

# Apps Real-Time y Gráficos 3D

alun.evans@upf.edu

# Contenidos

¿Que vamos a aprender?

Teoria Graficos

Teoria Apps Tiempo Real

OpenGL – 2D y 3D

Cocos2D – 2D



# Contenidos

Introducción a Apps Real-Time/Juegos para iOS

Conceptos Básicos de Gráficos 2D + 3D

Introducción a OpenGL

OpenGL y GLKit – cuatro ejemplos

El Scene Graph

Realidad Aumentada

Cocos2D prácticas 2D

OpenGL prácticas 3D

Pipelines para juegos

(Unity3D)

# Estructura de un App real-time

# Juegos

FPS

First Person Shooter

Acción/  
3º Persona

Deportes  
Real-time

Deportes  
Gestión

Simulacion  
'Realista'

RTS

Real Time Strategy

TBS

Turn Based Strategy

RPG

Role Playing Game

MMORPG

Massive Multiplayer

Puzzle

Real Time

Puzzle

por turnos

Visualizador  
Objetos 3D

Realidad  
Aumentada

Herramientas  
3D

# Opciones de crear un Juego/ App Real Time para iOS



..... y mas....

# Opciones de crear un Juego/ App Real Time



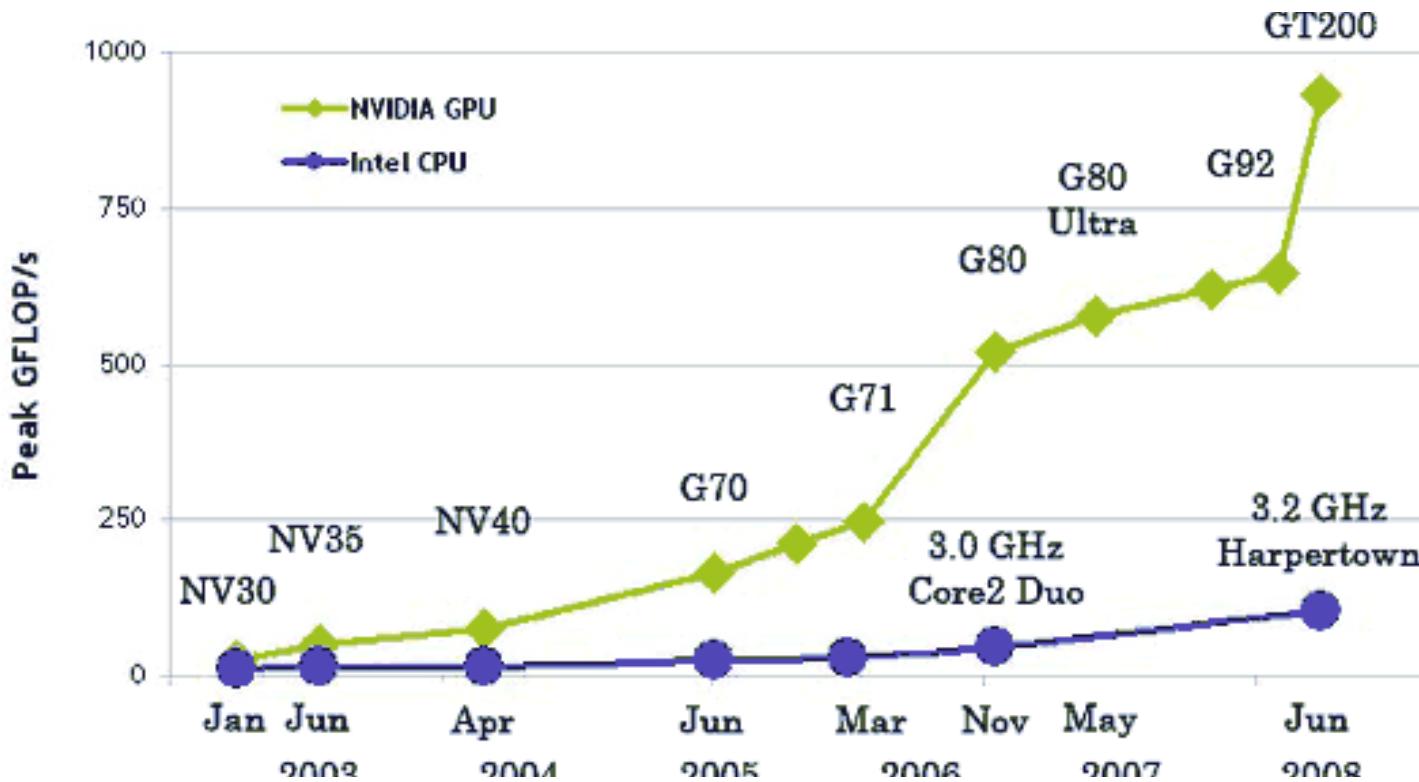
Aprendiendo OpenGL nativo, y los conceptos básicos de gráficos, nos ayuda mucho en entender sistemas de mas alto nivel

# Hoy

Tema	OpenGL	Cocos 2D	Unity3D
Intro a OpenGL	✓		
Conceptos básicos de Gráficos	✓	✓	✓
Texturas y Sprites	✓	✓	✓
Matemática básica	✓	✓	✓
El Update Loop	✓	✓	✓
Gestionar eventos iOS	✓	✓	
El Scene Graph	✓	✓	✓

# Conceptos básicos de gráficos

El GPU  
Pipeline de gráficos  
Shaders  
Primitivas  
Aliasing



## CPU vs GPU

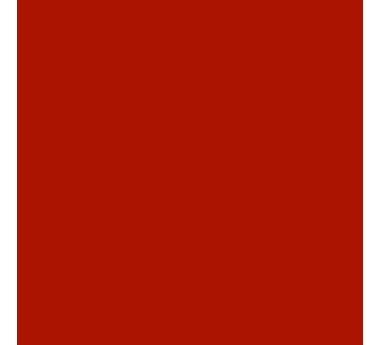
El mundo de gráficos 3D cambió radicalmente con la introducción de GPU

CPU:

Programas complicados en serie

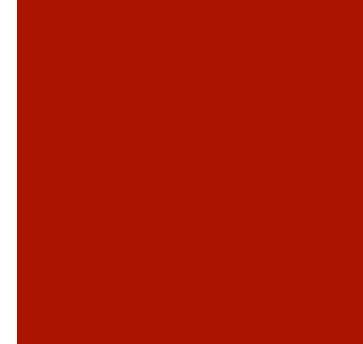
GPU:

Programas mas sencillos en paralelo



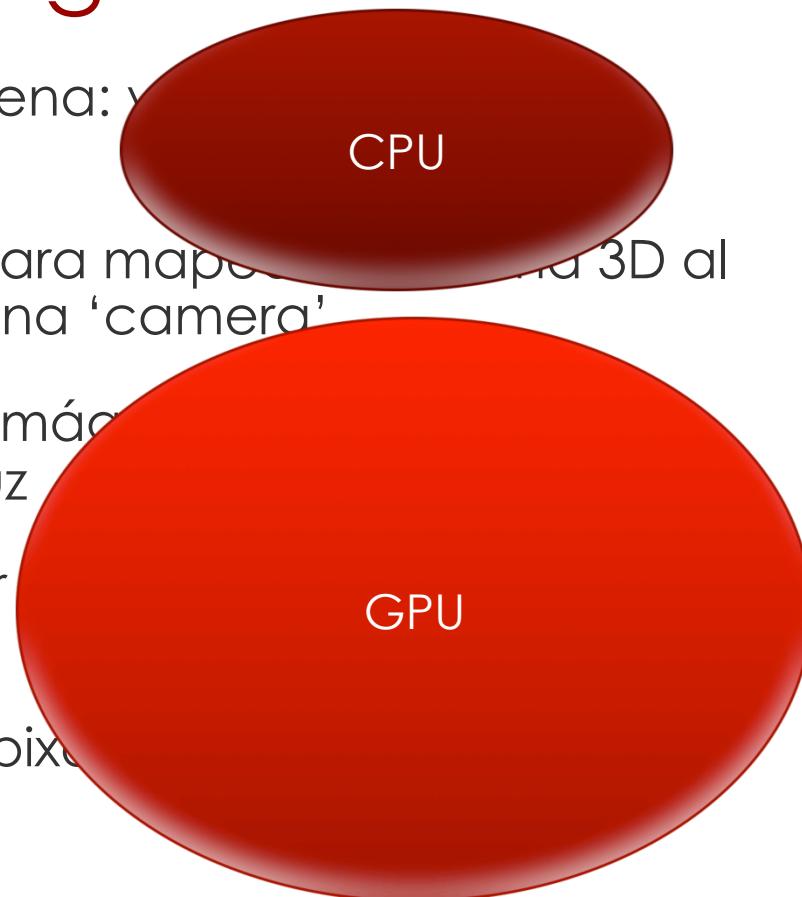
# El *Pipeline* de gráficos

1. Descripción de escena: vértices, triángulos, colores, luces
2. Transformaciones para mapear la escena 2D/3D al punto de vista de una 'camera'
3. 'Efectos': texturas (imágenes), sombras, calculaciones de luz
4. Rasterizar: Convertir Geometría a pixeles en la pantalla
5. Procesamiento de pixeles: pruebas de fondo y stencil



# El *Pipeline* de gráficos

1. Descripción de escena: vértices, colores, luces
2. Transformaciones para mapear la escena 3D al punto de vista de una 'camera'
3. 'Efectos': texturas (imágenes), sombras, calculaciones de luz
4. Rasterizar: Convertir la escena en pixeles de pantalla
5. Procesamiento de pixeles: colores, efectos de stencil



# Fixed vs Programmable Pipeline

- Open GL ES 1.0/1.1 (y Cocos 2D 1.x) solo soporta el *Fixed-Pipeline*, donde todos los efectos de luz y gráficos estan especificados ('hard-codeadas') por OpenGL
- Open GL ES 2.0 (y Cocos 2D 2.x) usa el *Programmable-Pipeline*, en el que usamos Shaders para hacer calculaciones de posición de vertices y de luz etc.

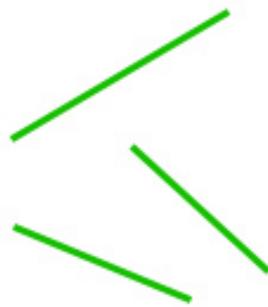
Con el programmable pipeline, nosotros nos encargamos de escribir mas código para controlar nuestra escena – es mas trabajo, pero tambien mucho mas control!

# Primitivas

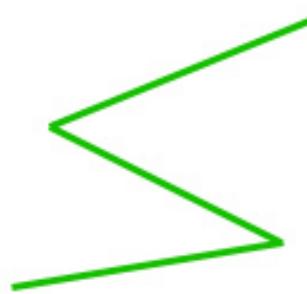
`GL_POINTS`



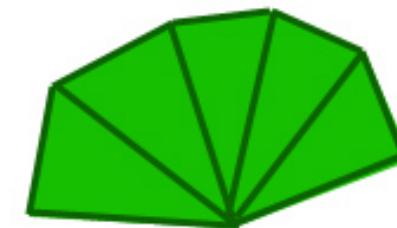
`GL_LINES`



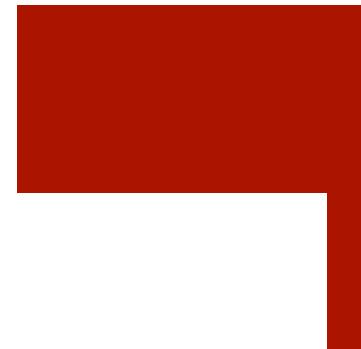
`GL_LINE_STRIP`



`GL_TRIANGLES`



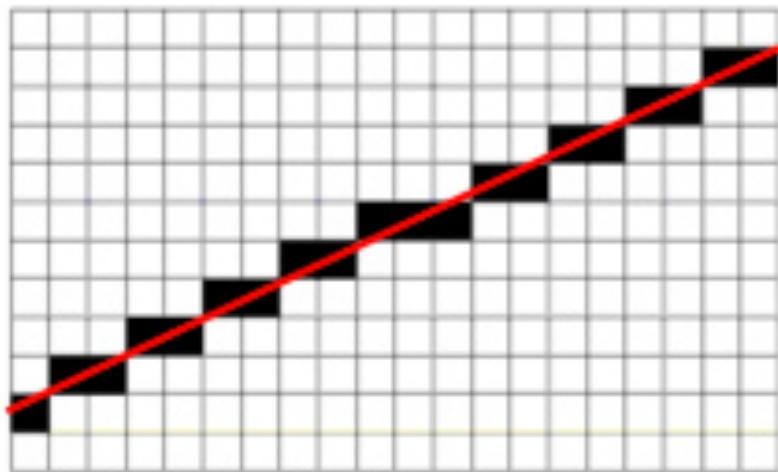
`GL_QUADS`



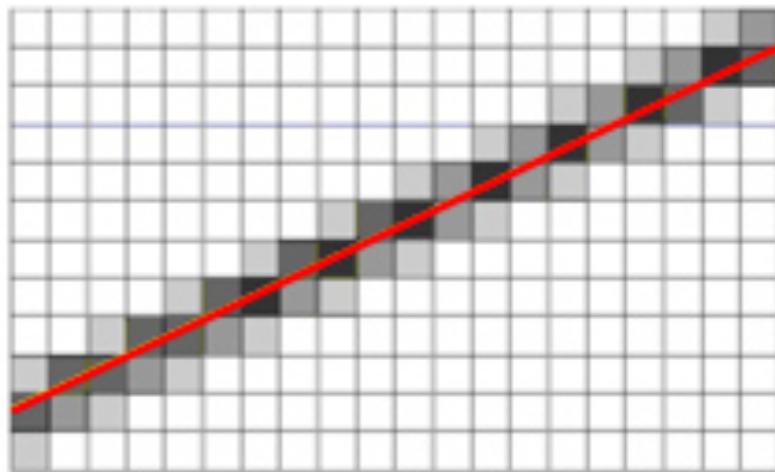
`GL_TRIANGLE_FAN`

# Aliasing y Antialiasing

El rasterizado de primitivas produce dientes de sierra en los contornos que pueden deteriorar la calidad de la imagen final.

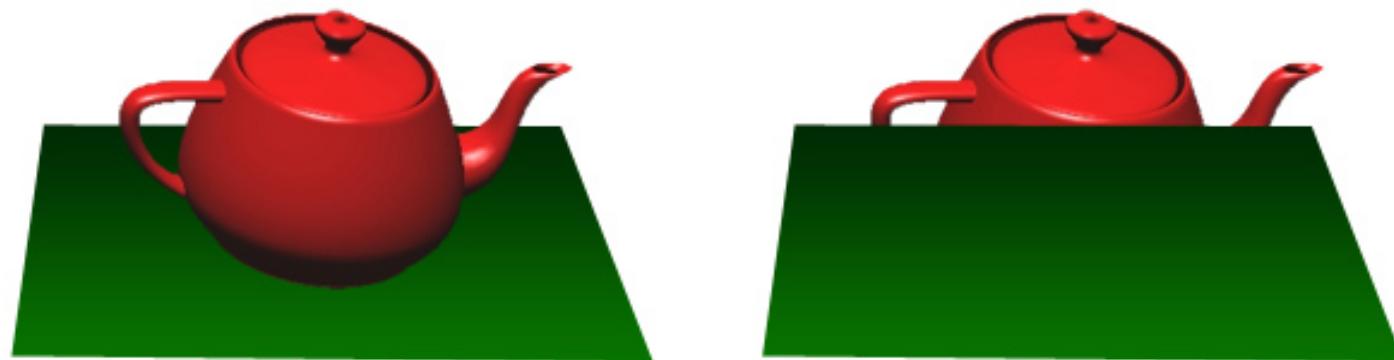


Aliasing



Anti-aliasing

# Z-Buffer y occlusiones



El modelo ‘pintor’ no nos sirve

En entornos 3D solemos querer que OpenGL tenga en cuenta qué objetos están delante de qué objetos (occlusiones en Z) a la hora de pintar (en función de su profundidad).

Para ello podemos activar el ZBuffer (tambien llamado DepthBuffer), entonces se almacenará el valor de profundidad de cada pixel en un buffer aparte

OpenGL es un  
API que nos  
permite usar  
estos conceptos



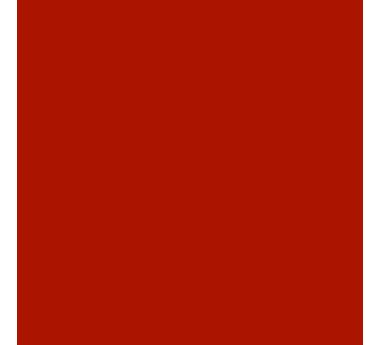
# ¿Qué es OpenGL?

**OpenGL** (**O**pen **G**raphics **L**ibrary) es una especificación **estándar** que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D.

Existen distintas versiones que representan conjuntos de funcionalidades (OpenGL 1.2, OpenGL ES ...)

# Ventajas

- Accede al **hardware** dedicado que lo acelera considerablemente (el GPU)
- Es **independiente** del Hardware
- Hay bindings para **cualquier** lenguaje (C, Python, Java, Javascript, C#...) y es **bajo nivel**.



# ¿Qué NO es OpenGL

Un lenguaje de programación. OpenGL se puede usar con casi cualquier lenguaje existente.

Un programa de dibujo, OpenGL solo pinta primitivas gráficas (triangulos, quads)

Un motor gráfico, no tiene funciones de carga de recursos, gestion de escena, GUI, etc. (que tiene, por ejemplo, DirectX)

Un gestor de entornos, no permite detectar colisiones, seleccionar objetos con el raton, etc.

Un motor de videojuegos, no gestiona temas de audio, fisica, interacción, etc.

Cocos2D y Unity están 'encima' de OpenGL

# Acciones de OpenGL



- Pintar figuras geométricas básicas (puntos, líneas y triángulos)
  - Efectuar transformaciones sobre los vértices (translaciones, rotaciones y escalados)
  - Definir proyección para convertir de coordenadas 3D a la pantalla 2D.
  - Tener en cuenta occlusiones en el pintado de primitivas.
  - Texturizar las figuras en base a una imagen y a coordenadas de textura para cada vértice.
  - Definir el algoritmo de pintado del pixel (shaders)
  - Definir el algoritmo de transformación del vértice.
  - Definir el algoritmo de teselación de una malla.
- etc.....

Todo esto se puede simplificar a: **OpenGL sirve para pintar triángulos en pantalla de forma eficiente.**

# OpenGL ES

**OpenGL ES (OpenGL for Embedded Systems)** es una variante simplificada de la API gráfica OpenGL diseñada para dispositivos integrados tales como teléfonos móviles, PDAs y consolas de videojuegos

Aunque la base de OPEN GL ES es muy parecida a Open GL, existen unas diferencias importantes

# Versiones de OpenGL ES

- 1.0 – Mucha funcionalidad quitada de OpenGL normal.
- 1.1 Un poco mejor pero todavía básica
- 2.0 OpenGL ‘de verdad’

# Como conectar a iOS

# Interactuar con el OS

## Lo que Open GL no hace:

- Crear ventana
- Gestionar input
- Inferfaz usuario
- Cargar texturas (imágenes)
- Gestionar tareas del sistema (p.ej.  
una llamada!)

Para todo esto  
necesitamos nuestro  
propio manera de  
interactuar con el Sistema  
Operativo

p.ej. Existe el *GLUT Toolkit* para gestionar estas tareas en Windows

# iOS5+ y GLKit

- A partir de iOS5 tenemos el framework *GLKit*

## Class References

[GLKBaseEffect](#)  
[GLKEffectProperty](#)  
[GLKEffectPropertyFog](#)  
[GLKEffectPropertyLight](#)  
[GLKEffectPropertyMatrix](#)  
[GLKEffectPropertyOrientation](#)  
[GLKEffectPropertyPosition](#)  
[GLKReflectionManager](#)  
[GLKSkyboxEffect](#)  
[GLKTextureInfo](#)  
[GLKTextureLoader](#)  
[GLKView](#)  
[GLKViewController](#)

## Protocol References

[GLKNamedEffect](#)  
[GLKViewControllerDelegate](#)  
[GLKViewDelegate](#)  
[Manager References](#)

## Other References

[GLKit Effects Constants](#)  
[GLKMatrix3](#)  
[GLKMatrix4](#)  
[GLKMatrixStack](#)  
[Quaternion](#)  
[Vector2](#)  
[Vector3](#)  
[Vector4](#)  
[History](#)

## GLKit:

Crear ViewController listo para OpenGL  
Cargar Texturas (imágenes)  
Métodos de matemática  
Shaders ('Efectos') básicos

# Práctica: Nuestro primer App OpenGL/GLKit

1. New Project – Empty Application
2. Añadir frameworks:
  - QuartzCore.framework
  - OpenGLES.framework
  - GLKit.framework
3. Crear sub-clase de GLKViewController
  - import GLKit.h
4. Añadir archivo storyboard
5. Arrastrar GLKit View Controller al storyboard y cambiar el tipo a nuestra clase de 3.
6. Asignar el storyboard en el Project Settings

# Cambiar el app delegate

```
- (BOOL)application:(UIApplication *)application  
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {  
  
    //quitar código de esta parte y devolver YES  
  
    return YES;  
}
```

```
#import "MainViewController.h"
@interface MainViewController ()
    @property (strong, nonatomic)EAGLContext *context;
@end

@implementation MainViewController
@synthesize context = _context;
- (void)viewDidLoad {
    [super viewDidLoad];
    self.context = [[EAGLContext alloc] initWithAPI:kEAGLRenderingAPIOpenGLGLES2];
    if (!self.context) {
        NSLog(@"Failed to create ES context");
    }

    GLKView *view = (GLKView *)self.view;
    view.context = self.context;
    [EAGLContext setCurrentContext:self.context];
}

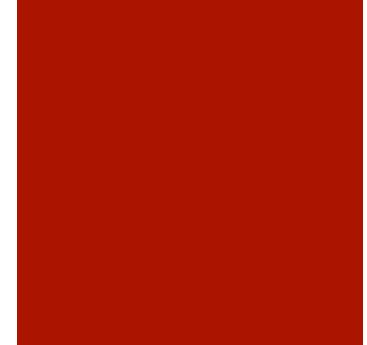
#pragma mark - GLKViewDelegate
- (void)glkView:(GLKView *)view drawInRect:(CGRect)rect {
    glClearColor(0, 104.0/255.0, 55.0/255.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT); }

- (void)update {
}

@end
```

```
#import "MainViewController.h"
@interface MainViewController ()  
    @property (strong, nonatomic)EAGLContext *context;  
@end  
  
  
@implementation MainViewController  
@synthesize context = _context;  
- (void)viewDidLoad {  
    [super viewDidLoad];  
    self.context = [[EAGLContext alloc] initWithAPI:kEAGLRenderingAPIOpenGLGLES2];  
    if (!self.context) {  
        NSLog(@"Failed to create ES context");  
    }  
  
    GLKView *view = (GLKView *)self.view;  
    view.context = self.context;  
    [EAGLContext setCurrentContext:self.context];  
}  
  
.....snip.....  
  
#pragma mark - GLKViewDelegate  
- (void)glkView:(GLKView *)view drawInRect:(CGRect)rect {  
    glClearColor(0, 104.0/255.0, 55.0/255.0, 1.0);  
    glClear(GL_COLOR_BUFFER_BIT); }  
  
- (void)update {  
}  
  
@end
```

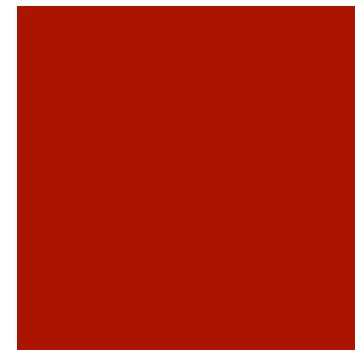
Crear el  
context para  
dibujar a la  
pantalla



```
#import "MainViewController.h"
@interface MainViewController ()
```

```
    @property (strong, nonatomic)EAGLContext *context; ←
```

Crear el  
context para  
dibujar a la  
pantalla



```
@implementation MainViewController
```

```
@synthesize context = _context;
```

```
- (void)viewDidLoad {
```

```
    [super viewDidLoad];
```

```
    self.context = [[EAGLContext alloc] initWithAPI:kEAGLRenderingAPIOpenGL2]; ←
```

```
    if (!self.context) {
```

```
        NSLog(@"Failed to create ES context");
```

```
}
```

Iniciar como  
tipo  
OpenGL2

```
GLKView *view = (GLKView *)self.view;
```

```
view.context = self.context;
```

```
[EAGLContext setCurrentContext:self.context]; ←
```

```
}
```

Asociar el  
objecto View

.....snip.....

```
#pragma mark - GLKViewDelegate
```

```
- (void)glkView:(GLKView *)view drawInRect:(CGRect)rect {
```

```
    glClearColor(0, 104.0/255.0, 55.0/255.0, 1.0);
```

```
    glClear(GL_COLOR_BUFFER_BIT); }
```

```
- (void)update {
```

```
}
```

```
@end
```

```
#import "MainViewController.h"  
@interface MainViewController ()  
 @property (strong, nonatomic)EAGLContext *context;  
@end
```

Crear el  
context para  
dibujar a la  
pantalla

```
@implementation MainViewController  
@synthesize context = _context;  
- (void)viewDidLoad {  
    [super viewDidLoad];  
    self.context = [[EAGLContext alloc] initWithAPI:kEAGLRenderingAPIOpenGL2];  
    if (!self.context) {  
        NSLog(@"Failed to create ES context");  
    }
```

Iniciar como  
tipo  
OpenGL2

```
GLKView *view = (GLKView *)self.view;  
view.context = self.context;  
[EAGLContext setCurrentContext:self.context];  
}
```

Asociar el  
objecto View

.....snip.....

```
#pragma mark - GLKViewDelegate  
- (void)glkView:(GLKView *)view drawInRect:(CGRect)rect {  
    glClearColor(0, 104.0/255.0, 55.0/255.0, 1.0);  
    glClear(GL_COLOR_BUFFER_BIT); }
```

```
- (void)update {  
}
```

Pintar el color  
verde

```
@end
```

```
#import "MainViewController.h"  
@interface MainViewController ()  
    @property (strong, nonatomic)EAGLContext *context;  
@end
```

Crear el  
context para  
dibujar a la  
pantalla

```
@implementation MainViewController
```

```
@synthesize context = _context;
```

```
- (void)viewDidLoad {  
    [super viewDidLoad];  
    self.context = [[EAGLContext alloc] initWithAPI:kEAGLRenderingAPIOpenGL2];  
    if (!self.context) {  
        NSLog(@"Failed to create ES context");  
    }
```

Iniciar como  
tipo  
OpenGL2

```
GLKView *view = (GLKView *)self.view;
```

```
view.context = self.context;
```

```
[EAGLContext setCurrentContext:self.context];
```

```
}
```

Asociar el  
objecto View

.....snip.....

```
#pragma mark - GLKViewDelegate
```

```
- (void)glkView:(GLKView *)view drawInRect:(CGRect)rect {  
    glClearColor(0, 104.0/255.0, 55.0/255.0, 1.0);  
    glClear(GL_COLOR_BUFFER_BIT); }
```

Pintar el color  
verde

```
- (void)update {
```

Sobreescribir el  
metodo  
update

```
@end
```

# GLKViewController y GLKView son sub-clases



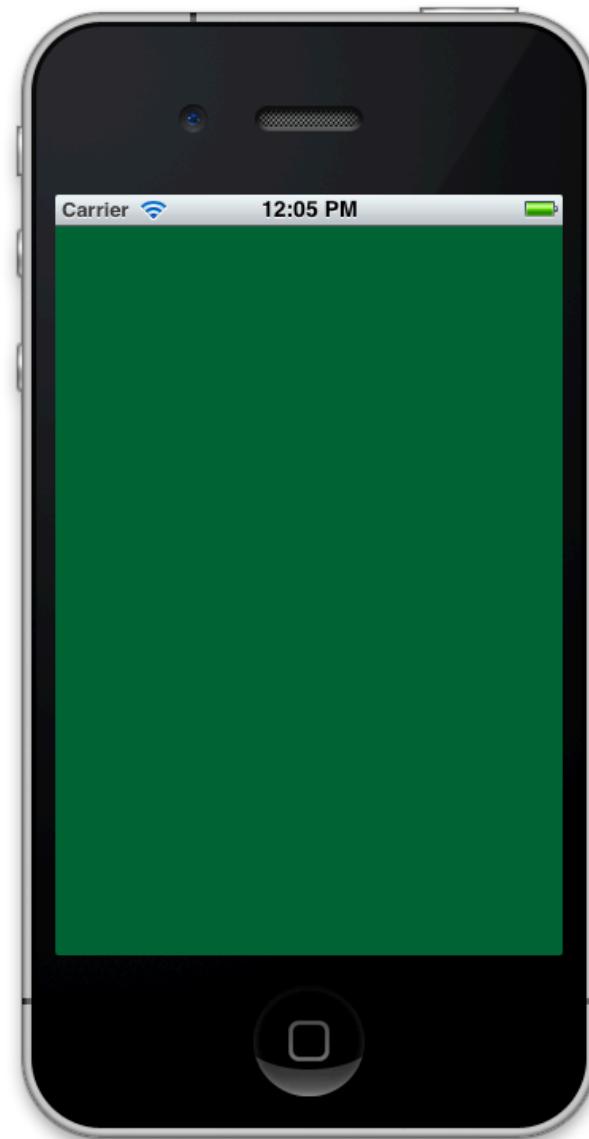
## GLKViewController Class Reference

Inherits from	<a href="#">UIViewController : UIResponder : NSObject</a>
Conforms to	<a href="#">NSCoding</a> <a href="#">GLKViewDelegate</a> <a href="#">NSCoding (UIViewController)</a> <a href="#">UIAppearanceContainer (UIViewController)</a> <a href="#">NSObject (NSObject)</a>
Framework	<a href="#">/System/Library/Frameworks/GLKit.framework</a>
Availability	Available in iOS 5.0 and later.
Declared in	<a href="#">GLKViewController.h</a>

## GLKView Class Reference

Inherits from	<a href="#">UIView : UIResponder : NSObject</a>
Conforms to	<a href="#">NSCoding</a> <a href="#">NSCoding (UIView)</a> <a href="#">UIAppearance (UIView)</a> <a href="#">UIAppearanceContainer (UIView)</a> <a href="#">NSObject (NSObject)</a>
Framework	<a href="#">/System/Library/Frameworks/GLKit.framework</a>
Availability	Available in iOS 5.0 and later.

# Hello OpenGL!

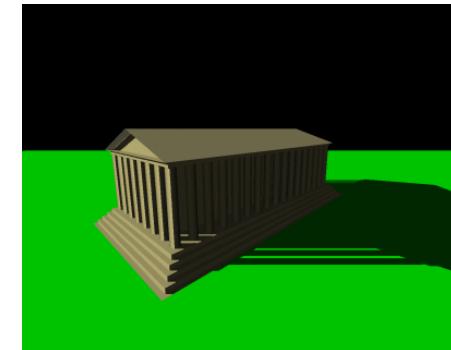


# Texturas

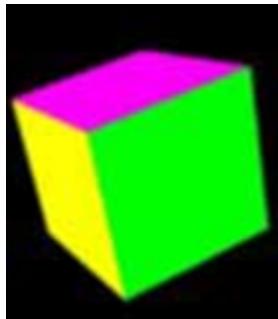
Añadir imágenes a nuestra escena

# ¿Qué es una textura?

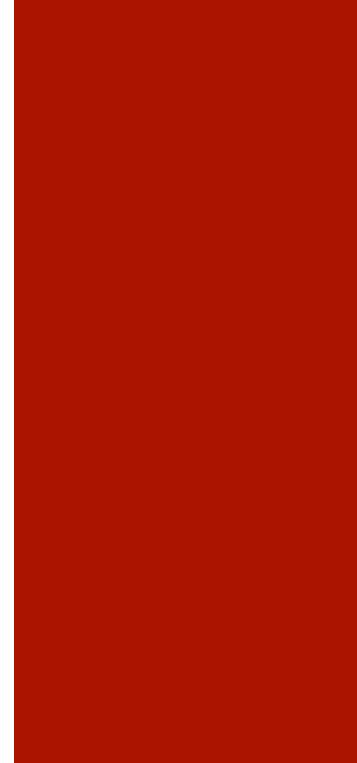
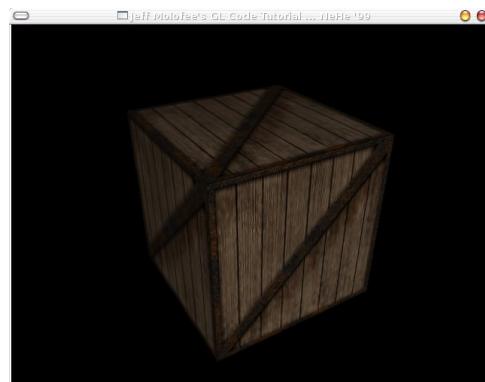
- Mediante una malla de polígonos podemos definir la topología (forma) de un objeto de nuestro entorno, pero incluso si lo coloreamos con algún algoritmo de sombreado, seguirá viendose demasiado plano y sintetico.



- Para darle un acabado realista podemos colorear las caras que lo forman utilizando imágenes.
- Las texturas son **imágenes** pensadas para ser pintadas sobre la superficie de un objeto 3D (malla).



+

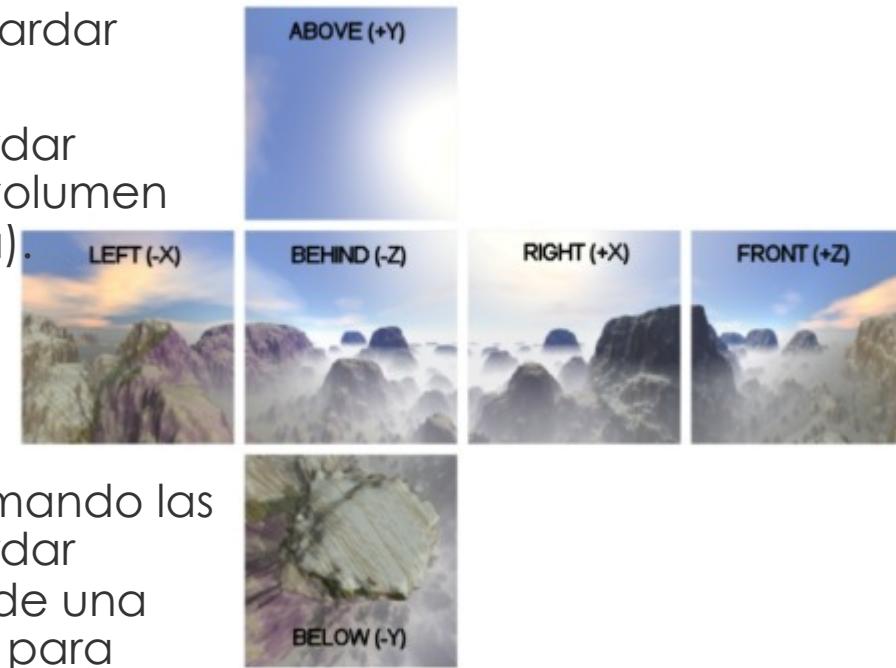


# ¿Qué tipo de texturas hay?

- Principalmente trabajamos con texturas **bidimensionales** pero los APIs permiten otras texturas:

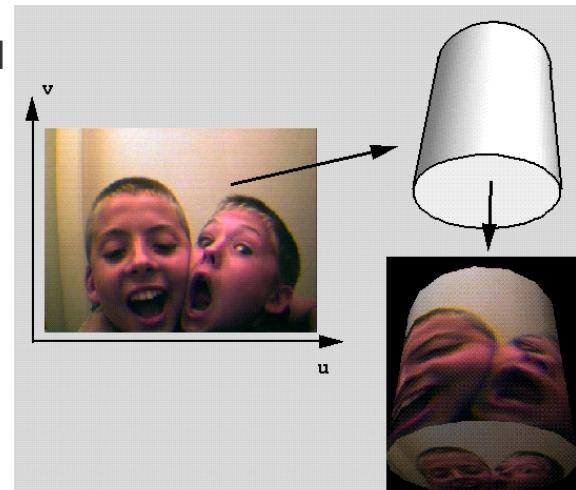
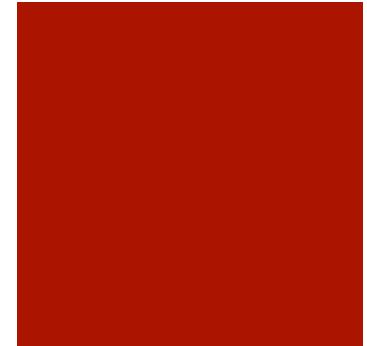
- **Unidimensionales** utilizadas para guardar degradados.
- **Tridimensionales** que permiten guardar información sobre los voxels de un volumen (empleada en visualización médica).

- **Cubemaps** que son 6 imágenes formando las caras de un cubo, se usa para guardar información sobre todos los puntos de una esfera (pero se simplifica a un cubo para ahorrar en memoria).



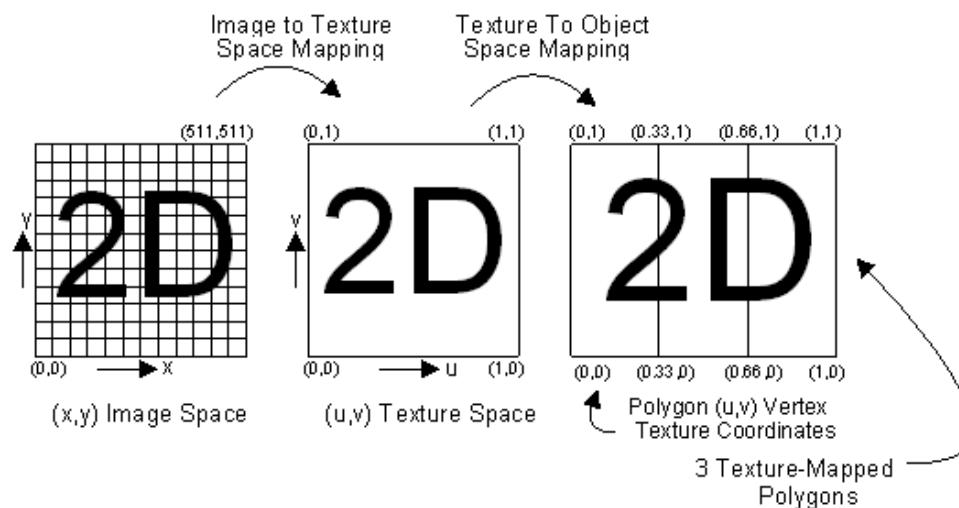
# Coordenadas de textura

- Uno de los principales inconvenientes de utilizar texturas 2D es que estas son bidimensionales mientras que nuestros objetos son tridimensionales.
- Necesitamos una manera de **asociar la imagen 2D con los triangulos 3D** de la malla.
- Para ello usamos las **coordenadas de textura** (tambien llamadas uvs en referencia a las componentes U y V).



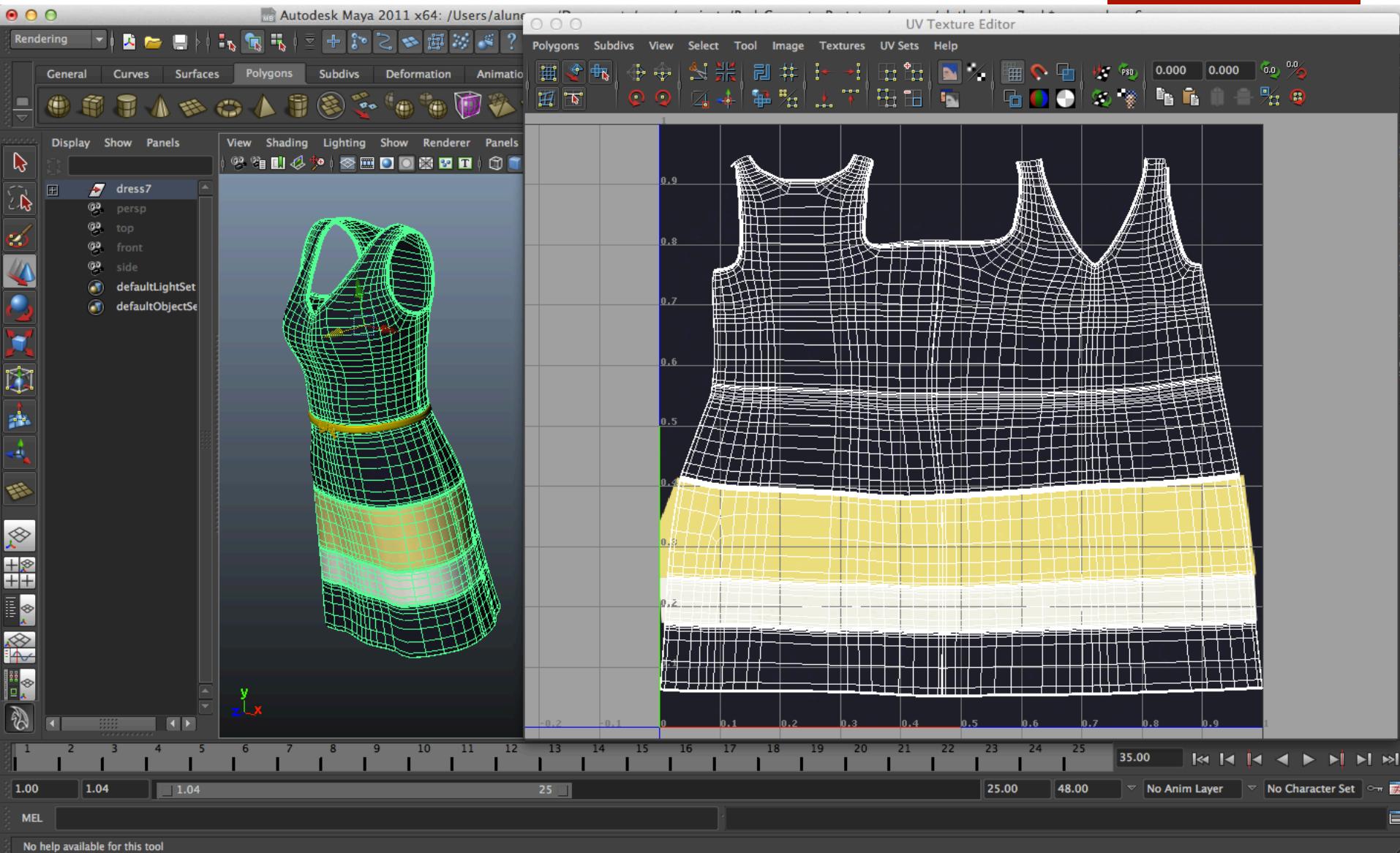
# Coordenadas de textura

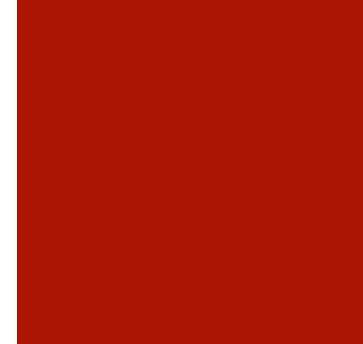
- Las coordenadas de textura **son valores 2D** (entre 0 y 1) que nos indican para cada vértice de cada cara de la malla qué punto de la imagen le corresponde, en coordenadas normalizadas.
- Luego el hardware se encargará de llenar el polígono interpolando con los píxeles interiores de la región definida por las coordenadas de textura.



A la hora de pintar el polígono **le decimos a cada vértice qué pixel** de la imagen le **corresponde**, pero para hacerlo independiente del tamaño de la imagen, se lo indicamos usando coordenadas normalizadas, es decir, 0,0 sería la esquina izquierda superior de la imagen y 1,1 la inferior derecha.

# Texturas





# Tiling

En algunos casos podemos querer que sobre un único polígono se repita la textura muchas veces, para ello podemos especificar números fuera del rango 0->1, por ejemplo, si pintamos un cuadrado con uvs que van de 0 a 10, la imagen se repetirá 10 veces en cada componente.

A esto se le llama **tiling**



# Texturas y GLKit

- GLKit nos **ayuda** con unas funciones muy utiles para cargar texturas

```
NSDictionary * options = [NSDictionary dictionaryWithObjectsAndKeys:  
    [NSNumber numberWithBool:YES],  
    GLKTextureLoaderOriginBottomLeft,  
    nil];
```

```
NSError * error;  
NSString *path = [[NSBundle mainBundle]  
    pathForResource:fileName ofType:nil];
```

```
self.textureInfo = [GLKTextureLoader textureWithContentsOfFile:path  
    options:options error:&error];
```

# Texturas y GLKit

- GLKit nos ayuda con unas funciones muy útiles para cargar texturas

Especifica coordenadas de texturas (mismas como OpenGL)

```
NSDictionary * options = [NSDictionary dictionaryWithObjectsAndKeys:  
    [NSNumber numberWithBool:YES],  
    GLKTextureLoaderOriginBottomLeft,  
    nil];
```

```
NSError * error;  
NSString *path = [[NSBundle mainBundle]  
    pathForResource:fileName ofType:nil];
```

```
self.textureInfo = [GLKTextureLoader textureWithContentsOfFile:path  
    options:options error:&error];
```

# Texturas y GLKit

- GLKit nos ayuda con unas funciones muy utiles para cargar texturas

```
NSDictionary * options = [NSDictionary dictionaryWithObjectsAndKeys:  
    [NSNumber numberWithBool:YES],  
    GLKTextureLoaderOriginBottomLeft,  
    nil];
```

```
NSError * error;  
NSString *path = [[NSBundle mainBundle]  
    pathForResource:fileName ofType:nil];
```

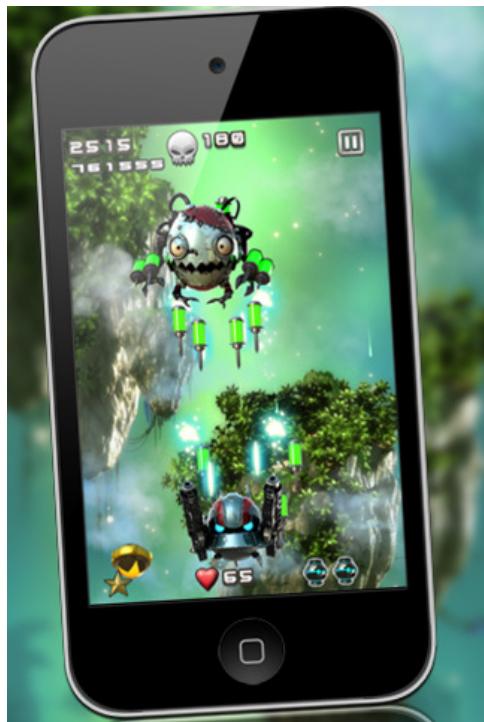
```
self.textureInfo = [GLKTextureLoader textureWithContentsOfFile:path  
    options:options error:&error];
```

Especifica coordenadas de texturas (mismas como OpenGL)

Usamos el textureInfo para acceder al tamaño de la imagen mas tarde

# Pero ¿cómo pintamos esta textura en la pantalla?

Lo aplicamos en encima de un objeto de  
polygonos. *Read on...*



# Juego 2D

Para mostrar el concepto de pintar texturas



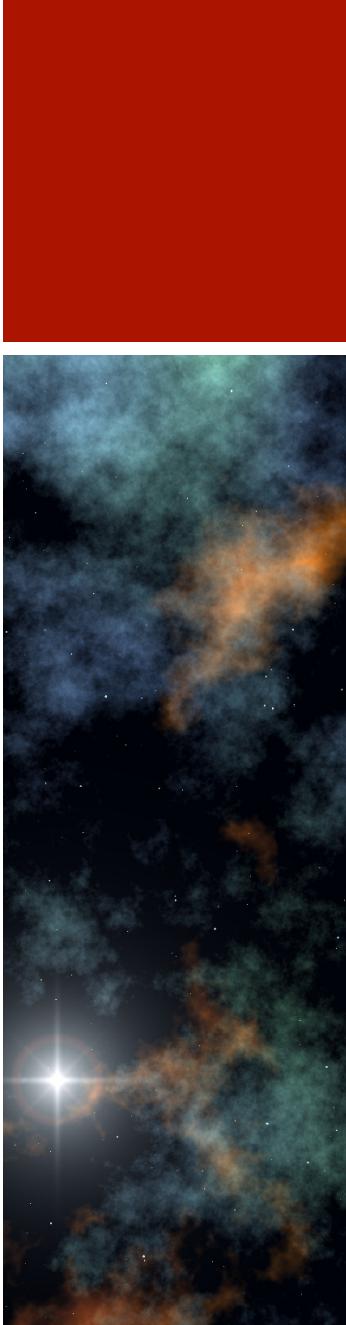
# Game v 0.1

Setup básico app GLKit

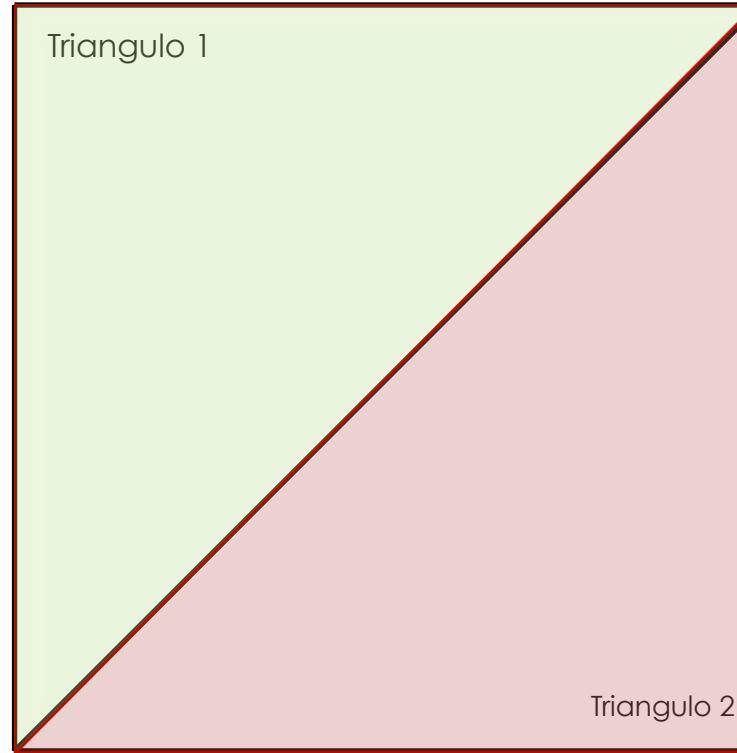
Crear clase sprite

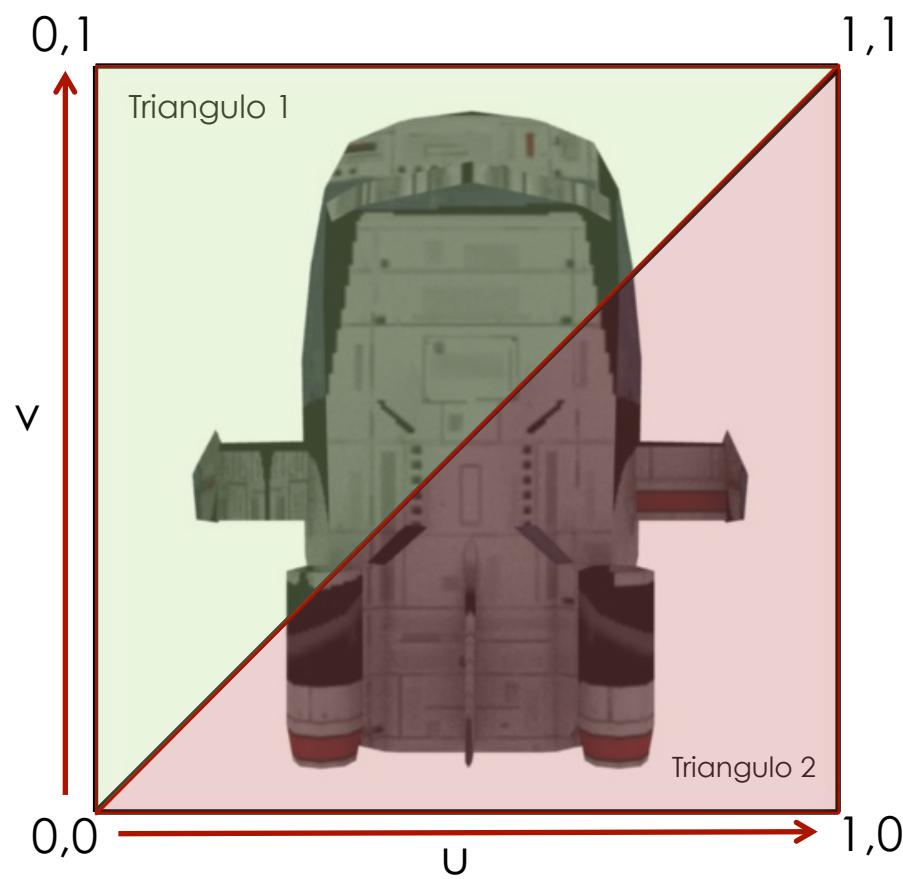
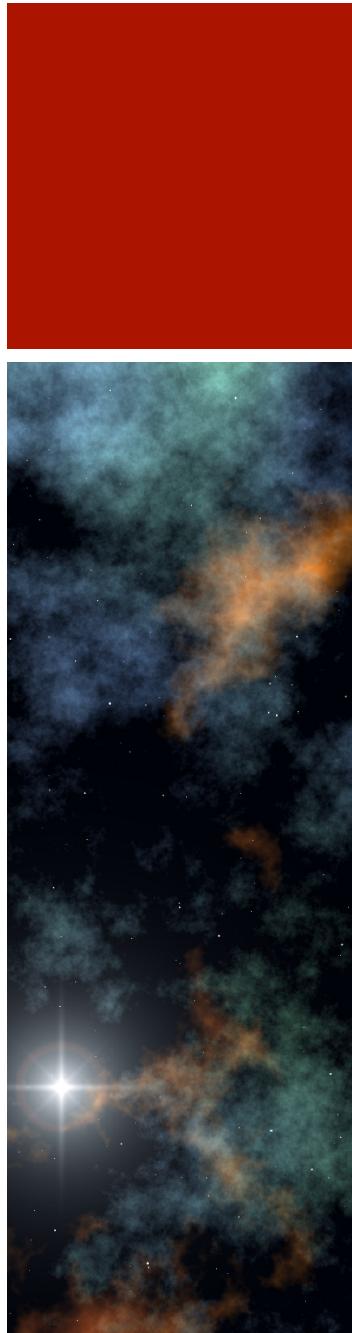
Renderizar Sprite (imagen 2D) en  
la pantalla





# Pintar una imagen 2D OpenGL





# Código para definir clase Sprite en Objective-C

```
typedef struct {  
    CGPoint geometryVertex;  
    CGPoint textureVertex;  
} TexturedVertex;
```

```
typedef struct {  
    TexturedVertex bl;  
    TexturedVertex br;  
    TexturedVertex tl;  
    TexturedVertex tr;  
} TexturedQuad;
```

# Código para definir clase Sprite en Objective-C

```
typedef struct {  
    CGPoint geometryVertex;  
    CGPoint textureVertex;  
} TexturedVertex;
```

```
typedef struct {  
    TexturedVertex bl;  
    TexturedVertex br;  
    TexturedVertex tl;  
    TexturedVertex tr;  
} TexturedQuad;
```

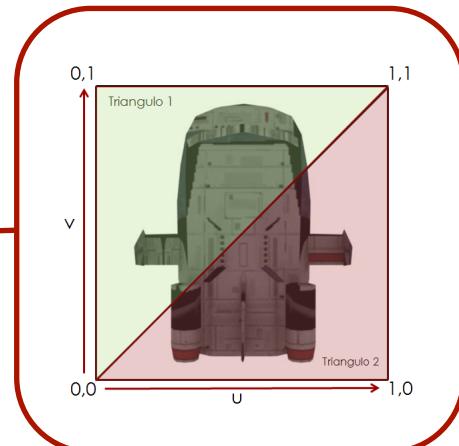
Definición de un vértice que tiene coordenadas de textura

Usamos la clase CGPoint, pero podemos usar cualquier cosa que guarda floats

# Código para definir clase Sprite en Objective-C

```
typedef struct {  
    CGPoint geometryVertex;  
    CGPoint textureVertex;  
} TexturedVertex;
```

```
typedef struct {  
    TexturedVertex bl;  
    TexturedVertex br;  
    TexturedVertex tl;  
    TexturedVertex tr;  
} TexturedQuad;
```



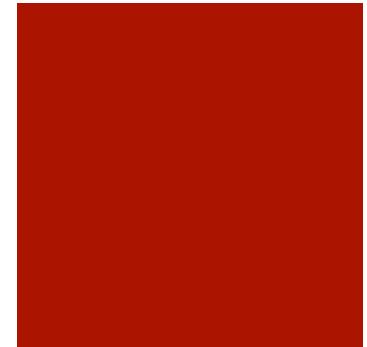
# Creando el Sprite

```
TexturedQuad newQuad;  
newQuad.bl.geometryVertex = CGPointMake(0, 0);  
newQuad.br.geometryVertex = CGPointMake(self.textureInfo.width, 0);  
newQuad.tl.geometryVertex = CGPointMake(0, self.textureInfo.height);  
newQuad.tr.geometryVertex = CGPointMake(self.textureInfo.width,  
                                         self.textureInfo.height);
```

Cambiamos la geometria al tamaño de la textura – si nuestra imagen mide 47x78, así será el tamaño del quad

**Nota:** esto es un caso especial para el sprite 2D.  
En gráficos 3D, casi siempre trabajamos con texturas cuadradas

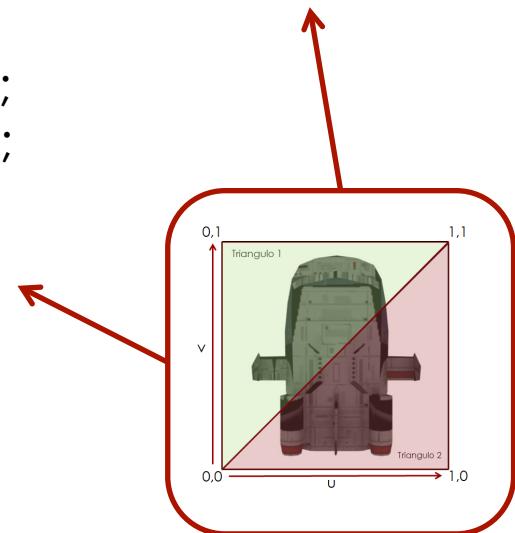
# Creando el sprite (2)



```
TexturedQuad newQuad;  
newQuad.bl.geometryVertex = CGPointMake(0, 0);  
newQuad.br.geometryVertex = CGPointMake(self.textureInfo.width, 0);  
newQuad.tl.geometryVertex = CGPointMake(0, self.textureInfo.height);  
newQuad.tr.geometryVertex = CGPointMake(self.textureInfo.width,  
                                         self.textureInfo.height);
```

```
newQuad.bl.textureVertex = CGPointMake(0, 0);  
newQuad.br.textureVertex = CGPointMake(1, 0);  
newQuad.tl.textureVertex = CGPointMake(0, 1);  
newQuad.tr.textureVertex = CGPointMake(1, 1);
```

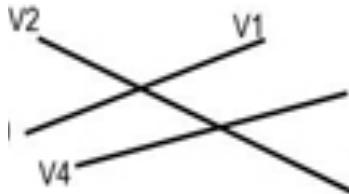
```
self.quad = newQuad;
```



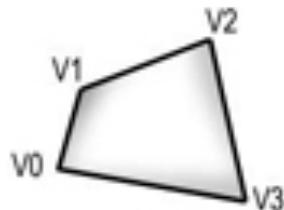
# Render Buffers

Decir a OpenGL que tenemos que pintar

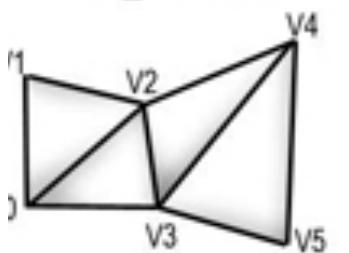
# OpenGL ES vs Open GL normal



GL\_LINES



GL\_QUADS

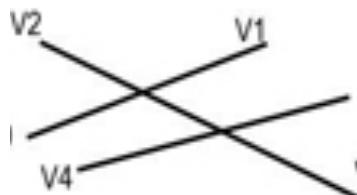


GL\_TRIANGLES\_STRIP

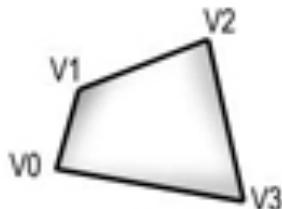
En OpenGL 'normal', la manera mas común de dibujar en la pantalla es algo parecido a:

```
glBegin(GL_TRIANGLES);  
    glVertex3f(1.0f, 1.0f, 1.0f);  
    glVertex3f(1.0f, 5.0f, 1.0f);  
    glVertex3f(5.0f, 5.0f, 1.0f);  
glEnd();
```

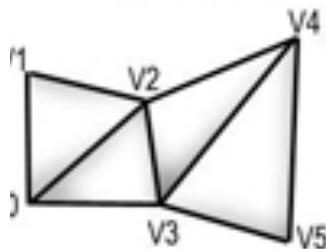
Definir cada elemento, uno por uno  
(por su puesto, se puede usar bucles!)



GL\_LINES



GL\_QUADS



GL\_TRIANGLES\_STRIP

# Open GL ES usa buffers

Hay tres pasos:

`glEnableVertexAttribArray()`

Activar el buffer de vertices

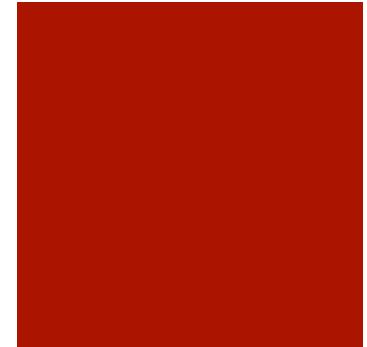
`glVertexAttribPointer()`

Definir un buffer

`glDrawArrays()`

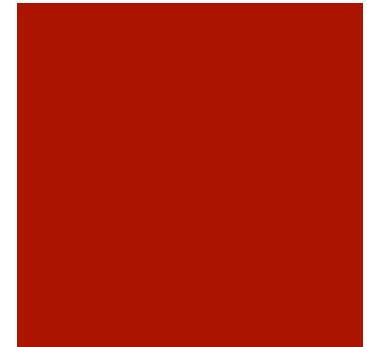
Dibujar

# Dibujar nuestro Sprite



```
- (void)render {  
    // .... Setup effect ...  
    glEnableVertexAttribArray(GLKVertexAttribPosition);  
    glEnableVertexAttribArray(GLKVertexAttribTexCoord0);  
  
    long offset = (long)&_quad; //la referencia a la parte memoria de los datos  
  
    glVertexAttribPointer(GLKVertexAttribPosition, // Este buffer contiene vértices  
        2, // Cada vertice tiene dos componentes (x, y)  
        GL_FLOAT, // cada componente es un GL_FLOAT  
        GL_FALSE, // (no normalizar nada)  
        sizeof(TexturedVertex), //adelantamos esta distance en el buffer  
        //abajo es un puntero al primer vértice  
        (void *) (offset + offsetof(TexturedVertex, geometryVertex)));  
  
    glVertexAttribPointer(GLKVertexAttribTexCoord0, 2, GL_FLOAT, GL_FALSE,  
        sizeof(TexturedVertex), (void *) (offset + offsetof(TexturedVertex, textureVertex)));  
  
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4); //dibujar triángulos, adelantar 4 veces en el buffer  
}
```

# Añadir el shader (1/2)



Setup  
OpenGL

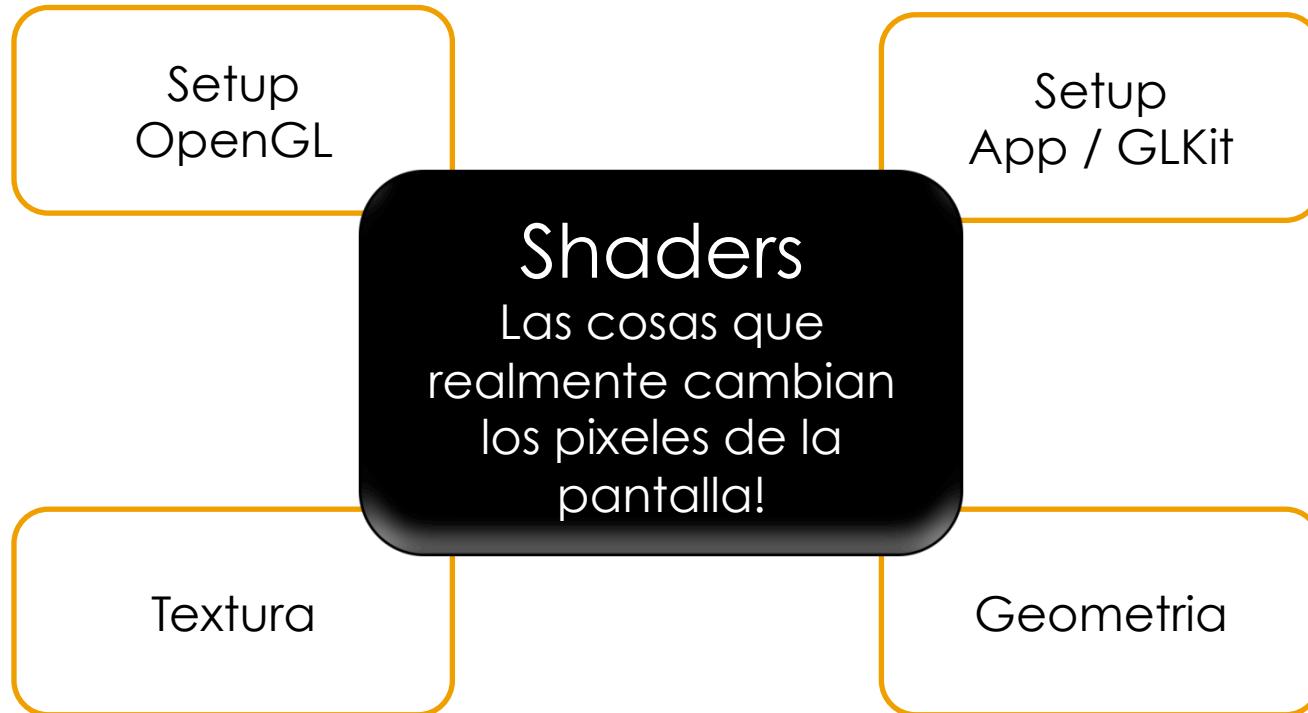
Setup  
App / GLKit

Textura

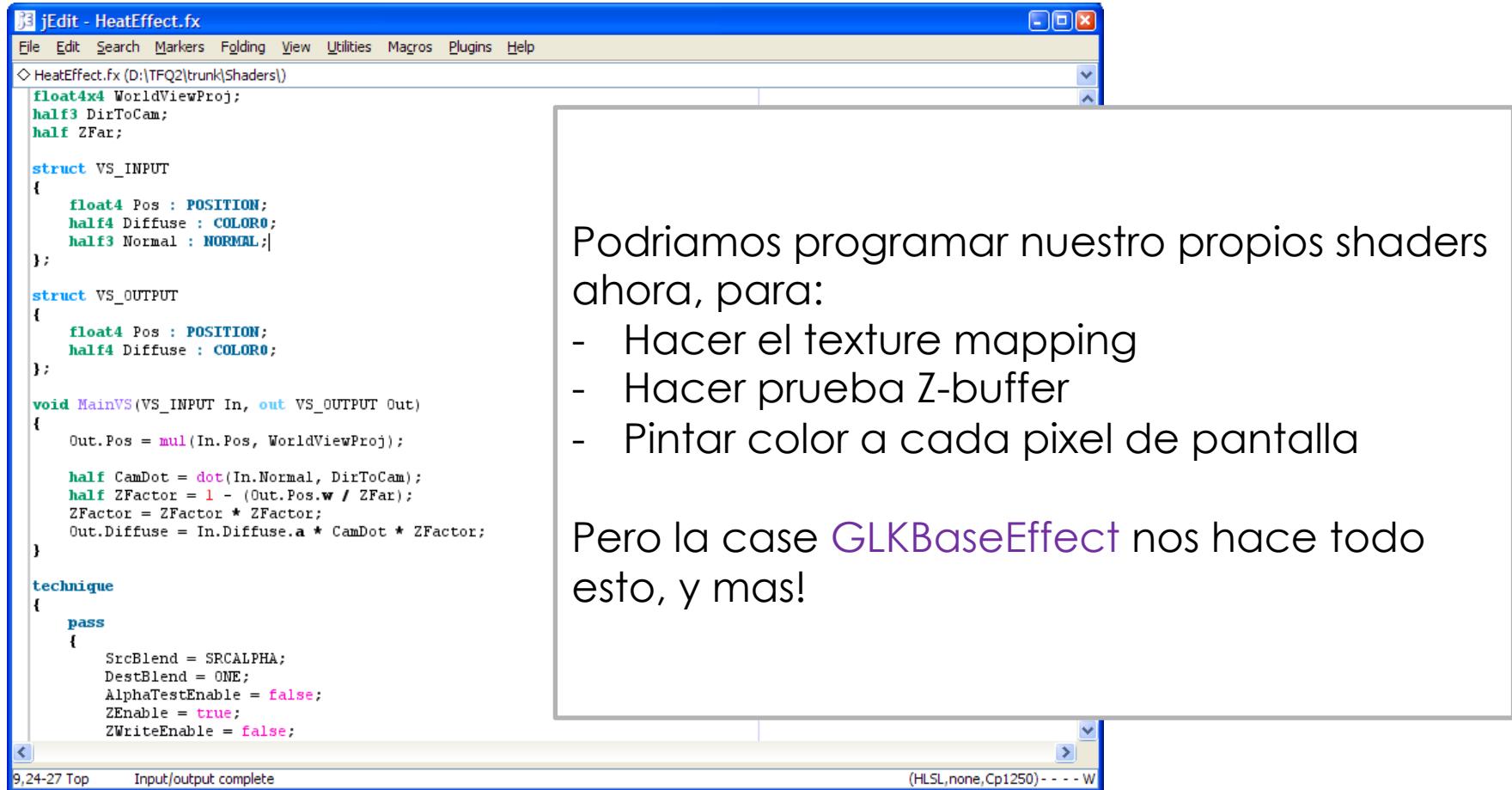
Geometria



# Añadir el shader (2/2)



# GLKit to the rescue!



The screenshot shows a window titled "jEdit - HeatEffect.fx" containing a fragment of HLSL code. The code defines a vertex structure (VS\_INPUT) with position, diffuse color, and normal, and a corresponding output structure (VS\_OUTPUT). It includes a MainVS function that performs a world-view-projection transformation and calculates a Z-factor. A technique block contains a pass block with blending and depth settings. The status bar at the bottom indicates "Input/output complete" and "(HLSL,none,Cp1250) - - - W".

```
float4x4 WorldViewProj;
half3 DirToCam;
half ZFar;

struct VS_INPUT
{
    float4 Pos : POSITION;
    half4 Diffuse : COLOR0;
    half3 Normal : NORMAL;
};

struct VS_OUTPUT
{
    float4 Pos : POSITION;
    half4 Diffuse : COLOR0;
};

void MainVS(VS_INPUT In, out VS_OUTPUT Out)
{
    Out.Pos = mul(In.Pos, WorldViewProj);

    half CamDot = dot(In.Normal, DirToCam);
    half ZFactor = 1 - (Out.Pos.w / ZFar);
    ZFactor = ZFactor * ZFactor;
    Out.Diffuse = In.Diffuse.a * CamDot * ZFactor;
}

technique
{
    pass
    {
        SrcBlend = SRCALPHA;
        DestBlend = ONE;
        AlphaTestEnable = false;
        ZEnable = true;
        ZWriteEnable = false;
    }
}
```

Podriamos programar nuestro propios shaders ahora, para:

- Hacer el texture mapping
- Hacer prueba Z-buffer
- Pintar color a cada pixel de pantalla

Pero la clase **GLKBaseEffect** nos hace todo esto, y mas!

# En viewDidLoad de la sub-clase GLKViewController

```
self.effect = [[GLKBaseEffect alloc] init];  
GLKMatrix4 projectionMatrix =  
    GLKMatrix4MakeOrtho(0, 480, 0, 320, -1024, 1024);  
self.effect.transform.projectionMatrix = projectionMatrix;  
self.player = [[Sprite alloc] initWithFile:@"Player.png" effect:self.effect];
```

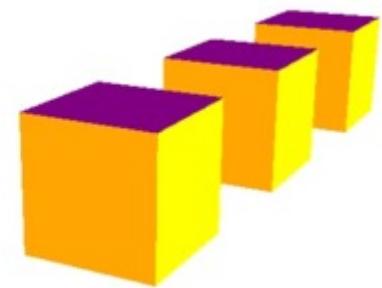
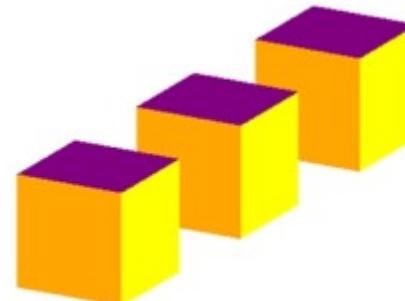
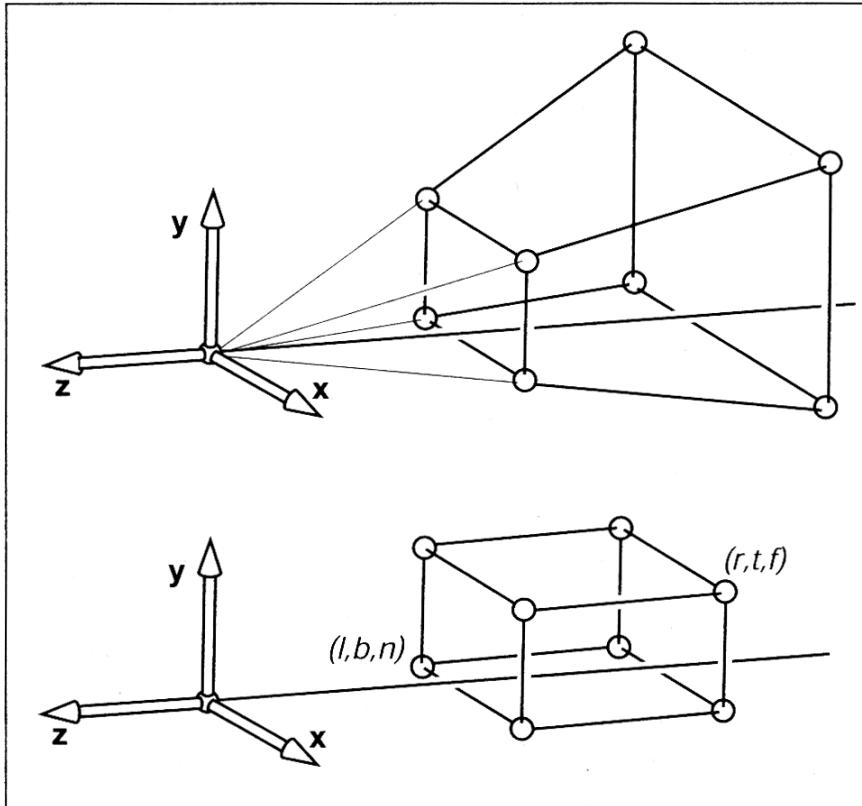
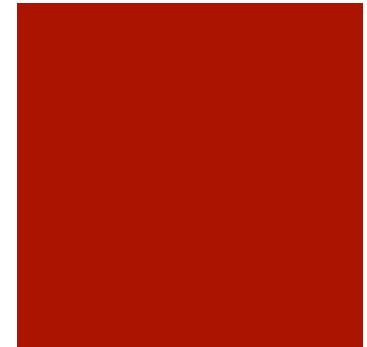
# En viewDidLoad de la sub-clase GLKViewController

```
self.effect = [[GLKBaseEffect alloc] init];  
GLKMatrix4 projectionMatrix =  
    GLKMatrix4MakeOrtho(0, 480, 0, 320, -1024, 1024);  
self.effect.transform.projectionMatrix = projectionMatrix;
```

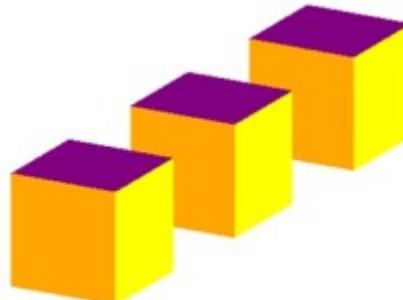
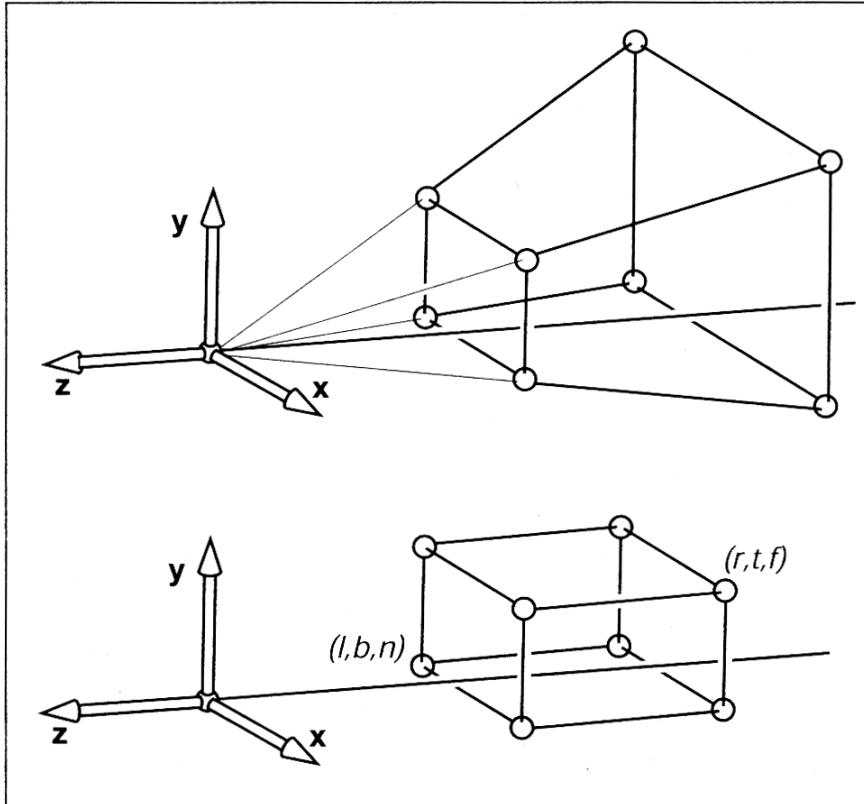
```
self.player = [Player
```

Hay dos tipos de matriz de proyección:  
Perspectiva y Ortográfica

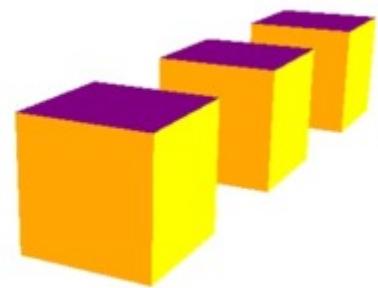
# Perspectivo vs Ortografico



# Perspectivo vs Ortografico



Orthographic Projection



Perspective Projection

Para nuestro juego 2D,  
usamos ortografico, por que  
nos tenemos 3D

# Resultado hasta ahora!

No es nada especial!

# En viewDidLoad de la sub-clase GLKViewController

```
self.effect = [[GLKBaseEffect alloc] init];
```

```
GLKMatrix4 projectionMatrix =
```

```
    GLKMatrix4MakeOrtho(0, 480, 0, 320, -1024, 1024);
```

```
self.effect.transform.projectionMatrix = projectionMatrix.
```

```
self.player =
```

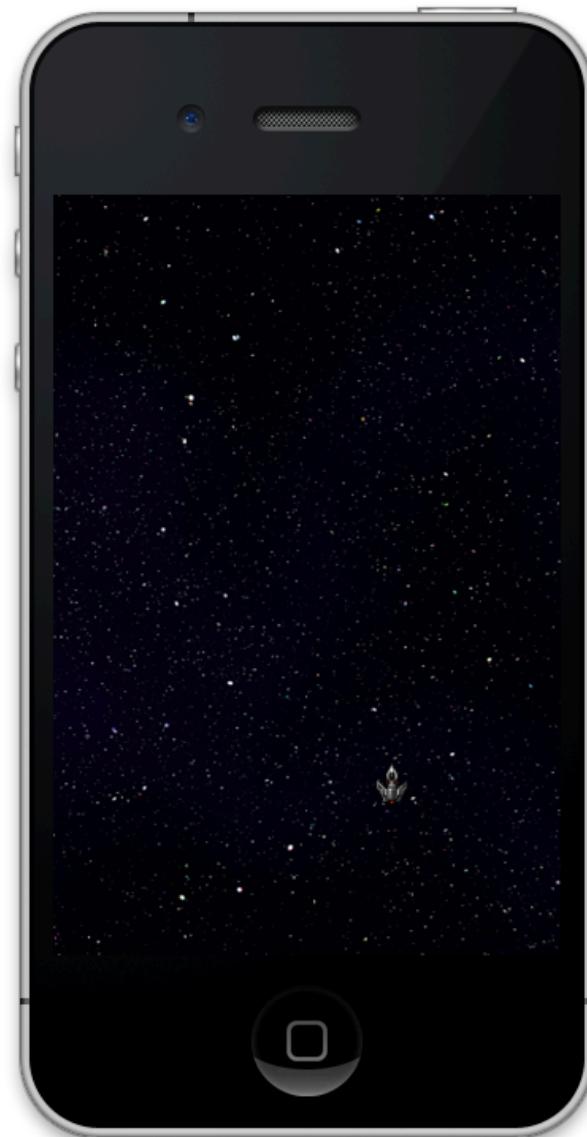
Aunque hay una diferencia entre las resoluciones de pantallas de modelos de iPhone diferentes, Apple no ayuda diciendo que el retina display tiene la misma cantidad de 'puntos' a la pantalla antigua.

Es decir:

CGRect screenFrame = [[UIScreen mainScreen] applicationFrame];  
siempre nos devuelve 320x480 para cualquier iPhone

# Game v0.2

Cambiar posición sprite  
Introducir metodo update – el  
*update loop*  
Detectando input usuario  
Tiler de fondo



# Transformaciones Geometricas

En 2D y 3D



# ¿Para qué sirven las transformaciones?

- Permiten **colocar** los diferentes objetos en el escenario.
- Son conceptos fáciles de entender y **asociar a acciones** de la vida real (mover, agrandar, girar)
- Son **fáciles de modelar** matemáticamente.
- Suelen **respetar la morfología** del objeto (no cambian su forma)
- **No tienen un coste computacional** elevado.
- Pueden **concatenarse** fácilmente

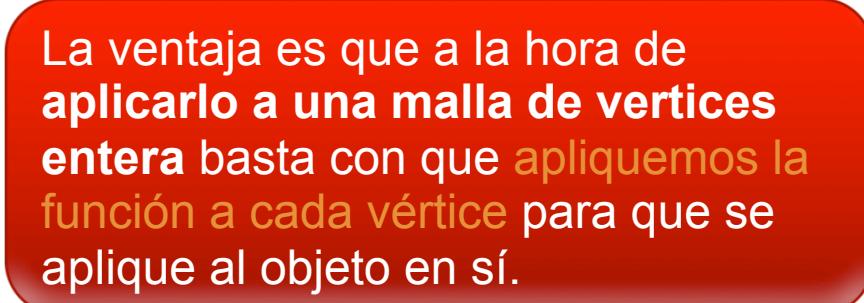


# ¿Cómo se usan en la práctica?

- Podemos interpretar las transformaciones como una función que recibe por parámetro un vértice y retorna dicho vértice transformado:

$$P' = f(P)$$

- Lo único que necesitamos es saber qué transformaciones queremos modelar en dicha función y construir tal función.



La ventaja es que a la hora de **aplicarlo a una malla de vértices entera** basta con que **apliquemos la función a cada vértice para que se aplique al objeto en sí**.

# Hay 3 transformaciones comunes

## ■ **Traslación:**

Desplazar un objeto, cambia su centro de coordenadas

## ■ **Rotación:** Orienta

al objeto en otro sentido aplicando una rotación sobre un eje que pasa por el centro del sistema.

## ■ **Escalado:** Cambia el tamaño del objeto con respecto a su centro de coordenadas.

Aunque existen otras transformaciones más complejas que afectan a la morfología, estas 3 son las más utilizadas ya que son vitales si queremos distribuir una serie de mallas para construir un escenario.

# Traslación: cambiar la posición de un objeto



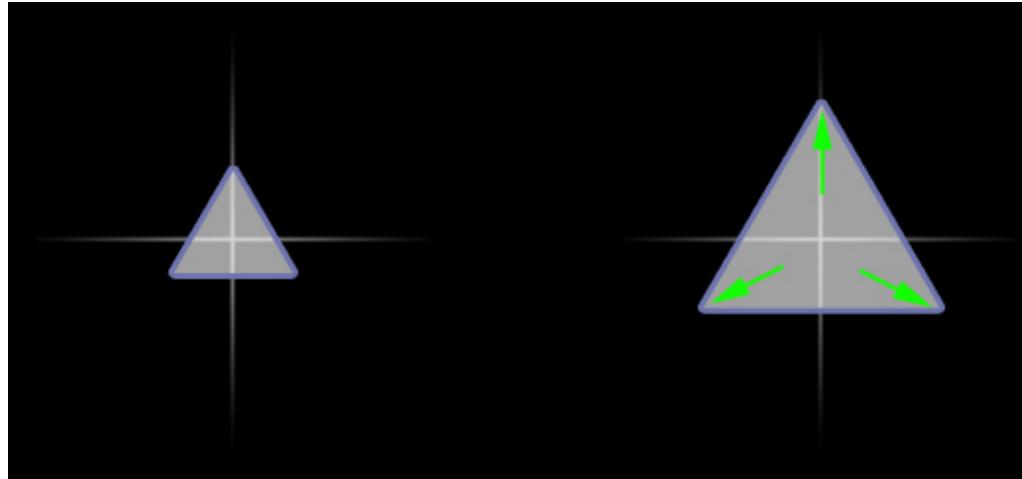
Para trasladar un objeto trasladamos todos sus vértices, a cada punto  $P(x,y)$  le sumamos a las coordenadas del punto las de la translación  $(dx,dy)$ :

$$P'(x',y') = P(x + dx, y + dy)$$

O lo que es lo mismo:

$$P' = P + T$$

# Escalado: cambiar el tamaño de un objeto

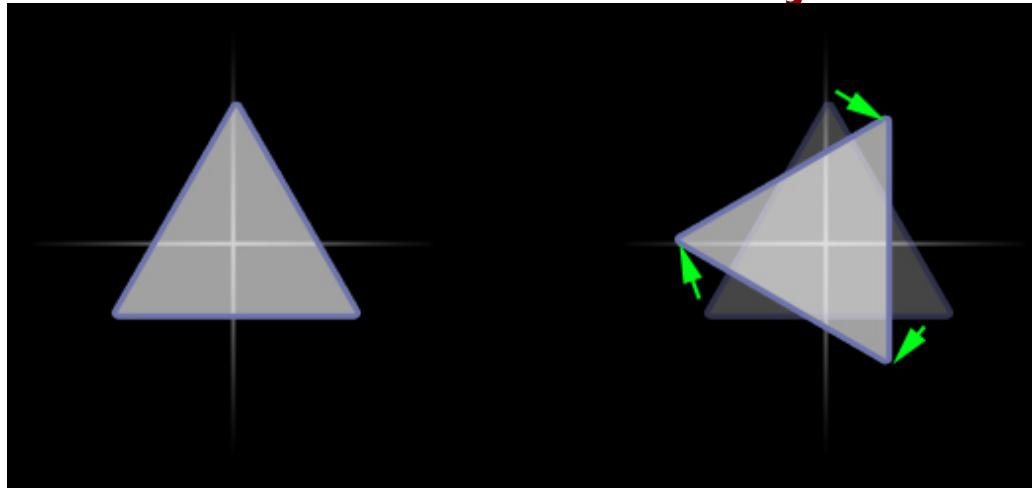


Para realizar el escalado aplicamos un factor a cada componente de cada punto:

$$P'(x',y') = P(x * fx, y * fy)$$

Por lo general cuando escalamos un objeto solemos aplicar el mismo factor a ambas componentes, de lo contrario estariamos deformando el objeto en una dimensión.

# Rotación: cambiar la orientación de un objeto



Para rotar un punto un ángulo alpha respecto a su origen utilizamos trigonometría:

$$P'(x',y') = P(x * \cos(\alpha) - y * \sin(\alpha), x * \sin(\alpha) + y * \cos(\alpha))$$

Notese que el centro de rotación es el origen de coordenadas.

# El centro o pivote

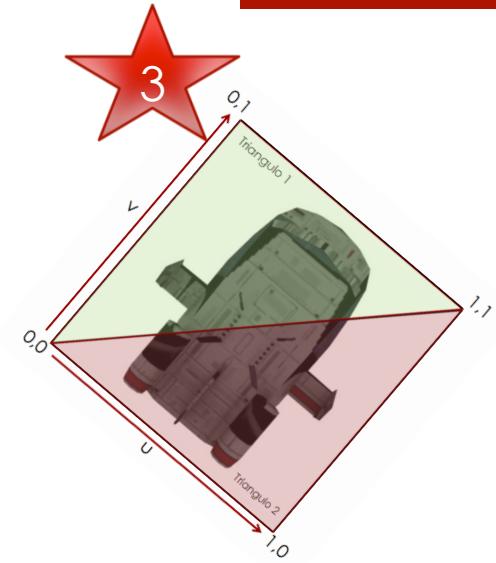
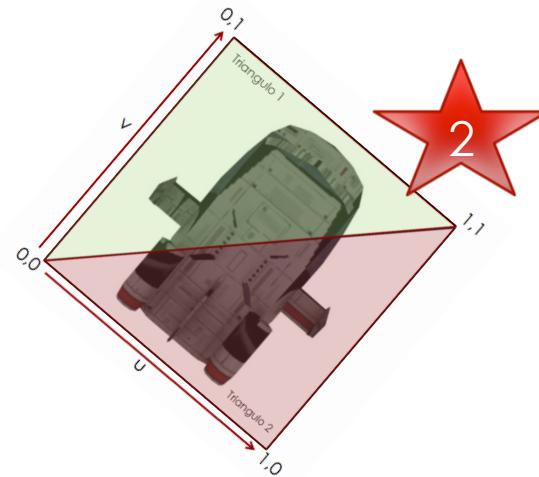
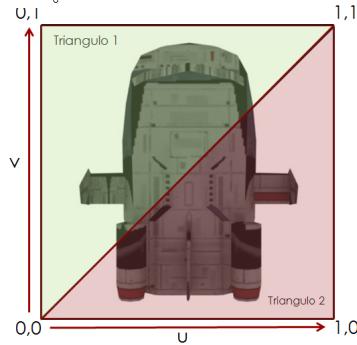
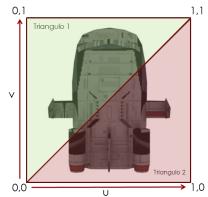
- Tanto las rotaciones como los escalados se aplican con respecto al **centro del sistema de coordenadas** (pivote).
- Es importante que el pivote esté en un punto centrado del objeto.

# Agrupar las transformaciones

- 1. Escalar
- 2. Rotar
- 3. Trasladar



3 calculaciones vectoriales  
\*  
4 vertices  
= 12 calculaciones



3

1

2

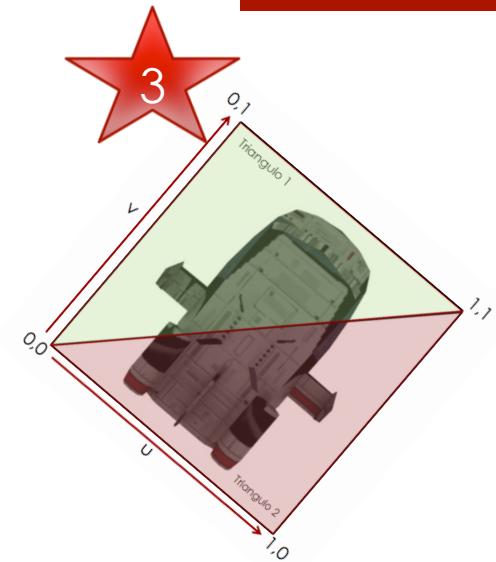
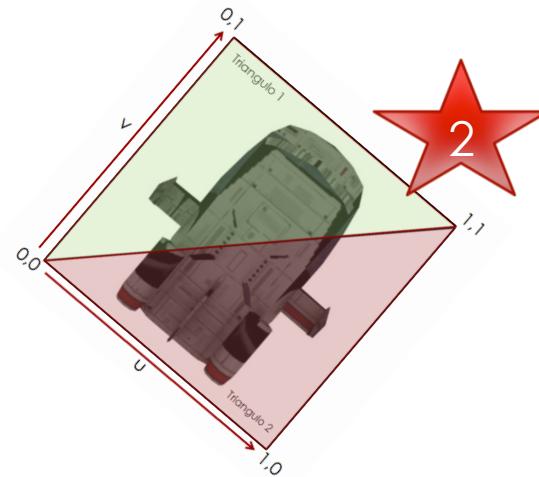
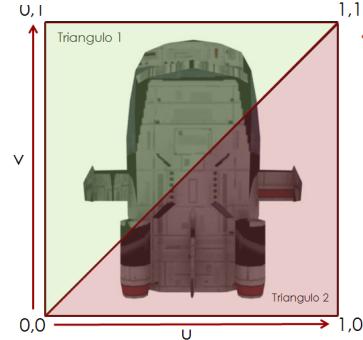
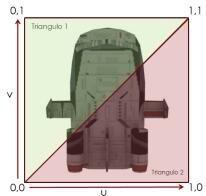
# Agrupar las transformaciones

1. Escalar
2. Rotar
3. Trasladar



3 calculaciones vectoriales  
\*  
400 vértices  
 $= 1200 \text{ calculaciones}$

Imaginemos una  
malla con 400  
puntos



# Matrices

- Utilizando multiplicacion por matrices podemos modelar cada transformacion como una matriz.
- Si trabajamos en 2D usaremos matrices de 3x3 y si trabajamos en 3D usaremos matrices de 4x4
- Trabajar con matrices tiene muchas ventajas, como por ejemplo **usar una operación común para todas las transformaciones.**

# Ejemplo de calculación

$$\begin{vmatrix} x' \\ y' \\ 1 \end{vmatrix} = \begin{vmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{vmatrix} \cdot \begin{vmatrix} x \\ y \\ 1 \end{vmatrix} \quad P' = T(d_x, d_y) \cdot P$$

La matriz T representa la matriz traslación, al multiplicarla por el vector P obtenemos el vector P' trasladado.

Es importante notar que al trabajar en 2D el vector solo tiene dos coordenadas, por ello necesitamos añadir un 1 como tercera componente para que el calculo sea correcto.

$$\begin{vmatrix} x' \\ y' \\ 1 \end{vmatrix} = \begin{vmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{vmatrix} \cdot \begin{vmatrix} x \\ y \\ 1 \end{vmatrix} \quad P' = S(s_x, s_y) \cdot P$$

$$\begin{vmatrix} x' \\ y' \\ 1 \end{vmatrix} = \begin{vmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{vmatrix} \cdot \begin{vmatrix} x \\ y \\ 1 \end{vmatrix} \quad P' = R(\alpha) \cdot P$$

En 3D las rotaciones son ligeramente más complejas ya que además de especificar un angulo tenemos que indicar el eje de rotación a utilizar. Los valores del eje se integran en diferentes valores de la matriz



# ¿Por qué usar matrices?



Cada escena 3D puede tener **decenas de miles** de vértices.

Debemos hacer todo posible **para reducir la cantidad de calculaciones** para cada vértice

Usando matrices podemos reducir drásticamente la cantidad de operaciones por **composición de matrices**

# Composición de transformaciones

Podemos **multiplicar las matrices** entre ellas para acumular diferentes transformaciones.

$$M = S * R * T$$

Luego solo tenemos que multiplicar los vectores de los vértices de nuestra escena **una vez**, por la matriz **M**

# Agrupar matrices

1. Escalar
2. Rotar
3. Trasladar



**M**

3 calculaciones

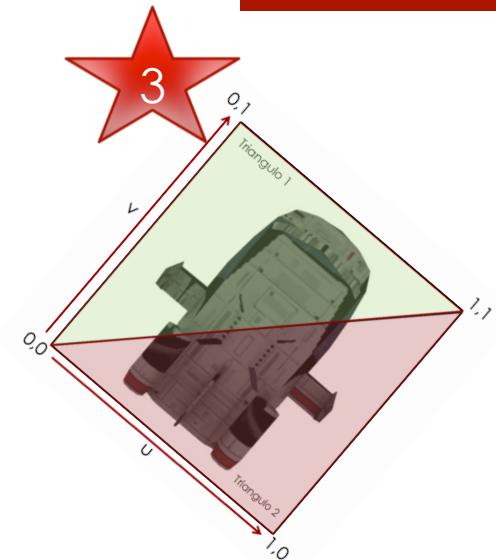
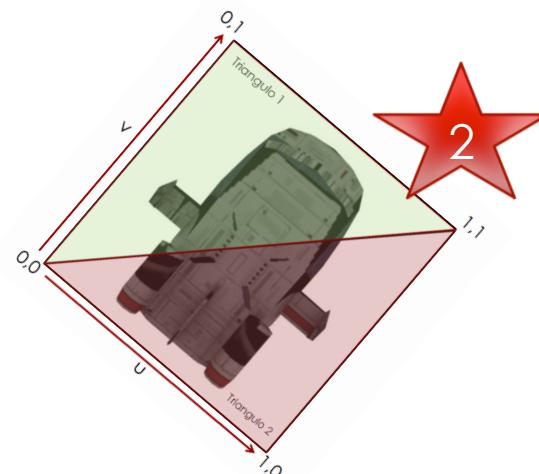
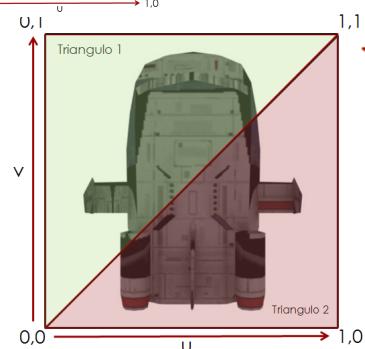
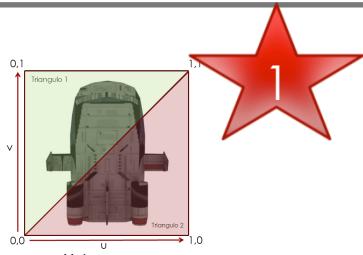
=

**M**  
\*

400 vértices

$$= 3 + 400 \text{ calculaciones}$$

$$= 403 \text{ calculaciones}$$



403 << 1200

Y una escena típica puede tener **decenas de miles** de vértices

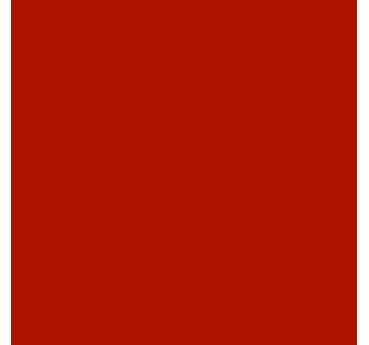
# Los 3 Matrices básicas de graficos 3D

- **Model** – Especifica posición, rotación y escalado.  
Existe una matriz model para cada objeto en la escena.
- **View** – Una matriz que determina la posición de la camera en nuestra escena
- **Projection** – Una matriz que define la relación entre la escena 3D y la pantalla 2D

angulo 1

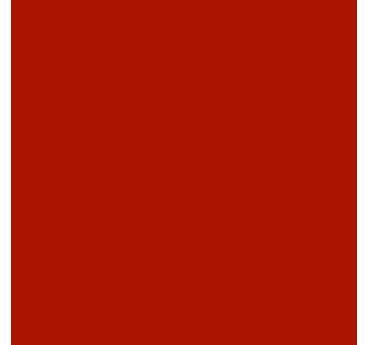


Posicionando  
nuestro sprite



```
- (GLKMatrix4) modelMatrix {  
    GLKMatrix4 modelMatrix = GLKMatrix4Identity;  
  
    modelMatrix = GLKMatrix4Translate(modelMatrix, self.position.x, self.position.y, 0);  
  
    return modelMatrix;  
}
```

// añadir en metodo render – aquí movemos nuestro objeto  
self.effect.transform.modelviewMatrix = self.modelMatrix;



```
- (GLKMatrix4) modelMatrix {  
    GLKMatrix4 modelMatrix = GLKMatrix4Identity;  
  
    modelMatrix = GLKMatrix4Translate(modelMatrix, self.position.x, self.position.y, 0);  
  
    return modelMatrix;  
}
```

En el ejemplo esta actualizado para  
posicionar el centro del sprite

// añadir en metodo render – aquí movemos nuestro objeto

```
self.effect.transform.modelviewMatrix = self.modelMatrix;
```

# *El Update Loop*

```
bool game_is_running = true;  
  
while( game_is_running ) {  
    update_game();  
    display_game();  
}
```

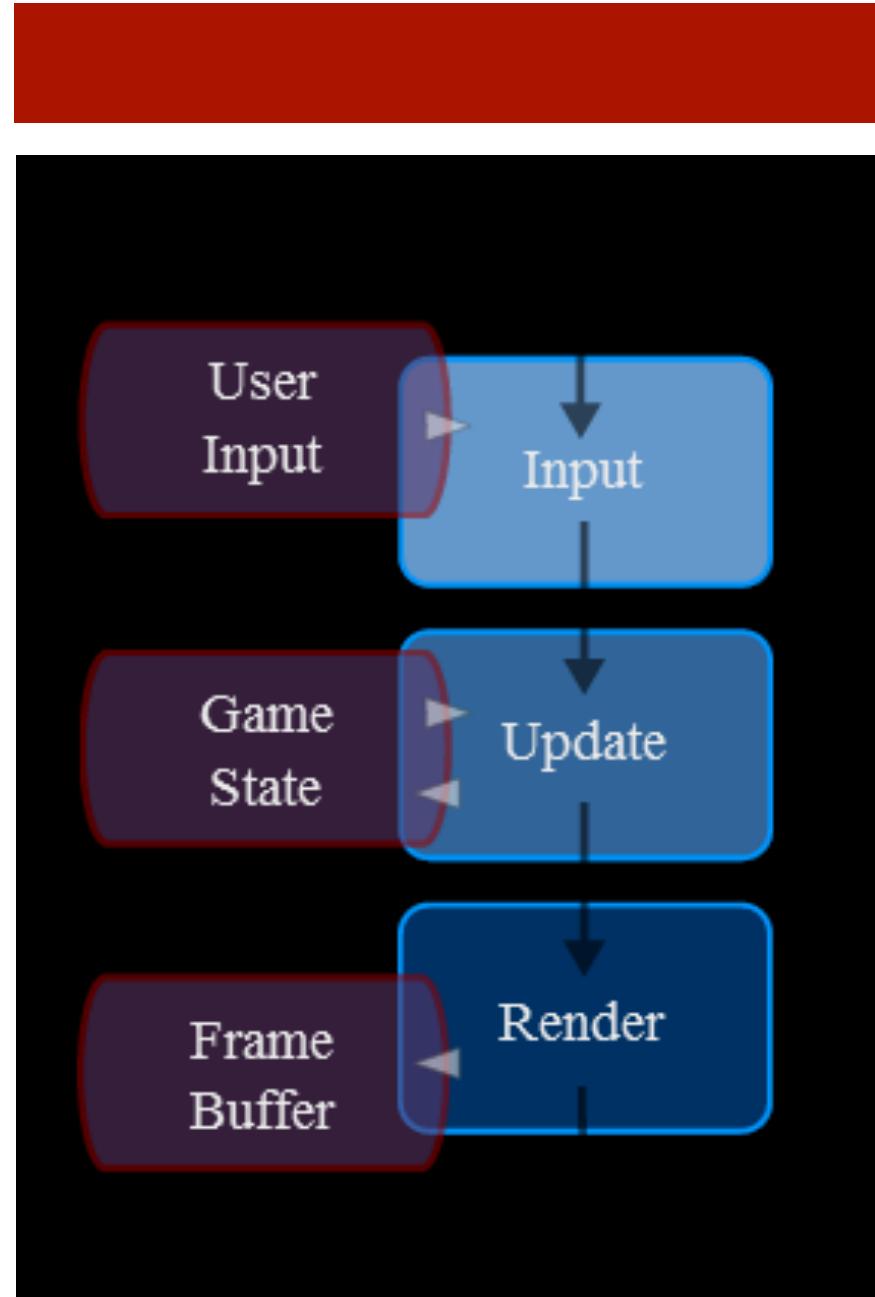
# Update loop

Se trata de unos de los elementos mas importantes de aplicaciones tiempo real como Juegos

Es un bucle que trata de actualizar constantemente cada componente en nuestro app. Por ejemplo:

- Posición de objetos
- Detección de colisiones
- Inteligencia artificial
- Sonido
- Conexión a redes

**NO actualiza los gráficos directamente!**



# El bucle mas sencillo

```
bool app_is_running = true;  
  
while( app_is_running ) {  
    update_game();  
    display_game();  
}
```

- Cada bucle de juego hace dos cosas:
  - Actualizar el juego
  - Renderizar los gráficos
  
- El problema con este ejemplo es que depende del hardware. Va rápido en hardware rápido, y lento en hardware lento

# $\Delta t$ - Tiempos de actualización

## Variable

```
while( game_is_running ) {  
    prev_frame_tick = curr_frame_tick;  
    curr_frame_tick = GetTickCount();  
    update( curr_frame_tick - prev_frame_tick );  
    render();  
}
```

Pro – Suave, facil de escribir y entender

Con – Comportamiento extraño con tiempos de update irregulares

## Fijo

```
while( game_is_running ) {  
    while( GetTickCount() > next_game_tick ) {  
        update();  
        next_game_tick += SKIP_TICKS;  
    }  
    interpolation = float( GetTickCount() + SKIP_TICKS -  
    next_game_tick ) / float( SKIP_TICKS );  
    render( interpolation );  
}
```

Pro – Fisica previsible. Mas fácil sincronizar a través de la red. El  $\Delta t$  siempre es regular.

Con – Muy difícil sincronizar con gráficos.

# GLKit to the Rescue - Again

- El GLKViewController nos configura un bucle update variable, que podemos configurar para simular un bucle fijo por limitar el framerate

# GLKit to the Rescue - Again

- El GLKViewController nos configura un bucle update variable, que podemos configurar para simular un bucle fijo por limitar el framerate

```
- (void)update {  
    [sprite update:self.timeSinceLastUpdate];  
}
```

El GLKViewController tiene un metodo update que sobreescribimos

pasamos el  $\Delta t$  a cada objeto en nuestra escena

# Detector Input

# GLKViewController es subclase de UIViewController

Por eso todavia podemos acceder a metodos de interaccion como:

- touchesBegan
- touchesMoved
- touchesEnded

## GLKViewController Class Reference

Inherits from	UIViewController : UIResponder : NSObject
Conforms to	NSCoding GLKViewDelegate NSCoding (UIViewController) UIAppearanceContainer (UIViewController) NSObject (NSObject)
Framework	/System/Library/Frameworks/GLKit.framework
Availability	Available in iOS 5.0 and later.
Declared in	GLKViewController.h

# Cambiar posición de un sprite según un toque

- Algoritmo:
  - Detectar toque y guardar posición
  - Crear vector entre:
    - Posición sprite
    - Posición toque
  - En update, mover el sprite
    - Crear Vector entre sprite y posición toque
    - Multiplicar matriz de traslación con ese vector
  - Si detectamos que el sprite esta en la 'zona muerte', paramos

# Detector toque

```
//Mover el sprite cuando el usuario toca la pantalla  
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {  
    UITouch * touch = [touches anyObject];  
    self.lastTouch = [touch locationInView:self.view];  
    self.touching = YES;  
}
```

# Mover sprite

```
-(void)movePlayer {  
}
```

En movePlayer:

- Detectar si el usuario esta tocando la pantalla
- Definir ‘zona muerte’ alrededor del sprite
- Calcular Vector entre Sprite y dedo
- Guardar ese Vector como property del sprite

# Update del sprite

```
// Metodo update del sprite
- (void)update:(float)dt {
    // deducir la distancia movido en el tiempo desde ultimo update
    GLKVector2 curMove = GLKVector2MultiplyScalar(
        self.moveVelocity, dt);

    self.position = GLKVector2Add(self.position, curMove);
}
```

# Update del sprite

```
// Metodo update del sprite  
- (void)update:(float)dt {  
    // deducir la distancia movido en el tiempo desde ultimo update  
    GLKVector2 curMove = GLKVector2MultiplyScalar(  
        self.moveVelocity, dt);  
  
    self.position = GLKVector2Add(self.position, curMove);  
}
```

## ¿Por qué no usar matrices aquí?

Porque solo estamos multiplicando dos vectores una vez.

En el metodo render, sí que multiplicamos la posición por el Model Matrix

# Update del juego

```
//main game loop
- (void)update {

    [self movePlayer];

    //para cada sprite en la escena, llamar al
    update

    for (idECSprite * sprite in self.allSprites) {
        [sprite update:self.timeSinceLastUpdate];
    }
}
```



# Tiler

- Queremos dar el efecto que nuestro astronave esta navegando entre las estrellas.
- Para hacer esto, creamos un nuevo objeto 'Tiler' para hacer un scroll constante de una textura

# Texturas de fondo

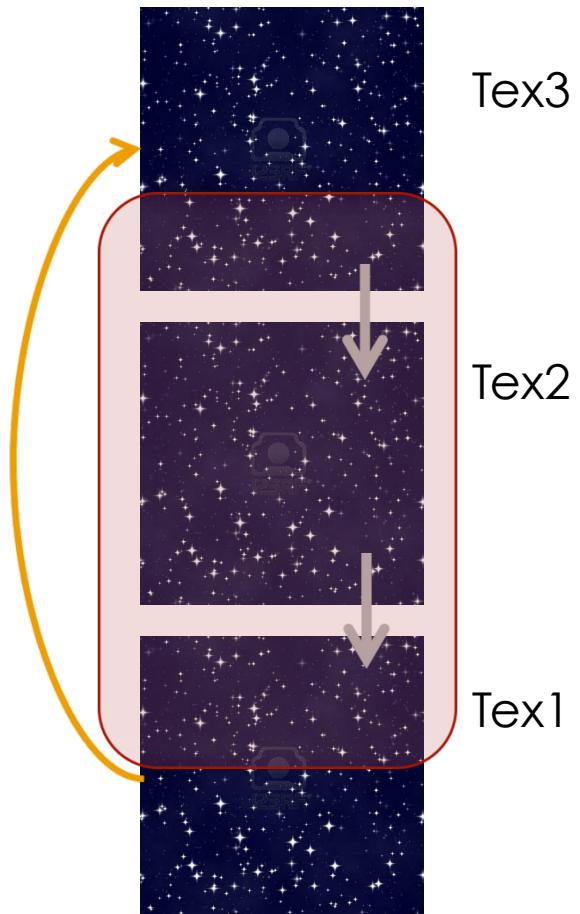
En otras plataformas podemos usar una textura sola, muy muy larga (como 6000px), y decir que esto es nuestro “nivel”



Pero con OpenGL ES2, el tamaño máximo de una textura es 2048x2048

# Concepto del tiler 1D

- Tenemos tres sprites con la misma imagen, cada uno esta moviendo hacia abajo
- Cuando detectamos que una textura ya no esta en la pantalla, la movemos hacia arriba
- Asi tenemos un efecto espacio infinito



# Update del juego para v0.2

```
//main game loop
- (void)update {

    [self movePlayer];

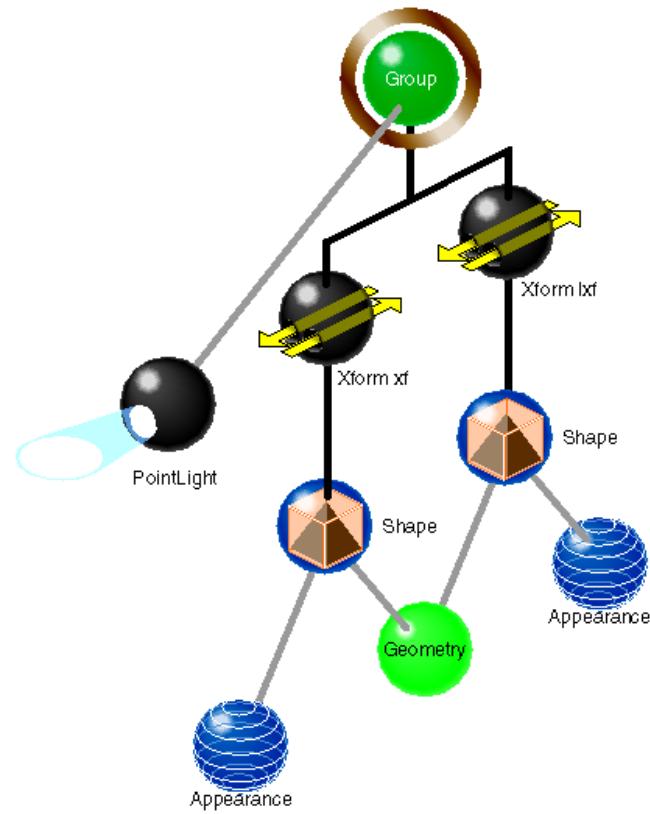
    //para cada sprite en la escena, llamar al update
    for (idECSprite * sprite in self.allSprites) {
        [sprite update:self.timeSinceLastUpdate];
    }

    //actualizar el fondo
    [self.tiler update:self.timeSinceLastUpdate];
}
```



# Game v0.3

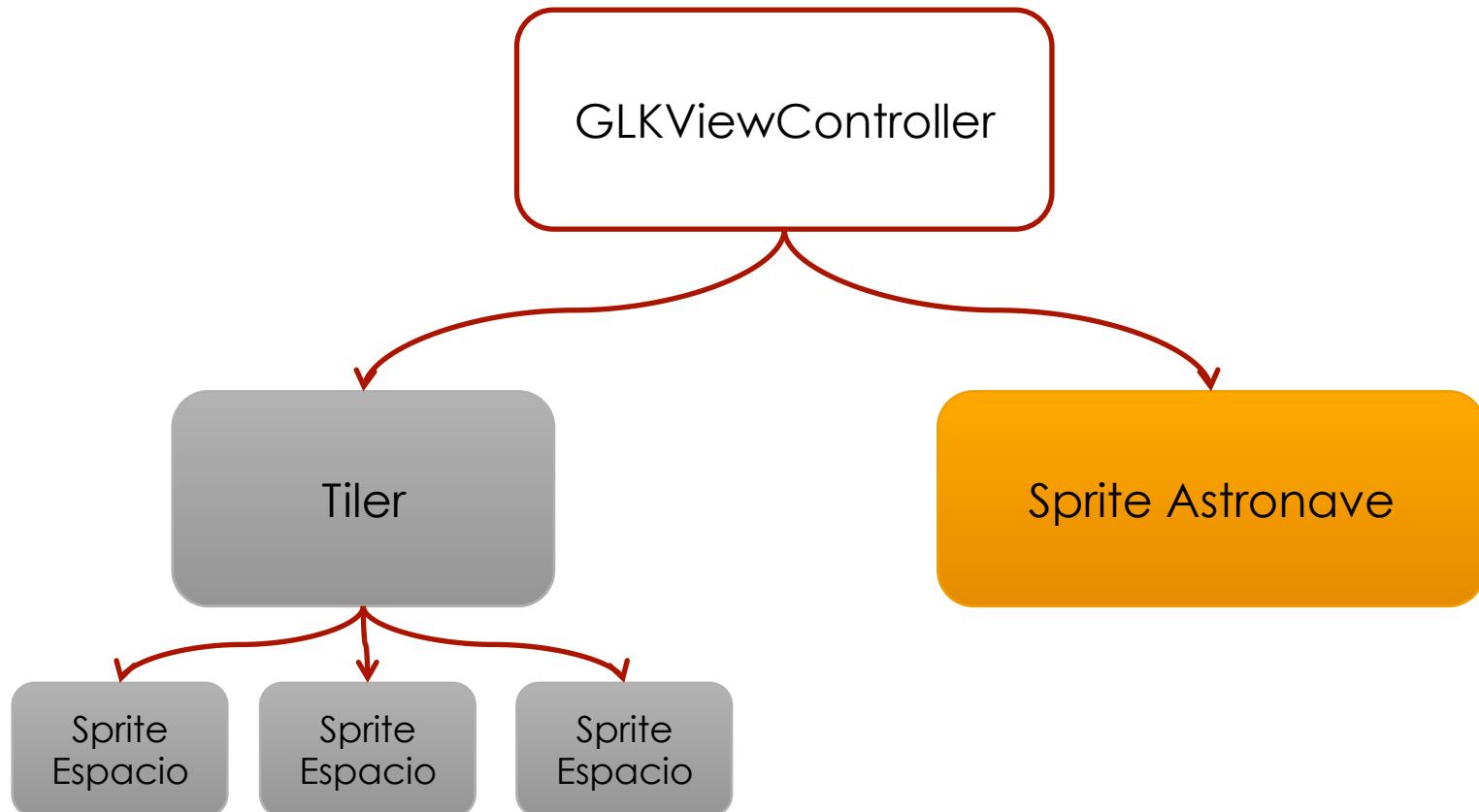
- Exactamente lo mismo
- Pero con el Scene Graph



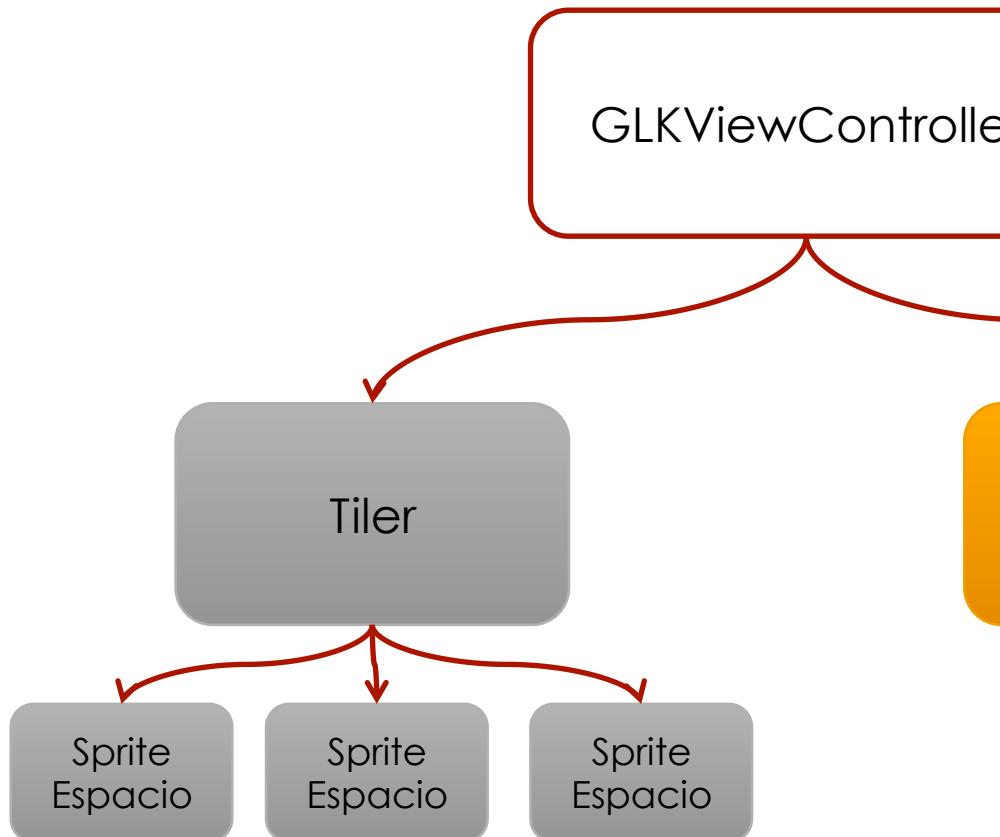
# Scene Graph

Ordenar nuestra aplicación

# Ahora tenemos...



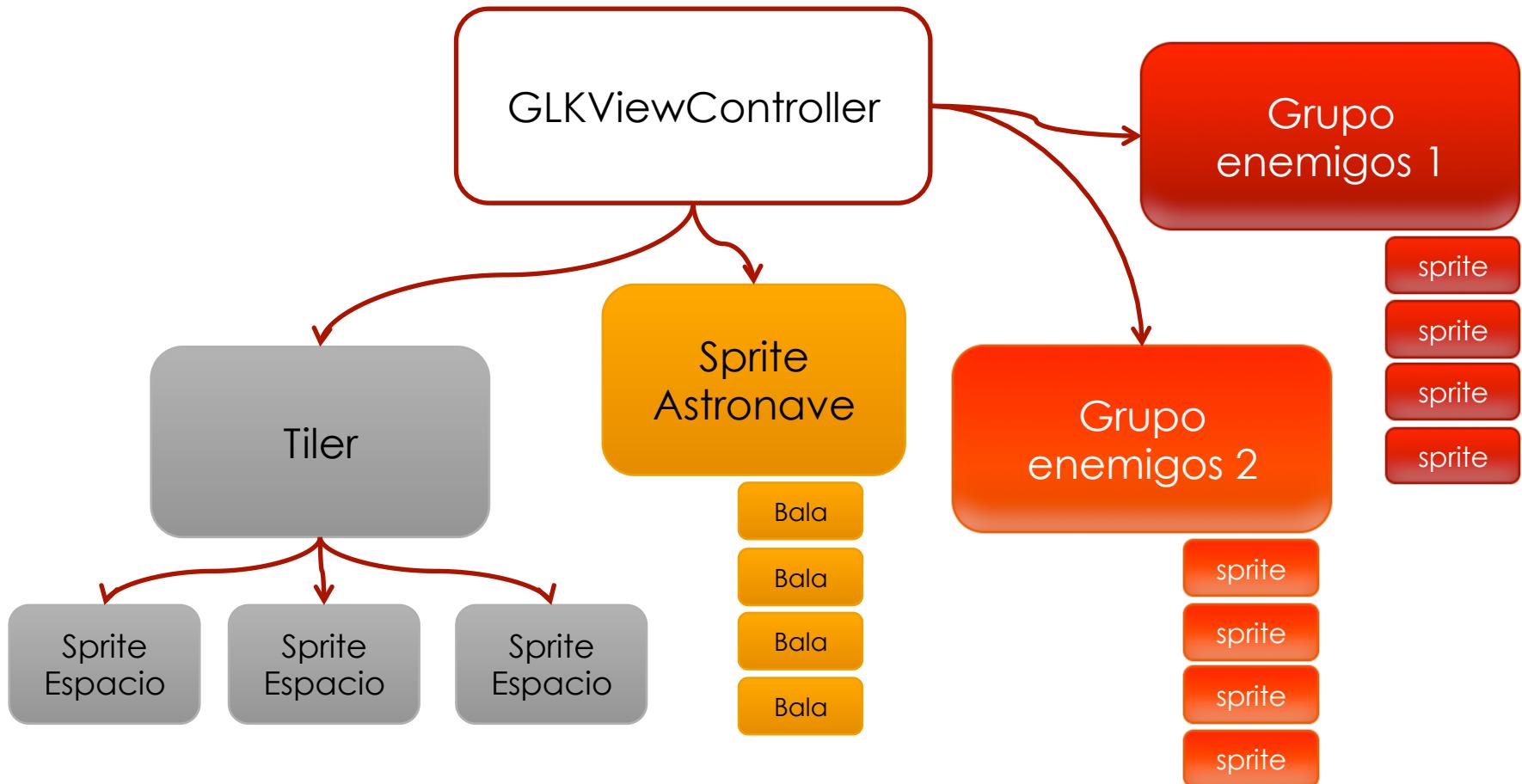
# Ahora tenemos...



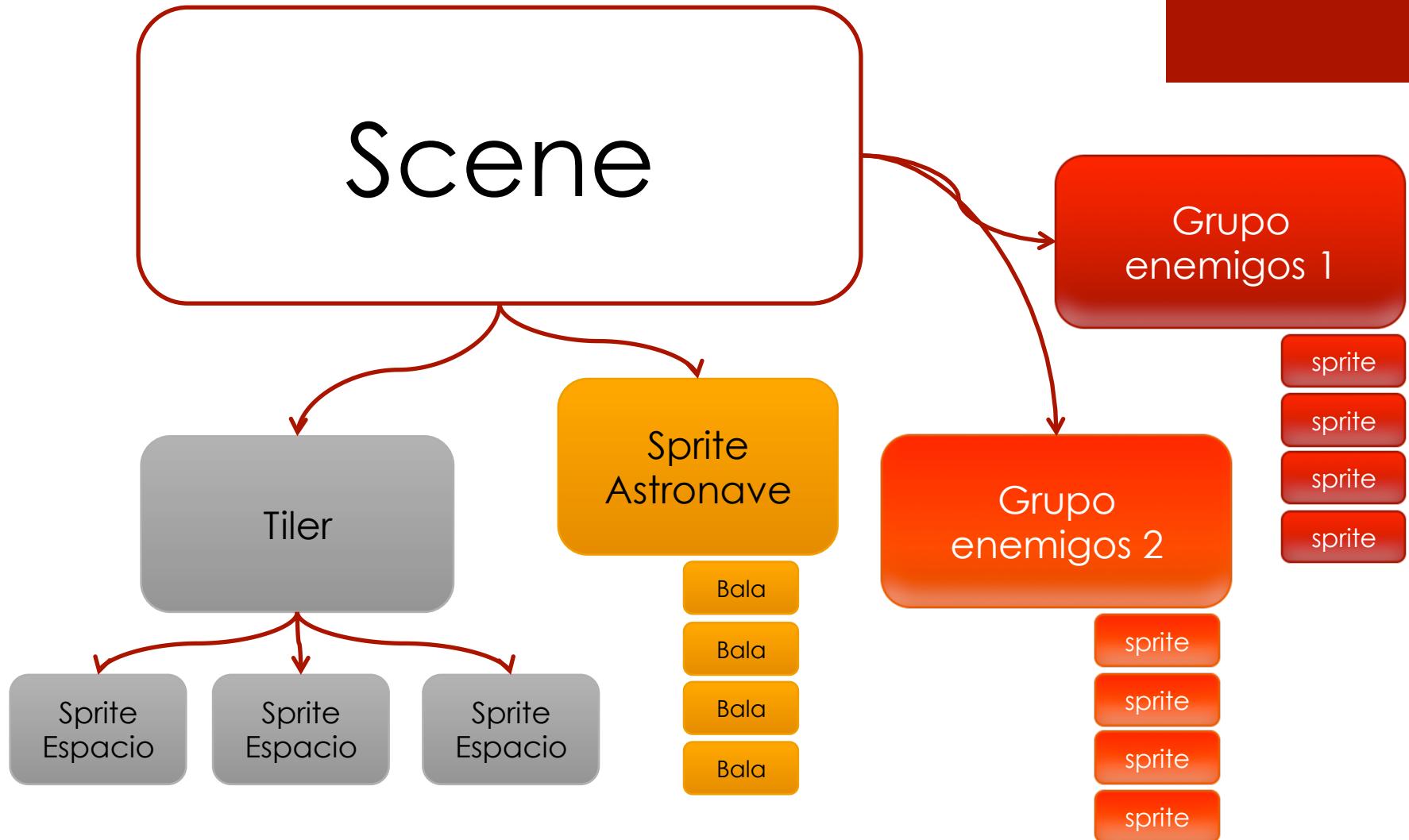
Pero, que pasa cuando añadimos:

- balas
- enemigos
- grupos de enemigos
- Power-ups
- Estaciones espaciales
- Monstruos de la dimension Z
- etc. etc.

# Podriamos tener:



# Formalizamos:



# El Scene Graph

- Un **estructura de datos** que organiza la representación lógica y espacial de una escena gráfica
- Se usa en la mayoría de aplicaciones que usan gráficos 3D (Adobe Illustrator, AutoCAD, Maya etc) y casi cada juego

Se puede considerar el ‘salto’ a usar el Scene Graph como el momento en que nuestro app de graficos se convierte a algo ‘serio’!

# Clase Node – la columna vertebral del scene graph

- Definimos una clase se llama **Node** (o bien *Entity*) que es la clase padre de cada elemento del Scene Graph
- Cualquier elemento (Sprite, malla, tiler, etc.) en nuestro escena lo **definimos como una sub-clase** de Node
- La clase Node tambien nos sirve para mas cosas. P.ej. **no toda entidad de la escena tiene una representación visual** (por ejemplo, un foco de luz, un emisor de sonido, un volumen 3D que activa un evento cuando lo cruzas, etc). Pero igual puede ser nodos en nuestra escena

# Clase Node

```
@interface Node : NSObject

@property (assign) GLKVector2 position;
@property (assign) CGSize contentSize;
@property (assign) GLKVector2 moveVelocity;
@property (retain) NSMutableArray * children;
@property (assign) float rotation;
@property (assign) float scale;
@property (assign) float rotationVelocity;
@property (assign) float scaleVelocity;

- (void)renderWithModelViewMatrix:(GLKMatrix4)modelViewMatrix;
- (void)update:(float)dt;
- (GLKMatrix4) modelMatrix:(BOOL)renderingSelf;
- (CGRect)boundingBox;
- (void)addChild:(Node *)child;
- (void)handleTouchDown:(CGPoint)touchLocation;
- (void)handleTouchMoved:(CGPoint)touchLocation;
- (void)handleTouchUp:(CGPoint)touchLocation;

@end
```

# Clase Node

```
@interface Node : NSObject

@property (assign) GLKVector2 position;
@property (assign) CGSize contentSize;
@property (assign) GLKVector2 moveVelocity;
@property (retain) NSMutableArray * children;
@property (assign) float rotation;
@property (assign) float scale;
@property (assign) float rotationVelocity;
@property (assign) float scaleVelocity;

- (void)renderWithModelViewMatrix:(GLKMatrix4)modelViewMatrix;
- (void)update:(float)dt;
- (GLKMatrix4) modelMatrix:(BOOL)renderingSelf;
- (CGRect)boundingBox;
- (void)addChild:(Node *)child;
- (void)handleTouchDown:(CGPoint)touchLocation;
- (void)handleTouchMoved:(CGPoint)touchLocation;
- (void)handleTouchUp:(CGPoint)touchLocation;

@end
```

Cada Node tiene las variables básicas que necesitamos, y un array de hijos

# Clase Node

```
@interface Node : NSObject

@property (assign) GLKVector2 position;
@property (assign) CGSize contentSize;
@property (assign) GLKVector2 moveVelocity;
@property (retain) NSMutableArray * children;
@property (assign) float rotation;
@property (assign) float scale;
@property (assign) float rotationVelocity;
@property (assign) float scaleVelocity;

- (void)renderWithModelViewMatrix:(GLKMatrix4)modelViewMatrix;
- (void)update:(float)dt;
- (GLKMatrix4) modelMatrix:(BOOL)renderingSelf;
- (CGRect)boundingBox;
- (void)addChild:(Node *)child;
- (void)handleTouchDown:(CGPoint)touchLocation;
- (void)handleTouchMoved:(CGPoint)touchLocation;
- (void)handleTouchUp:(CGPoint)touchLocation;

@end
```

El metodo render recibe una matriz del padre.  
Esto significa que

**los hijos de un Node se renderizan respecto al Padre**

# Clase Node

```
@interface Node : NSObject

@property (assign) GLKVector2 position;
@property (assign) CGSize contentSize;
@property (assign) GLKVector2 moveVelocity;
@property (retain) NSMutableArray * children;
@property (assign) float rotation;
@property (assign) float scale;
@property (assign) float rotationVelocity;
@property (assign) float scaleVelocity;

- (void)renderWithModelViewMatrix:(GLKMatrix4)modelViewMatrix;
- (void)update:(float)dt;
- (GLKMatrix4) modelMatrix:(BOOL)renderingSelf;
- (CGRect)boundingBox;
- (void)addChild:(Node *)child;
- (void)handleTouchDown:(CGPoint)touchLocation;
- (void)handleTouchMoved:(CGPoint)touchLocation;
- (void)handleTouchUp:(CGPoint)touchLocation;

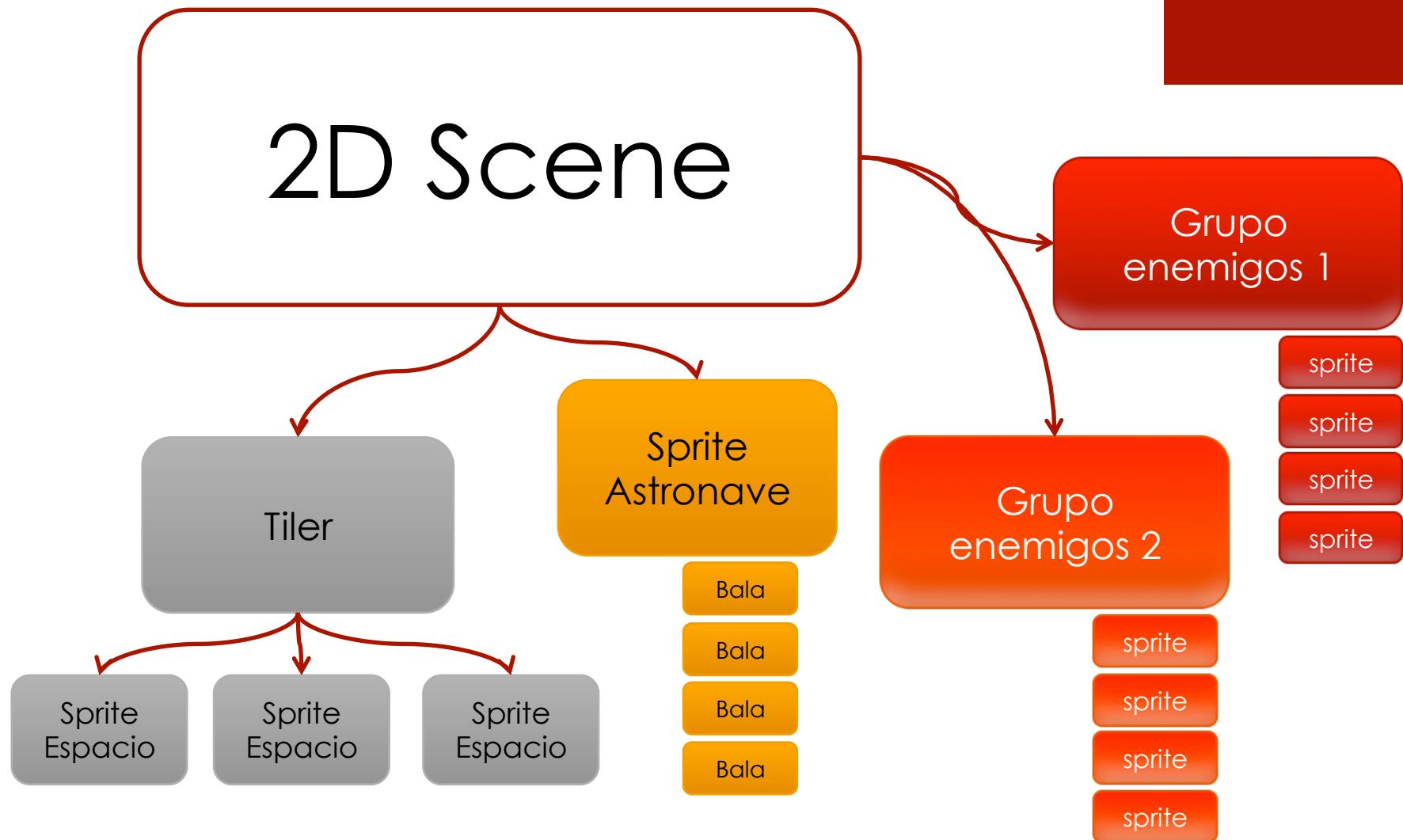
@end
```

Aquí definimos uno  
métodos útiles para  
nuestro app.

# Un poco de caos

- Aunque el concepto del Scene Graph esta muy bien definido, la realidad es que no hay ninguna definición clara.
- Cada programador tiene su version de lo que se necesita en el Scene Graph, y como se ordena (p. ej. se puede añadir mas sub-clase *NodeMesh*, *NodeLight*, *NodeCamera* etc,
- Incluso un Nodo puede representar un sonido, un *Trigger* (zona que provoca algun evento en el juego)

# Diferentes escenas



# Diferentes escenas

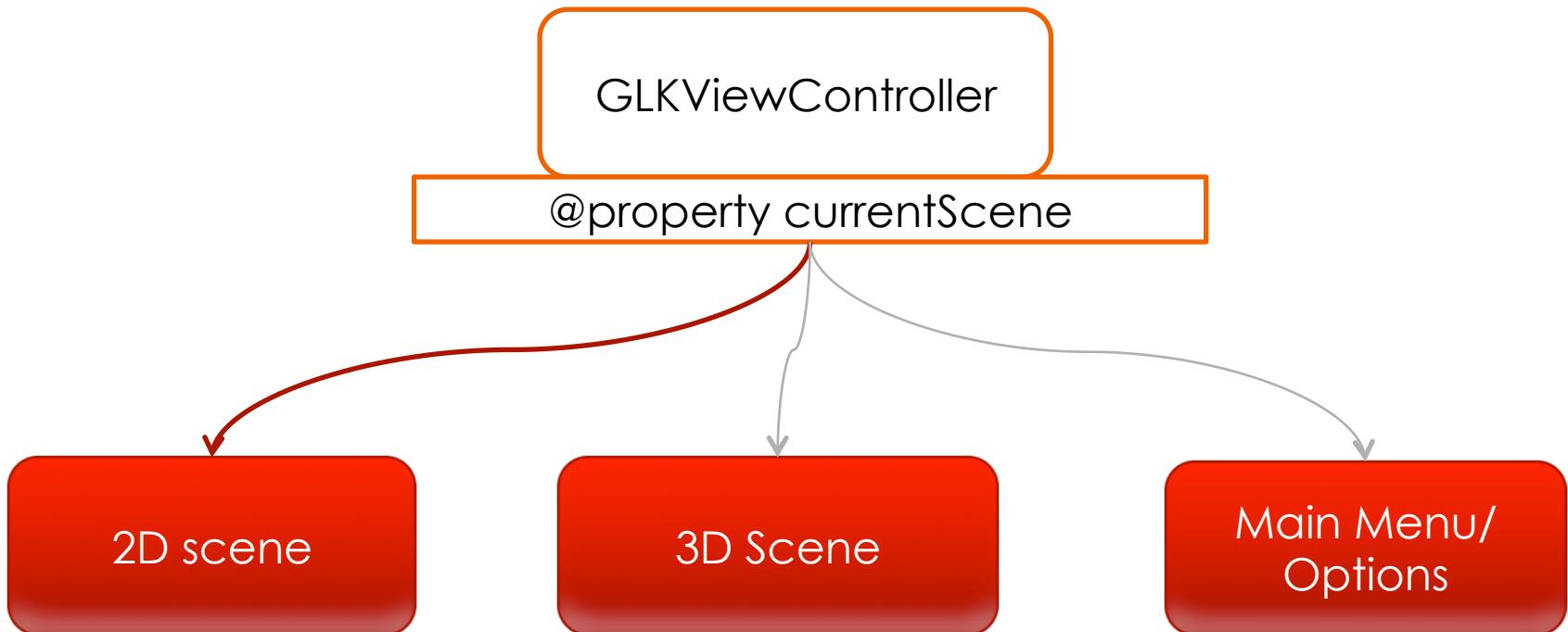
GLKViewController

2D scene

3D Scene

Main Menu/  
Options

# Diferentes escenas



# Game v0.4

- Bullets
- Enemies
- Collisions
- fps counter

