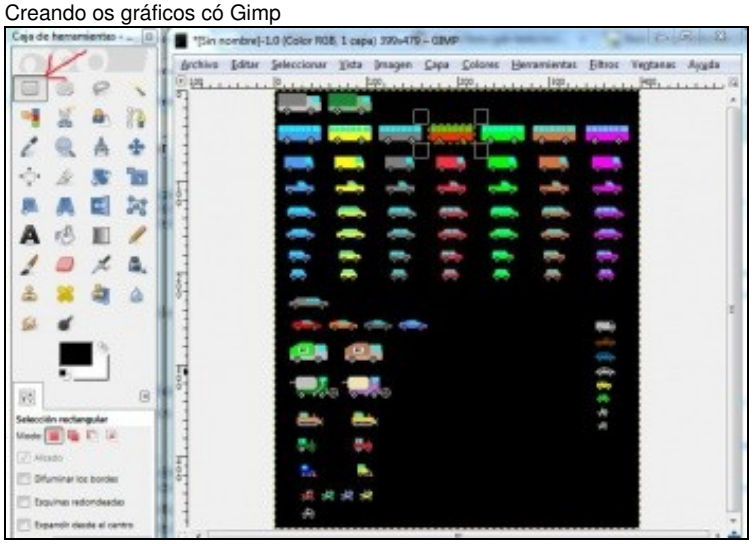
Imaxe obtida de http://opengameart.org/content/basic-2d-car-collection

Nesta imaxe só nos interesa uns poucos dos gráficos para engadilos ó noso proxecto.

O único que temos que facer é cortar e pegar o que nos interese, pero tendo en conta que cando exportemos o gráfico teremos que desmarcar algunha das opcións por defecto.

Lembrar que como comentamos anteriormente, se necesitamos que o gráfico teña algunha cor transparente teremos que exportalo como png e se non é necesario como jpg.

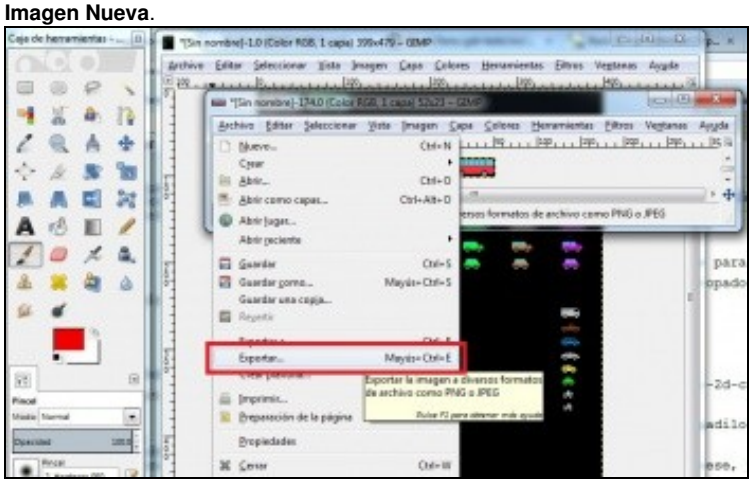
Se o facemos có Gimp:

- Creando os gráficos có Gimp
-


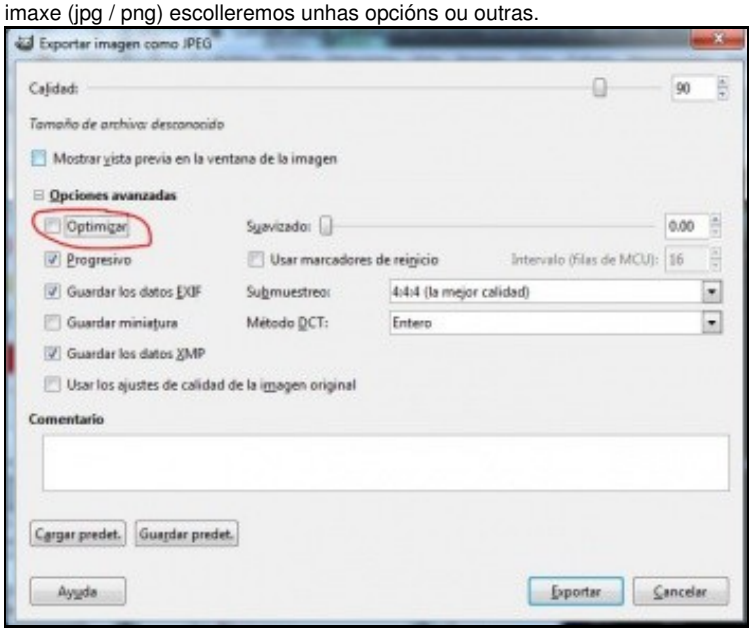
Escollemos a ferramenta de selección e seleccionamos o gráfico que nos interese.

- 

Escollemos a opción do menú de Gimp **Edición => Copiar** e despois escollemos no mesmo menú a opción **Edición => Pegar como => Imagen Nueva**.

- 


Unha vez pegada teremos unha nova xanela coa imaxe. Teremos que ir a opción de menú **Archivo => Exportar**. Dependendo do tipo de imaxe (jpg / png) escolleremos unhas opcións ou outras.

- 


Se é JPG **teremos que desmarcar** a opción que ven por defecto **Optimizar**.

- 


Se é PNG deixaremos as opcións por defecto.

## 1.13 Tarefas avanzadas

---

**TAREFA AVANZADA OPTATIVA 1:** Esta parte está asociada á realización dunha tarefa avanzada. Uso da Atlas..

---

---

**TAREFA AVANZADA OPTATIVA 2:** Esta parte está asociada á realización dunha tarefa avanzada. Uso da clase AssetManager.

---

-- Ángel D. Fernández González -- (2014).