Índice general

Índice general **1**

1. *Rendering* en iOS: ISGL3D **2**
1.1. Introducción 2
1.2. Conceptos básicos de ISGL3D 3
1.3. FOV y ejes de ISGL3D 4
1.4. Primitivas de ISGL3D 4
1.5. Importación de modelos a ISGL3D. 7
1.6. Luz en ISGL3D 10
1.7. Método - *(void) tick:(float)dt* 10
1.8. ISGL3D en la aplicación 10
1.9. Conclusión 11

Bibliografía **12**

CAPÍTULO 1

Rendering en iOS: ISGL3D

1.1. Introducción

Rendering es un término en inglés que denota el proceso de generar una imagen 2D a partir de un modelo digital 3D o un conjunto de ellos, a los que se les llama "escena". Puede ser comparado con tomar una foto o filmar una escena en la vida real. Afortunadamente, existen varias herramientas de *rendering*, también llamadas "motores de juego 3D", para plataformas móviles, en especial que funcionen sobre iOS. Algunas de ellas son Unity 3D, ISGL3D, Cocos3D, Open GL ES y ShiVa3D. A continuación serán comentadas tan sólo las consideradas durante el presente proyecto por ser populares y gratuitas.

La primera en ser tomada en cuenta fue "Open Graphics Library Embedded Systems" (Open GL ES), que es un subconjunto de las herramientas de gráficos 3D de Open GL. Fue diseñada para ser utilizada sobre sistemas embebidos (dispositivos móviles, consolas de video juegos, etc.); Open GL es el estándar más ampliamente usado alrededor del mundo para la creación de gráficos 2D y 3D, es gratis y multiplataforma. Como la programación en Open GL y en particular en Open GL ES es de muy bajo nivel y por lo tanto bastante complicada, se optó por investigar otras herramientas. Se descubrió entonces ISGL3D un *framework* escrito en Objective-C que trabaja sobre Open GL ES y que busca facilitar la tarea del programador al momento de crear y manipular escenas 3D mediante una "*Application Program Interface*" (API) sencilla e intuitiva. Es un proyecto gratis y en código abierto. Luego de algunas semanas de trabajo con la herramienta e importantes avances desde el punto de vista del manejo de la misma se descubrió la existencia de otro *framework* de idénticas características llamado "Cocos3D". Cocos3D es una extensión de "Cocos2D", una herramienta para la generación de gráficos 2D, muy popular entre los desarrolladores de aplicaciones para iOS. Como no se identificaron diferencias significativas entre ISGL3D y Cocos3D, se priorizó el tiempo dedicado a ISGL3D y se decidió continuar trabajando de forma inalterada. Al día de hoy, sobre el final del proyecto, se cree que si bien técnicamente ambos *frameworks* son muy buenos y a la vez similares entre sí, ISGL3D parece estar algo más avanzado en cuanto a su desarrollo. Sin embargo debido a la gran popularidad de Cocos2D, Cocos3D ha heredado muchos usuarios y cuenta con una comunidad mucho más activa, lo que facilita mucho su uso y hace pensar que en un futuro cercano resulte en un *framework* más desarrollado.

En este capítulo se comentarán algunas características y conceptos de ISGL3D que fueron importantes para el proyecto; por detalles de algunos temas en particular referirse a la referencia de la ya mencionada API en: `www.isgl3d.com/resources/api`. Además se trazará una hoja de ruta para todo aquel que quiera iniciarse en el manejo de la herramienta.

1.2. Conceptos básicos de ISGL3D

ISGL3D es un motor de juegos 3D para *iPad*, *iPhone* y *iPod touch* escrito en *Objective-C*, que sirve para crear escenas y *renderizarlas* de forma sencilla. Es un proyecto en código abierto y gratis. En su sitio web oficial: `www.isgl3d.com`, se puede descargar su código, y de forma sencilla ISGL3D puede ser agregado como un complemento de *Xcode*. Además se pueden encontrar tutoriales, detalles de su API y un acceso a un grupo de *Google* donde la comunidad pregunta y responde dudas propias y ajenas. Una buena manera de iniciarse en manejo de la herramienta es siguiendo los tutoriales en: `www.isgl3d.com/resources/tutorials`; al menos este fue el camino elegido por el grupo. Los tutoriales son 6, y abarcan distintos tópicos:

- **Tutorial 0:** primer paso en el creado de una aplicación ISGL3D. Cubre algunos conceptos básicos y muestra cómo integrar la herramienta a *Xcode*.
- **Tutorial 1:** mustra cómo crear una escena bien simple, con tan sólo un cubo en rotación continua.
- **Tutorial 2:** enseña cómo agregar luz a una escena. Se ven las distintas fuentes de luz que existen en el *framework*.
- **Tutorial 3:** se ve cómo hacer para mapear texturas en los objetos 3D con el objetivo de hacerlos más realistas.
- **Tutorial 4:** muestra cómo crear interacción entre el usuario y los distintos objetos ISGL3D, cuando este los toca a través de la pantalla.
- **Tutorial 5:** se ven algunas nuevas primitivas (modelos básicos) y se muestra cómo agregar transparencia a los objetos.

Al descargar e instalar ISGL3D, se puede ver que la herramienta incluye un proyecto *Xcode* integrado por varios ejemplos para ejecutar y a la vez ver su código, otra buena forma de aprender cómo realizar distintas tareas de interés. Entre los ejemplos se encuentra la solución a cada uno de los tutoriales.

Cuando se crea una aplicación ISGL3D, el núcleo de la misma es la llamada "*view*" ("vista" en Español). Una *view* está compuesta principalmente por una escena y una cámara:

- Una **escena** (*Isgl3dScene3D*) es donde los objetos o modelos 3D son agregados como nodos. Todos los nodos pueden ser tanto trasladados como rotados y pueden tener otros nodos hijos; los nodos hijos son trasladados y rotados con sus padres. Así como objetos 3D, se pueden agregar luces de distinto tipo, que generarán en la escena efectos de sombra que luego serán adecuadamente *renderizados* en función de dónde se encuentre y hacia dónde esté mirando la cámara.
- Una **cámara** que es utilizada para para ver la escena desde una posición y un ángulo en particular. La cámara se manipula como cualquier otro objeto o nodo en la escena, se puede trasladar, rotar y hasta indicar hacia dónde quiere uno que esta apunte. Es importante ajustar la cámara de manera que su arquitectura sea la que uno busca. Se pueden entonces ajustar ciertos parámetros intrínsecos a esta como por ejemplo su campo visual, su distancia focal, la altura y la anchura del plano imagen, etc.

Es importante entender que el llamado *render* se realiza sumando la información de la escena, objetos 3D y sus hijos, luces, etc.; más la información de dónde se encuentra la cámara, sus características y hacia dónde esta apunta.

1.3. FOV y ejes de ISGL3D

Una particularidad de la cámara de ISGL3D es que el parámetro intrínseco "distancia focal" visto en la sección **??**, no es directamente configurable. En cambio, el valor que sí se puede alterar es el llamado "FOV", acrónimo de "Field Of View". El *field of view* de una cámara no es más que su campo visual, y se mide como la extensión angular máxima mapeable en el plano imagen, medida desde el centro óptico O. Puede ser medido de forma horizontal o de forma vertical; sin embargo, en ISGL3D es definido verticalmente. Ver figura 8.1.

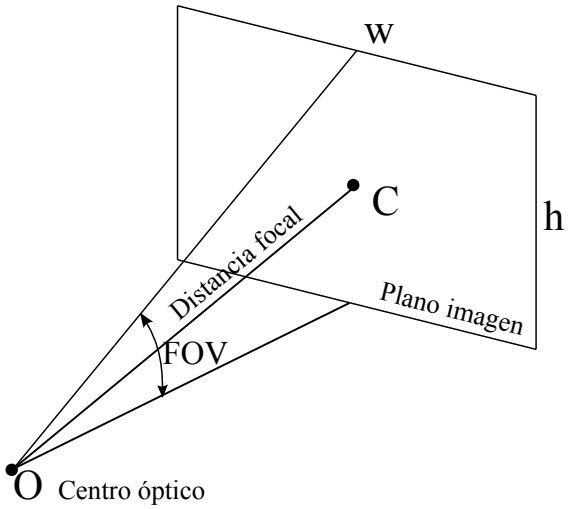

Figura 1.1: Definición gráfica del FOV.

Realizando algo de geometría se ve que la relación entre la distancia focal y el FOV es:

$$FOV = 2.arctg\left(\frac{h}{2.f}\right)$$

donde h denota la altura del plano imagen y f la distancia focal de la cámara.

Otra particularidad de ISGL3D es el sistema de coordenadas que difiere del que se usa en algunas herramientas de modelado, como por ejemplo Blender, en las que el plano X-Y se corresponde con el plano horizontal. Para ISGL3D los ejes están orientados como en la Figura 8.2, con el origen en el centro del plano de la pantalla del dispositivo que coincide con el plano X-Y, y el eje Z saliente de la misma.

1.4. Primitivas de ISGL3D

ISGL3D cuenta con algunas estructuras primitivas que pueden ser usadas como modelos, o incluso combinadas de manera de formar modelos algo más complejos. Las principales estructuras

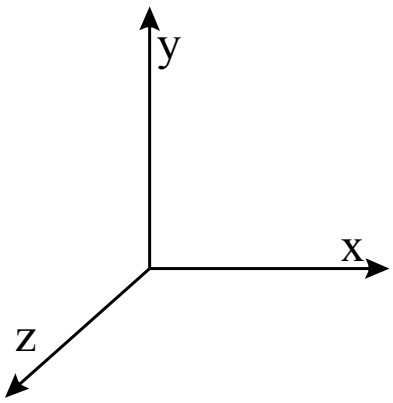

Figura 1.2: Sistema de coordenadas de ISGL3D.

primitivas de ISGL3D son:

- **Isgl3DArrow:** modelo correspondiente a una flecha. Tiene 4 parámetros configurables:
  - *headHeight:* altura de la punta.
  - *headRadius:* radio de la punta.
  - *height:* altura total de la flecha.
  - *radius:* radio de la base.
- **Isgl3DCone:** modelo correspondiente a un cono. Tiene 3 parámetros configurables:
  - *bottomRadius:* radio de la base inferior.
  - *height:* altura del cono.
  - *topRadius:* radio de la base superior.
- **Isgl3DCube:** modelo correspondiente a un cubo. Tiene 3 parámetros configurables:
  - *depth:* profundidad del cubo.
  - *height:* altura del cubo.
  - *width:* anchura del cubo.
- **Isgl3DCylinder:** modelo correspondiente a un cilindro. Tiene 3 parámetros configurables:
  - *height:* altura del cilindio.
  - *radius:* radio del cilindro.
  - *openEnded:* indica si el cilindro cuenta con sus extremos abiertos o no.
- **Isgl3DEllipsoid:** modelo correspondiente a una elipsoide. Cuenta con 3 parámetros configurables:
  - *radiusX:* radio de la elipsoide en la dirección *x*.
  - *radiusY:* radio de la elipsoide en la dirección *y*.
  - *radiusZ:* radio de la elipsoide en la dirección *z*.

- **Isgl3DOvoid:** modelo ovoide. Cuenta con 3 parámetros configurables:
  - *a:* radio del ovoide en la dirección *x*.
  - *b:* radio del ovoide en la dirección *y*.
  - *k:* factor que modifica la forma de la curva. Cuando toma el valor 0, el modelo se corresponde con el de una ellipsoide.
- **Isgl3DSphere:** modello correspondiente a una esfera. Tiene un único parámetro configurable:
  - *radius:* radio de la esfera.
- **Isgl3DTorus:** modelo correspondiente a un toroide. Cuenta con 2 parámetros configurables:
  - *radius:* radio desde el origen del toroide hasta el centro del tubo.
  - *tubeRadius:* radio del tubo del toroide.

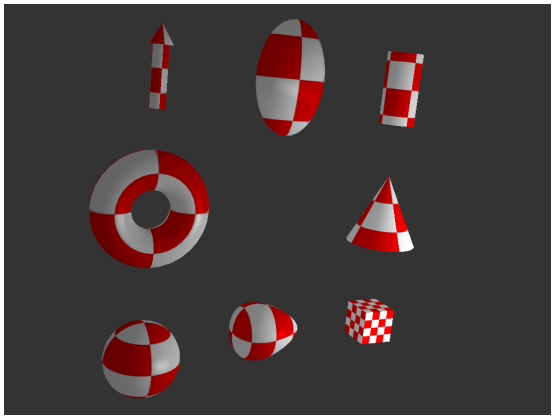

Figura 1.3: Principales primitivas en ISGL3D.

Para la creación de cada primitiva, se debe especificar además, la cantidad de segmentos que la forman en las distintas dimensiones. En la figura 8.3 se pueden ver todas las primitivas anteriores. Es fácil ver que dichas primitivas cuentan con cierta textura cuadriculada de colores rojo y blanco, que fue lograda mapeando una imagen sobre cada una de ellas. La porción de código que se usó para realizar tal mapeo se muestra a continuación:

```
Isgl3dTextureMaterial * material = [Isgl3dTextureMaterial
                  materialWithTextureFile:@"red_checker.png" shininess:0.9];

Isgl3dTorus * torusMesh = [Isgl3dTorus meshWithGeometry:2 tubeRadius:1 ns:32 nt:32];

Isgl3dMeshNode * _torus = [self.scene createNodeWithMesh:torusMesh andMaterial:material];
```

En la primera línea de código se crea el material. Dicho material es del tipo *Isgl3dTextureMaterial*; y la imagen con la que este se crea es la de la figura 8.4. Luego, se crea el toroide asignándole los parámetros vistos más atrás en esta sección; y finalmente, se crea y se agrega a la escena el nodo asociado al toroide, con el material creado al principio.

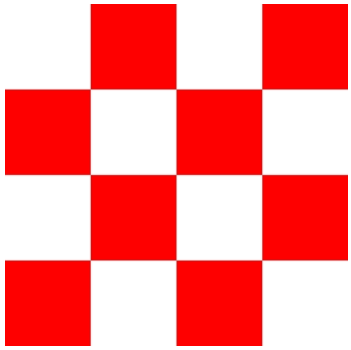

Figura 1.4: Imagen *red_checker.png*, utilizada para crear la textura asociada a las primitivas de la figura 8.3.

1.5. Importación de modelos a ISGL3D.

A veces lo que se quiere no es agregar a la escena una primitiva sino un modelo previamente creado. Los modelos son realizados en herramientas de creado y animación de gráficos 3D como por ejemplo *Blender*, *MeshLab*, *Autodesk Maya* o *Autodesk 3ds Max*. Luego deben ser exportados en un formato llamado *COLLADA*, acrónimo de "COLLAborative Design Activity", que sirve para el intercambio de contenido digital 3D entre distintas aplicaciones de modelado. Por su parte, ISGL3D permite importar modelos pero en un formato llamado "POD". Se usó entonces, una aplicación llamada *Collada2POD* que lo que hace es convertir modelos tridimensionales en formato *COLLADA* al formato POD. *Collada2POD* puede ser descargada gratuitamente de la página oficial de *Imagination Technologies*, su desarrollador: `http://www.imgtec.com`.

Una vez que se tiene al objeto 3D en el formato requerido, este puede ser importado en ISGL3D de forma sencilla:

```
Isgl3dPODImporter * podImporter = [Isgl3dPODImporter podImporterWithFile:@''modelo.pod''];

Isgl3dNode * _model = [self.scene createNode];

[podImporter addMeshesToScene:_model];

 _model.position = iv3(2, 6, 0);
```

En la primera línea de código se instancia la clase *Isgl3dPODImporter* que sirve para transformar modelos POD a objetos ISGL3D, y se le asigna a la misma el modelo "modelo.pod". Luego, se crea un nodo llamado "_model", al que se le asigna el modelo; y se agrega a la escena. Finalmente, se le asigna al nodo una posición. En la figura 8.5 se puede ver un modelo de José Artigas, agregado dos veces a una misma escena, pero visto desde ángulos distintos.

Si lo que se quiere es que los modelos sean animados, o lo que es lo mismo, que tengan movimiento, hay dos soluciones posibles a tomar en consideración:

- **Modelo animado**: muchas veces lo que se tiene es un modelo 3D animado desde su construcción. Estos pueden ser creados, al igual que los modelos 3D inanimados, con las herramientas para el creado y la animación de gráficos 3D antedichas. Existe mucha bibliografía al respecto, además de haber múltiples sitios en internet de donde bajar los modelos, incluso en forma gratuita. Luego de obtenido el modelo animado, lo que se tiene es precisamente al modelo, pero con una línea de tiempo con las animaciones. Nuevamente, hay que convertirlo a formato POD para ser usado en ISGL3D. El código para poder visualizar al modelo es:

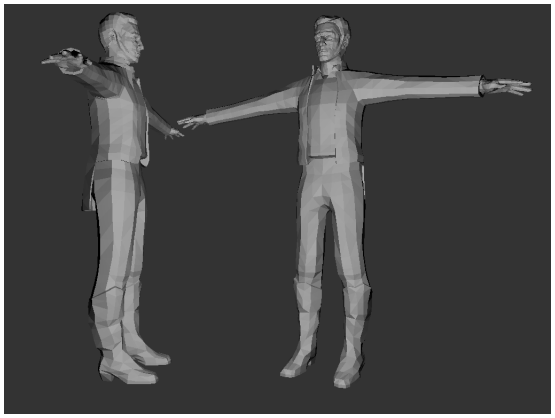

Figura 1.5: Modelo de José Artigas agregado dos veces a una misma escena, pero visto desde ángulos distintos.

```
Isgl3dPODImporter * podImporter = [Isgl3dPODImporter
            podImporterWithFile:@"animated_model.pod"];

Isgl3dSkeletonNode *_model = [self.scene createSkeletonNode];

[podImporter addMeshesToScene:_model];

Isgl3dAnimationController * _animationController = [[Isgl3dAnimationController alloc]
           initWithSkeleton:_model andNumberOfFrames:[podImporter numberOfFrames]];

[_animationController start];
```

En la primera línea de código se instancia la clase *Isgl3dPODImporter*, y se le asigna a la misma el modelo animado "animated_model.pod". Luego, se crea y se agrega a la escena un nodo del tipo *Isgl3dSkeletonNode* llamado "_model" que contiene al modelo animado. La clase *Isgl3dSkeletonNode* provee una interfaz sencilla para animar al ahora objeto ISGL3D, que con la ayuda de la clase *Isgl3dAnimationController*, logra automatizar el movimiento del mismo. Finalmente, se instancia y configura la clase *Isgl3dAnimationController* y se le da inicio a la animación en la última línea.

- **Múltiples modelos inanimados:** otra forma de animar un modelo 3D es usando múltiples modelos inanimados. Estos pueden ser cargados en ISGL3D como un único objeto o nodo y mediante ciertas instrucciones sencillas, se le dice al *framework* que presente uno a continuación del otro, interpolando entre posiciones contiguas, lo que genera una sensación de movimento. Este fue el método utilizado en el presente proyecto para animar al modelo de José Artigas. En la figura 8.6 se puede ver al mismo en 3 posiciones distintas.

  El código para realizar la interpolación mencionada, aplicado por ejemplo a dos modelos, será:

```
Isgl3dPODImporter * podImporter = [Isgl3dPODImporter
          podImporterWithFile:@"model_1.pod"];

[podImporter buildSceneObjects];

Isgl3dPODImporter * podImporter2 = [Isgl3dPODImporter
```

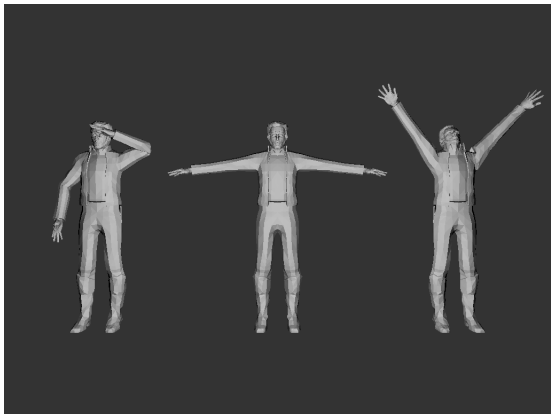

Figura 1.6: Modelo de José Artigas en 3 posiciones distintas, utilizadas para generar en ISGL3D una sensación de movimiento.

```
          podImporterWithFile:@"model_2.pod"];

[podImporter2 buildSceneObjects];

Isgl3dGLMesh* _modelMesh = [podImporter meshAtIndex:0 ];

Isgl3dGLMesh* _modelMesh2 = [podImporter2 meshAtIndex:0 ];

Isgl3dKeyframeMesh * _mesh = [Isgl3dKeyframeMesh keyframeMeshWithMesh:_modelMesh];

[_mesh addKeyframeMesh:_modelMesh2];

[_mesh addKeyframeAnimationData:0 duration:1.0f];
[_mesh addKeyframeAnimationData:0 duration:2.0f];
[_mesh addKeyframeAnimationData:1 duration:1.0f];
[_mesh addKeyframeAnimationData:1 duration:2.0f];

[_mesh startAnimation];

Isgl3dNode * node = [_container createNodeWithMesh:_mesh
           andMaterial:[podImporter materialWithName:@"material_0"]];

node.position = iv3(-90, -60, -150);

[podImporter addMeshesToScene:node];
```

En las primeras 4 líneas de código se instancia en dos oportunidades la clase *Isgl3dPODImporter*, y se les asigna a las instancias los modelos inanimados “model_1.pod” y “model_2.pod’. La instrucción *buildSceneObjects* crea todos los objetos de la escena del modelo POD, pero no los agrega a la escena ISGL3D. Luego se obtienen los modelos indexados de cada uno de los PODs (cada POD puede tener una escena con más de un modelo, mediante un índice se referencia qué modelo se quiere obtener) y se almacenan en “_modelMesh” y “_modelMesh2” respectivamente. Se genera a continuación un nuevo modelo al que se le asignan los dos modelos anteriores, luego se programa la animación y se le da inicio mediante la instrucción *startAnimation*. Finalmente, se genera un nuevo objeto o nodo ISGL3D al que se le asigna el modelo, y un material también cargado desde el archivo POD; se le asigna además una posición y se lo agrega a la escena.

1.6. Luz en ISGL3D

Un tema importante al momento de proyectar modelos 3D en una escena es la luz. La visualización de un modelo puede cambiar significativamente en función de las características de luz que tenga una escena. En ISGL3D la se logra mediante la suma de tres componentes independientes: *ambiente*, *difusa* y *especular*. La luz ambiente es no direccional y está presente en toda la escena. La luz difusa ya implica la reacción que tiene la luz proveniente de las distintas fuentes de luz direccionales sobre las superficies de los objetos que existen en la escena generando haces de luz en direcciones aleatorias. La luz especular modela el comportamiento de la luz reflejada sobre las distintas superficies en ciertas direcciones particulares (no aleatorias) que dependen del coeficiente de reflexión de los materiales. Cada uno de estos tres tipos de luz que modelan el mundo real, existen en ISGL3D y son representados como características configurables de los objetos de luz. A su vez existen fuentes lumínicas de distintos tipos: *puntual*, *direccional* y *cónica*. Cada tipo tiene una función distinta de la atenuación con respecto a la distancia. Un ejemplo de la creación de un elemento lumínico para una escena se ve en el siguiente código:

```
Isgl3dLight * _redLight = [Isgl3dLight lightWithHexColor:@"FF0000" diffuseColor:
@"FF0000" specularColor:@"FFFFFF" attenuation:0.02];
_redLight.renderLight = YES;
[self.scene addChild:_redLight];
```

En la primera línea de código se crea una luz con una componente ambiente de color rojo, una componente difusa también de color rojo y una componente especular de color blanca. En todos los casos se usa la notación hexadecimal de las componentes RGBA. Además se escogió un valor $0,02$ de atenuación. Luego, se pide que el foco de luz sí sea *renderizado* y finalmente, este es agregado a la escena. Por defecto, el tipo de luz utilizado es puntual.

1.7. Método - *(void) tick:(float)dt*

Cuando se instancia la clase encargada de generar el render (clase *HelloWorldView*), la misma ejecuta la configuración básica de inicialización del objeto. Entre otras cosas, dentro del código de inicialización, se agrega lo siguiente:

```
[self schedule:@selector(tick:)];
```

Esto lo que hace es "agendarse" una invocación del método *tick* en forma periódica. Dentro de dicho método es que se hace la actualización de la escena, y el en caso particular de este proyecto, es en dónde se actualiza la posición de los objetos 3D en función de la posición de la cámara respecto de los ejes del mundo. Ver capítulo **??**. Este método es de vital importancia por tratarse de uno de los dos *callbacks* que toda aplicación de realidad aumentada tiene (el otro es la captura y procesamiento de la imagen para obtener la pose).

1.8. ISGL3D en la aplicación

En el capítulo **??** se explican algunos detalles sobre el uso de ISGL3D dentro de la aplicación final. En particular se dan detalles constructivos sobre cómo utilizar esta herramienta para generar *renders* sobre un fondo que sea la captura de la cámara y de la convivencia de los dos *callbacks* de la aplicación.

1.9. Conclusión

En el presente capítulo se mencionaron algunas herramientas existentes para realizar *renders* en dispositivos móviles con sistema operativo iOS. Luego, se justificó la elección de ISGL3D para tal función y se dieron algunos conceptos introductorios a la herramienta; además se dió una hoja de ruta para todo aquel que le interese ampliar sus conocimientos en el *framework*. Se indicaron dos formas distintas de embeber un modelo 3D en ISGL3D y se aclaró cuál fue la forma escogida por el grupo. Finalmente, se referenció el capítulo **??**, en donde se explica cómo se usó ISGL3D en la aplicación final.

Bibliografía

[1] J. García Ocón. Autocalibración y sincronización de múltiples cámaras plz. 2007.

[2] B. Furht. *The Handbook of Augmented Reality*. 2011.

[3] C. Avellone and G. Capdehourat. Posicionamiento indoor con señales wifi. 2010.

[4] Philip David, Daniel Dementhon, Ramani Duraiswami, and Hanan Samet. Simultaneous pose and correspondence determination using line features. pages 424–431, 2003.

[5] Philip David, Daniel Dementhon, Ramani Duraiswami, and Hanan Samet. Softposit: Simultaneous pose and correspondence determination. pages 424–431, 2002.

[6] Daniel F. DeMenthon and Larry S. Davis. Model-based object pose in 25 lines of code. *International Journal of Computer Vision*, 15:123–141, 1995.

[7] Denis Oberkampf, Daniel F. DeMenthon, and Larry S. Davis. Iterative pose estimation using coplanar feature points. *Comput. Vis. Image Underst.*, 63(3):495–511, may 1996.

[8] R. Grompone von Gioi, J. Jakubowicz, J. M. Morel, and G. Randall. Lsd: A fast line segment detector with a false detection control. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(4):722–732, April 2010.

[9] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second edition, 2004.

[10] V. Lepetit and P. Fua. Monocular model-based 3d tracking of rigid objects: A survey. *Foundations and Trends in Computer Graphics and Vision*, 1(1):1–89, 2005.

[11] Zhengyou Zhang. Flexible camera calibration by viewing a plane from unknown orientations. In *ICCV*, pages 666–673, 1999.

[12] Jane Heikkilä and Olli Silvén. A four-step camera calibration procedure with implicit image correction. In *1997 Conference on Computer Vision and Pattern Recognition (CVPR 97), June 17-19, 1997, San Juan, Puerto Rico*, page 1106. IEEE Computer Society, 1997.

[13] Jean-Yves Bouguet. Camera calibration toolbox for matlab. `http://www.vision.caltech.edu/bouguetj/calib_doc/`, November 2012.

[14] Helmut Zollner and Robert Sablatnig. Comparision of methods for geometric camera calibration using planar calibration targets. In W. Burger and J. Scharinger, editors, *Digital Imaging in Media and Education, Proc. of the 28th Workshop of the Austrian Association for Pattern Recognition (OAGM/AAPR)*, volume 179, pages 237–244. Schriftenreihe der OCG, 2004.

[15] Stuart Caunt. Isgl3d homepage. `http://www.isgl3d.com`, November 2012.

[16] R.Y. Tsai. An efficient and accurate camera calibration technique for 3d machine vision. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 364–374, 1986.

[17] Y. I. Abdel-Aziz and H. M. Karara. Direct linear transformation from comparator coordinates into object space coordinates in close-range photogrammetry. In *Proceedings of the Symposium on Close-Range photogrammetry*, volume 1, pages 1–18, 1971.