MESHES

Durante los próximos días aprenderemos todo lo necesario para exportar objetos 3D desde el 3D Studio MAX hasta nuestro videojuego.

Template Manager

Para realizar diferentes managers a lo largo del máster vamos a utilizar una clase templatizada que nos facilite el trabajo. A continuación ponemos la definición de la clase que queremos realizar:

```
template<class T>
class CMapManager
{
	protected:
		typedef std::map<std::string, T*>	TMapResource;
		TMapResource				m_Resources;
	public:
		virtual T * GetResource(const std::string &Name);
		virtual void AddResource(const std::string &Name, T *Resource);
		void Destroy();
};
```

Nos encontramos con tres métodos iniciales como son:

- *GetResource*: que nos devolverá el resource según el *Name* que le pasamos como key del mapa.
- *AddResource*: añadirá el resource según el nombre *Name* en el mapa.
- *Destroy*: destruye todos los resources del mapa y lo limpia.

CTemplatedVectorMapManager

Al igual que implementamos una clase templatizada CTemplatedMapManager que nos permitía contener cualquier tipo de clase ordenado por un mapa, vamos a implementar una clase también templatizada que nos va a permitir gestionar un conjunto de elementos bien por su nombre como un mapa bien como un vector.

Esta clase va a contener un mapa de elementos CMapResourceValue con clave string (el nombre del elemento) y un vector con los elementos ordenados linealmente, lo realizaremos de esta forma porque así podremos recorrer el conjunto de elementos de forma lineal como un vector o de forma por clave como un mapa.

La clase CMapResourceValue es una clase que contiene dos propiedades el elemento y el índice del elemento dentro del vector de la clase CTemplatedVectorMapManager.

La clase va a ser como sigue:

```
template<class T>
class CTemplatedVectorMapManager
{
public:
	class CMapResouceValue
	{
	public:
```

```
            T                   *m_Value;
            size_t              m_Id;
            CMapResouceValue(T *Value, size_t Id)
                  : m_Value(Value)
                  , m_Id(Id)
            {
            }
      };
      typedef std::vector<T *>                              TVectorResources;
      typedef std::map<std::string, CMapResouceValue>       TMapResources;
protected:
      TVectorResources                              m_ResourcesVector;
      TMapResources                                 m_ResourcesMap;
public:
      CTemplatedVectorMapManager();
      virtual ~CTemplatedVectorMapManager();
      void RemoveResource(const std::string &Name);
      virtual T * GetResourceById(size_t Id);
      virtual T * GetResource(const std::string &Name);
      virtual bool AddResource(const std::string &Name, T *Resource);
      virtual void Destroy();
      TMapResources & GetResourcesMap();
      TVectorResources & GetResourcesVector();
};
```

Esta clase contiene los siguientes métodos:

- *Construtor,* construye los elementos necesarios de nuestra clase
- *Destructor,* llama al método destroy de la clase.
- *RemoveResource,* elimina un elemento de la clase, tanto del vector como del mapa, según el nombre del elemento.
- *GetResourceById,* nos devuelve un elemento del vector según el índice.
- *GetResourceByName,* nos devuelve un elemento según el nombre.
- *Destroy,* destruye los elementos que están introducidos dentro del mapa y del vector, los elementos estarán duplicados como punteros, pero únicamente estarán una vez en memoria, por tanto deberemos eliminarlos sólo una vez bien por el vector bien por el mapa.
- *GetResourcesMap,* nos devuelve el mapa de elementos de la clase.
- *GetResourcesVector,* nos devuelve el vector de elementos de la clase.

Texture

La clase CTexture va a ser la clase que utilizaremos para envolver la clase textura del DirectX.

Intentaremos, siempre que podamos, evitar utilizar directamente las funciones, tipos, estructuras de DirectX. Si lo hacemos correctamente realizar un port a otra plataforma diferente de DirectX debería ser tan sencillo como modificar la capa inferior de renderizado.

```
class CTexture
{
      protected:
            LPDIRECT3DTEXTURE9                  m_Texture;
```

```
        std::string                                     m_FileName;

        virtual bool LoadFile();
        void Unload();
    public:
        CTexture();
        ~CTexture();
        const std::string & GetFileName() const;
        bool Load(const std::string &FileName);


        bool Reload();
        void Activate(size_t StageId);
};
```

Nos encontramos con los siguientes métodos:

- *LoadFile*: cargará la textura del fichero en memoria de video.
- *Unload*: hará el *Release* de la textura y establecerá a *NULL* el *m_Texture*.
- *CTexture*: el constructor inicializará los atributos de la clase *CTexture*.
- *~CTexture:* el destructor eliminará la textura llamando al método *Unload*.
- *GetFileName*: nos devuelve el nombre de la textura.
- *Load*: llamar al método *LoadFile* para cargar la textura.
- *Reload*: descarga y carga la textura.
- *Activate*: activa la textura en la etapa según el *StageId*.

Texture Manager

La clase *CTextureManager* será el mánager de texturas que utilizaremos para cargar cualquier textura dentro de nuestro juego. Su implementación será:

```
class CTextureManager : public CMapManager<CTexture>
{
    public:
        CTextureManager();
        ~CTextureManager();
        void Reload ();
};
```

Nos encontramos con los siguientes métodos:

- *CTextureManager*: el constructor inicializará los atributos de la clase *CTextureManager*.
- *~CTextureManager:* el destructor eliminará la textura llamando al método *Destroy*.
- *Reload*: recorrerá todos los resources del MapManager y hará el *reload* de cada una de las texturas.

CRenderableVertexs

Para poder utilizar Vertex Buffers y Index Buffers de cualquier tipo y de forma sencilla vamos a realizar una clase base que se llamará CrenderableVertexs que tendrá una estructura como la siguiente.

```
class CRenderableVertexs
{
protected:
    LPDIRECT3DVERTEXBUFFER9          m_VB;
    LPDIRECT3DINDEXBUFFER9           m_IB;
    size_t                           m_IndexCount, m_VertexCount;
public:
    CRenderableVertexs();
    virtual ~CRenderableVertexs() {}
    virtual bool Render(CRenderManager *RM) const = 0;
    virtual inline size_t GetFacesCount() const;
    virtual inline size_t GetVertexsCount() const;
    virtual inline unsigned short GetVertexType() const = 0;
    virtual inline size_t GetVertexSize() = 0;
    virtual inline size_t GetIndexSize() = 0;
}
```

Esta clase incorpora los métodos siguientes:

- *Render*: se encargará de renderizar la malla según.
- *GetFacesCount*: devolverá el número de caras que contiene la malla.
- *GetVertexsCount*: devolverá el número de vértices que contiene la malla.
- *GetVertexSize*: devolverá el tamaño del vértice en bytes, según la estructura de vértice.
- *GetIndexSize*: devolverá el tamaño del índice en bytes, según la estructura de índice.

Una vez tenemos la clase base creada, vamos a crear una clase templatizada por cada tipo de vértice que derive de CRenderableVertexs. La clase la vamos a definir de la siguiente manera.

```
template<class T>
class CIndexedVertexs : public CRenderableVertexs
{
protected:
    inline size_t GetVertexSize() {return sizeof(T);}
    inline size_t GetIndexSize() {return sizeof(unsigned short);}
public:
    CIndexedVertexs(CRenderManager *RM, void *VertexAddress, void *IndexAddres, size_t
VertexCount, size_t IndexCount);
    ~CIndexedVertexs();
    virtual bool Render(CRenderManager *RM);
```

```
    virtual inline unsigned short GetVertexType() const {return T::GetVertexType();}
}
```

Por último creamos los diferentes vértices que vamos a utilizar en nuestra aplicación de forma similar a la siguiente estructura.

```
#define VERTEX_TYPE_GEOMETRY            0x0001
#define VERTEX_TYPE_NORMAL              0x0002
#define VERTEX_TYPE_TANGENT             0x0004
#define VERTEX_TYPE_BINORMAL            0x0008
#define VERTEX_TYPE_TEXTURE1            0x0010
#define VERTEX_TYPE_TEXTURE2            0x0020
#define VERTEX_TYPE_DIFFUSE             0x0040

struct TCOLORED_VERTEX
{
    float                               x, y, z;
    unsigned long                       color;
    static inline unsigned short GetVertexType()
    {
        return VERTEX_TYPE_GEOMETRY|VERTEX_TYPE_DIFFUSE;
    }
    static inline unsigned int GetFVF()
    {
        return D3DFVF_XYZ|D3DFVF_DIFFUSE;
    }
};
```

MAX Script

Para la exportación de las mallas desde el 3D Studio MAX vamos a utilizar el lenguaje de scripting que tiene integrado la propia herramienta.

Según la ayuda de la aplicación:

What is MAXScript?

MAXScript is the built-in scripting language of **Autodesk® 3ds Max®** and **Autodesk® 3ds Max® Design**.

MAXScript provides users of these products with the ability to:

- Script most aspects of the program's use, such as modeling, animation, materials, rendering, and so on.
- Control the program interactively through the command-line Listener window.
- Package scripts within custom Utility panel rollouts or modeless windows to give them a standard user interface.
- Package scripts as macro scripts, and install these macro scripts as buttons in the product's toolbars, as items in menus, or assign them to keyboard shortcuts.
- Extend or replace the user interface for objects, modifiers, materials, textures, render effects, and atmospheric effects.
- Build scripted plug-ins for custom mesh objects, modifiers, render effects and more.
- Build custom import/export tools using ASCII and binary file I/O.
- Write procedural controllers that can access the entire state of the scene.
- Build batch-processing tools, such as batch-rendering scripts.
- Set up live interfaces to external systems through OLE Automation.
- Record your actions in the product as MAXScript commands.
- Store scripts in scene files to run at each of the supported notification events, for example when a scene has been reset, a file has been opened or saved, rendering has been started or stopped, object selection has changed and so on.

Por tanto para realizar exportaciones directamente desde la propia herramienta debemos conocer como funciona este lenguaje de scripting y su sintaxis.

Conceptos básicos

Declaración de variables

Para declarar una variable deberemos escribir el nombre de la variable y asignarle un valor, como vemos no especificamos el tipo de la variable.

```
MyVariable=”hola mundo”
MyVarNumber=3.0
```

Las variables podrán ser de tipo locales o globales, pero locales sólo podremos utilizarlas dentro de un ámbito definido.

```
global MyVariable=”hola mundo global ”
...
(
	local MyVarNumber=3.0
)
```

format

Para escribir texto, valores o mostrar datos por la consola utilizaremos el comando format.

```
MyVariable =3
format “hola mundo”
format “texto con fin de linea\n”
format “texto con variable % \n” MyVariable
```

if

La expresión *if* tiene la siguiente sintaxis

```
MyVariable=3
if MyVariable==3 then
(
	local MyVarOk=true
)
```

Si queremos utilizar la expresión else

```
MyVariable=3
if MyVariable==3 then
(
	local MyVarOk=true
)
else
(
	local MyVarOk=false
)
```

for

La expresión *for* tiene la siguiente sintaxis

```
for i=1 to 10 do
(
	local MyVar=I*2
)

for i=1 to 10 by 2 do
(
	local MyVar=I*2
)
```

while
La expresión *while* tiene la siguiente sintaxis

```
var I=0
while i<10 do
(
    i=i+1
)
```

do ... while
La expresión *do ... while* tiene la siguiente sintaxis

```
var I=0
do
(
    i=i+1
) while i<10
```

Array
Para crear un array en MAXScript utilizaremos el carácter # al crear la variable y para saber el número de elementos el atributo count del array. Mencionar que los arrays en MAXScript comienzan por 1, el primer eleemnto se accede mediante el operador [ ] con índice 1.

```
local MyAray=#()
format “el array tiene % elementos\n” MyArray.count
format “el primer elemento tiene el valor de %” MyArray[1]
```

function
Si queremos declarar una función lo implementaremos de la siguiente manera

```
function function_name parameter1 parameter2 =
(
    local VarLocal=3
    VarLocal=parameter1+parameter2
    return VarLocal
)
```

extras
A continuación ponemos un carro de funciones que nos interesará conocer en MAXScript.

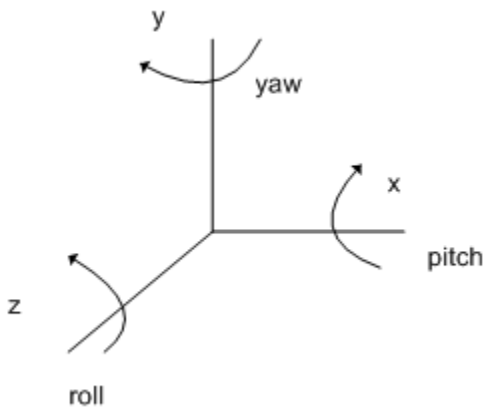
```
-- Esto es un comentario de una única línea
/* Esto es un comentario
de varias líneas */
$
-- Nos mostrará el objeto seleccionado

classof $
-- Nos mostrará la clase del objeto que tenemos seleccionado

(classof $)==ObjectSet
-- Nos dirá si tenemos seleccionado varios elementos o un único elemento

$.count
-- Nos dirá el número de elementos seleccionados en una selección múltiple

$[1]
```

```
-- Nos devolverá el objeto seleccionado en primer lugar

showProperties $
-- Nos devolverá las propiedades del objeto seleccionado

select $*
-- Selecciona todos los elementos de la escena

local ArrayName=#()
-- Crea una variable de tipo array con nombre ArrayName

append ArrayName 5
-- Añade el elemento 5 al array

ArrayName.count
-- Devuelve el número de elementos que tiene introducidos el array
```

RH TO LH Translation, YAW, PITCH, ROLL

Como sabréis el 3D Studio MAX utiliza los ejes de coordenadas en formato Right Handed, mientras que en DirectX utilizamos la regla Left Hended, es por esto que deberemos hacer conversiones de las coordenadas de RH a LH.

Para realizar estas conversiones de las matrices deberemos extraer de las la translación y los ángulo yaw, pitch y roll.



Para realizar esta tarea extraeremos los ángulos de rotación de la matriz de transformación del objeto del 3D Studio MAX mediante el código fuente siguiente.

```
fn GetYaw transform =
(
    return (transform as eulerangles).z
)

fn GetPitch transform =
(
    return (transform as eulerangles).x
)

fn GetRoll transform =
(
    return (transform as eulerangles).y
```

```
)
YawAngle=GetYaw $.transform
PitchAngle= GetPitch $.transform
RollAngle= GetRoll $.transform
```

Y por último para convertir una posición de RH a LH utilizaremos el siguiente fragmento de código.

```
fn RHTranslationToLH translation =
(
    return point3 translation.x translation.z translation.y
)
$.transform.translation
```

Materiales

Para poder exportar los materiales aplicados sobre la malla con la que estamos trabajando deberemos acceder al material aplicado sobre el objeto con el siguiente fragmento de código.

```
$.material
-- Podemos ver todas las propiedades del material con la llamada de la función showProperties
showProperties $.material
```

En primera instancia nos van a interesar las propiedades del material de difuso, bump y reflection. Para acceder a estos mapas del material y sus propiedades utilizamos el código siguiente.

```
-- Para ver el mapa de difuso
$.material.diffuseMap
-- Para ver el nombre del fichero del diffuseMap cuando el artista creo el archivo
$.material.diffuseMap.filename
-- Para ver el nombre del fichero en el ordenador local deberemos implementar el siguiente código
(openBitMap $.material.diffuseMap.bitmap.filename).fileName
-- Para ver el mapa de Bump
$.material.bumpMap
-- Para ver el mapa de Reflection
$.material.reflectionMap
-- Para ver el color de difuso aplicado sobre el objeto en formato RGB
$.material.diffuse
```

Por último comentar que habrá situaciones dónde el artista aplicará un multimaterial sobre la malla para poder aplicarl diferentes materiales sobre el mismo objeto, en ese caso nos encontramos que el material no será de tipo material si no de tipo multimaterial.

```
-- Para saber si el objeto tiene aplicado un multimaterial
if classof(material) == Multimaterial then
(
    format “el objeto tiene aplicado un multimaterial\n”
)
else
(
    format “el objeto tiene aplicado un material standard\n”
)
/* En caso de tener un multimaterial, para saber el número de materiales que tiene aplicados
utilizaremos la función numsubs */
```

```
$.material.numsubs
-- Para acceder al primer material de los materiales de un multimaterial
$.material[1].diffuseMap
```

Faces y Vertexs

Con tal de extraer la información de las caras y vértices lo realizaremos mediante un objeto de tipo *Editable Mesh*.

El siguiente fragmento de código nos permitirá extraer toda la información necesaria de la malla.

```
-- Para saber el número de caras totales que tiene el objeto seleccionado
getNumFaces $

-- Para saber el número de vértices totales que tiene el objeto seleccionado
getNumVerts $

-- Para coger los indices de la primera cara
local IdxsFace=getFace $ 1

-- Para coger los vértices de la primera cara
local Vtx1=getVert $ IdxsFace.x
local Vtx2=getVert $ IdxsFace.y
local Vtx3=getVert $ IdxsFace.z

-- Para coger las normales de los vértices de la primera cara
local Normal1=getNormal $ IdxsFace.x
local Normal2=getNormal $ IdxsFace.y
local Normal3=getNormal $ IdxsFace.z

/* Para coger los índices de las coordenadas de textura de la primera cara, el 1 señala al primer
conjunto de coordenadas aplicadas sobre la cara */
local IdxsMap=meshop.getMapFace $ 1 IdFace

-- Para coger las coordenadas de los vértices de la primera cara
local TUVMap1= meshop.getMapVert $ 1 IdxsMap.x
local TUVMap2= meshop.getMapVert $ 1 IdxsMap.y
local TUVMap3= meshop.getMapVert $ 1 IdxsMap.z

/* Por ultimo para saber el Id del material que tiene aplicada esta cara en caso de estar utilizando
un multimaterial sobre el objeto */
local MaterialID=getFaceMatID $ IdFace
```

Tratamiento de ficheros en MAX Script

Para poder escribir en ficheros desde MAXScript nos encontramos con las siguientes funciones que nos permitirá realizar estas tareas.

```
-- Para crear un fichero en modo escritura binaria
local file=fopen filename "wb"

-- Para escribir un elemento con nombre Value en formato string
WriteString file Value

-- Para escribir un elemento con nombre Value en formato unsigned short
WriteShort file Value #unsigned
```

```
-- Para escribir un elemento con nombre Value en formato unsigned long
WriteLong file Value #unsigned

-- Para escribir un elemento con nombre Value en formato float
WriteFloat file Value

-- Para hacer un flush de los bytes no escritos en el fichero
fflush file

-- Para cerrar el fichero
fclose file
```

Código extra de MAX Script

A continuación vamos a ver código que podemos utilizar en MAXScript no incluído todavía

```
--Para incluir un fichero de maxscript de forma externa
fileIn "UABFunctions.ms"

--Para sacar las instancias de un objeto
InstanceMgr.GetInstances Obj &instances

--Para saber si un objeto es instancia de otro objeto
areNodesInstances Obj Objs[b]

--Para extraer una propiedad de usuario de un elemento 3D del MAX
getUserProp obj "export_type"

--Para establecer una propiedad de usuario de un elemento 3D del MAX
setUserProp obj "export_type" “animated_instance_model”

--Para realizar operaciones binarias
local value=VERTEX_TYPE_GEOMETRY
value=bit.or value VERTEX_TYPE_TEXTURE

--Para comprobar si un bit está activado
(bit.and VertexType VERTEX_TYPE_TEXTURE)==VERTEX_TYPE_TEXTURE

--Para extraer el nombre del fichero de un path global
file="g:\\subdir1\\subdir2\\myImage.jpg"
filenameFromPath file -- returns: "myImage.jpg"

--Para extraer el path del path global
getFilenamePath file -- returns: "g:\subdir1\subdir2\"

/* Para copiar un fichero de una carpeta origen a una carpeta destino, cabe destacar que tenemos
que ponerle el nombre del fichero de destino y que el fichero destino no puede existir*/
copyFile "c:\\a.bin" “d:\\a.bin”

-- Para borrar un fichero
deleteFile “d:\\a.bin”

--Para crear un directorio
makeDir “c:/UABProject/Run/data” all:true

--Para mostrar un mensaje mediante un messagebox
messageBox “Mensaje de atención” title:"ATTENTION"

--Para crear una ventana de explorador dónde seleccionar un path
local data_path=getSavePath caption:"Data path" initialDir:PathLabel.text
```

```
if data_path!=undefined then

--Para crear una ventana de explorador dónde seleccionar un fichero
local out_name=GetSaveFileName filename:(GetMeshesPathExport()+$.name+".mesh")
        caption:"Select output file to export file" types:" mesh(*.mesh)|*.mesh|All Files(*.*)|*.*|"
if out_name!=undefined then


-- Para hacer una copia del objeto seleccionado
copy $

-- Para eliminar el objeto seleccionado
delete $

-- Para convertir el objeto seleccionado en EditableMesh
convertToMesh $
```

CStaticMesh

Para poder utilizar las mallas exportadas desde 3D Studio MAX crearemos una clase de tipo CstaticMesh con una estructura similar al siguiente código.

```
class CStaticMesh
{
protected:
    std::vector<CRenderableVertexs*>          m_RVs;
    std::vector<std::vector<CTexture *>>      m_Textures;
    std::string                               m_FileName;
    unsigned int                              m_NumVertexs, m_NumFaces;
public:
    CStaticMesh();
    ~CStaticMesh();
    bool Load (const std::string &FileName);
    bool ReLoad ();
    void Render (CRenderManager *RM) const;
};
```

Nos encontramos con los diferentes métodos que nos permitirán:

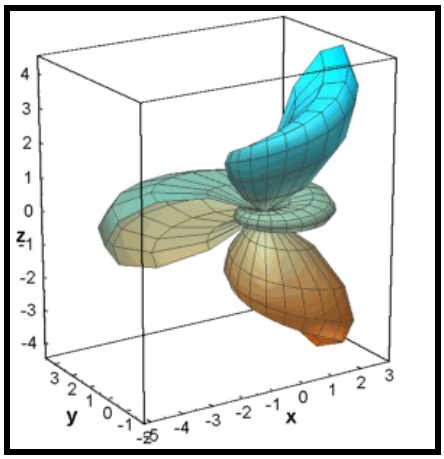
- *Load*: cargará la malla según el fichero que le pasamos como parámetro.
- *Reload*: hará una recarga de la malla en caliente del objeto.
- *Render*: renderizará la malla en pantalla.

Cajas y esferas contenedoras

Una vez tenemos creada las clase CStaticMesh debemos introducir la idea de las cajas contenedoras y las esferas contenedoras, elementos lógicos que definen:

- una caja contenedora (Bounding Box) es una caja que contiene todos los vértices del objeto en sus contenido, estará definida por dos coordenadas geométricas XYZ, la posición mínima y máxima de la caja, para encontrar estos puntos lo que haremos será recorrer todos los vértices de la malla y coger la coordenada X mínima, Y mínima y Z mínima y lo mismo para la X

máxima, Y máxima y Z máxima. Dándonos como resultado la caja contenedora de nuestra malla. Algo parecido al ejemplo siguiente.



- Una esfera contenedora (Bounding Sphere) es una una esfera que contiene todos los vértices del objeto en su interior, está definida por una posición que será el centro de la esfera y el radio de la esfera. El centro de la esferá saldrá del punto intermedio de los puntos mínimos y máximos de la caja contenedora, y el radio saldrá de realizar el cláculo de distáncia máxima desde el centro hasta todos los vértices del objeto.

CStaticMeshManager

Realizaremos un mánager de StaticMeshManager dónde cargaremos todos las cores de las StaticMeshes.

Para ello realizaremos una clase como la descrita a continuación que permitirá leer un fichero .xml.

```
class CStaticMeshManager : public CMapManager<CStaticMesh>
{
protected:
	std::string 	m_FileName;
public:
	CStaticMeshManager();
	~ CStaticMeshManager ();
	bool Load(const std::string &FileName);
	bool Reload();
};
```

Para poder grabar en fichero xml desde MAXScript utilizaremos el siguiente código para escribir en formato texto.

```
-- Para crear un fichero de tipo carácter stream
local file=createfile "static_mesh_manager.xml"

-- Para escribir sobre el fichero de tipo carácter stream utilizando la función format
format "<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?>\n" to:file
format "<static_meshes>\n" to:file
format "\t<static_mesh name=\"%\" filename=\"%\">\n" "Box" “data/static_meshes/box.mesh” to:file
format "</static_meshes>\n" to:file

-- Para cerrar el fichero de tipo carácter stream
close file
```

CRenderableObjectsManager

Una vez tenemos creado nuestra CStaticMeshManager vamos a crear una clase que se va a encargar de controlar todos los elementos renderizables de nuestro videojuego, para ello crearemos la clase CRenderableManager que derivará de la clase CMapManager para poder acceder a un elemento mediante su nombre y un vector de CRenderableObjects para para poder renderizar todos sus objectos de forma lineal.

Para ello realizaremos una clase como la descrita a continuación que permitirá leer un fichero .xml.

```
class CRenderableObjectsManager : CMapManager<CRenderableObject>
{
private:
	std::vector<CRenderableObject *>	m_RenderableObjects;
public:
```

```
        CRenderableObjectsManager();
        ~CRenderableObjectsManager();
        void Update(float ElapsedTime);
        void Render(CRenderManager *RM);
        CRenderableObject * AddMeshInstance(const std::string &CoreMeshName, const
                std::string &InstanceName, const Vect3f &Position);
        CRenderableObject * AddAnimatedInstanceModel(const std::string &CoreModelName,
                const std::string &InstanceModelName, const Vect3f &Position);
        void AddResource(const std::string &Name, CRenderableObject *RenderableObject);
        void CleanUp();
        void Load(const std::string &FileName);
        CRenderableObject * GetInstance(const std::string &Name) const;
};
```

CRenderableObject

Una vez tenemos creado el manager de CRenderableObject crearemos cada la clase que hará a modo de interfaz para cualquier elemento renderizable en nuestro proyecto.

Para ello realizaremos una clase como la descrita a continuación.

```
class CRenderableObject : public CObject3D, public CNamed
{
public:
        CRenderableObject();
        virtual ~CRenderableObject() {}
        virtual void Update(float ElapsedTime) {}
        virtual void Render(CRenderManager *RM) = 0;
};
```

Es interesante añadir la propiedad m_Visible y sus métodos Get y Set en la clase CObject3D para renderizar ese elemento sólo en el caso de ser visible.

CMeshInstance

Una vez tenemos la clase CRenderableObject crearemos la clase CMeshInstance que nos permitirá renderizar una instancia de una CStaticMesh, utilizaremos un código como el siguiente.

```
class CInstanceMesh : public CRenderableObject
{
        private:
                CStaticMesh                     *m_StaticMesh;
        public:
                CInstanceMesh(const std::string &Name, const std::string &CoreName);
                ~CInstanceMesh();

                void Render(CRenderManager *RM);
};
```

GUI en MAX Script

Para terminar vamos a explicar como crear una utility que nos permite MAX Script realizar GUI's de nuestros scripts.

¿Qué es una utility?

Una utility es una herramienta que contiene GUI realizada con MAX Script y que engloba las difererentes funcionalidades que nos permite el interfaz.

Para crear una utility deberemos escribir un código similar al siguiente.

```
utility NombreUtility "Nombre que saldrá en las herramientas"
(
    -- Contenido de la utility
)
```

Una vez tenemos creada la utility podemos introducir controles similares a los que podemos encontrar en la librería de MFC por ejemplo.

Encontramos los controles típicos como:

- Bitmap, button, check button, combo box, list box, edit box, label, etc

Para introducir un control dentro de la GUI pulsaremos sobre la opción de menú Tools/Edit Roll Out.

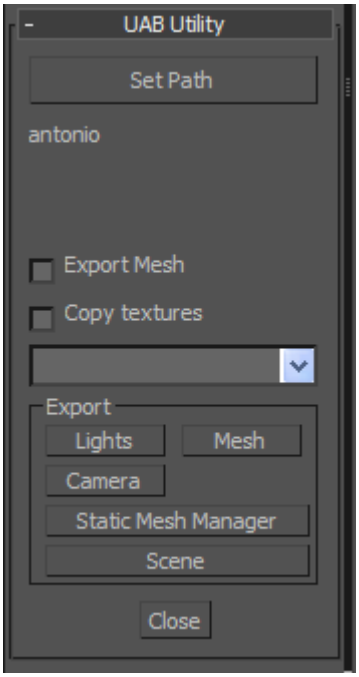
Una vez tenemos la utility terminada pasaremos a evaluarla mediante la opción de teclado Ctrl+E y la tendremos disponible en el menú de Utilities en el apartado de MaxScript.

UAB Utility

A continuación vamos a crear una utility que va a permitir a los artistas exportar el escenario y lógica del videojuego directamente desde el MAX.

```
utility UABUtility “UAB Utility”
(
    -- Contenido de la utility
)
```

Para ello vamos a crear una utility con un roll out similar al siguiente.



En nuestra utility vemos los siguientes controles que debemos aprender a utilizar.

- Botón Set Path, nos permitirá establecer el path dónde queremos que se genere la estructura de datos de nuestra nivel.
- Label path, nos dirá el path dónde se generará el nuestra escena.
- Checkboxes de Export Mesh y Copy Textures, nos permitirá activar o desactivar la exportación de las mallas y el copiado de texturas.
- Combo box con los diferentes niveles, si queremos crear un nuevo nivel lo podremos meter en ese combo box y a la hora de exportarse se añadirá al combo box
- Diferentes botones que nos permitirá exportar luces, cámaras, una malla estática determinada, cámara, el manager de mallas estáticas y la escena

Para poder implementar esta utility vamos a necesitar conocer las siguientes funcionalidades.

```
--Esta función se llama al abrirse la utility en la sección de herramientas del 3D Studio MAX
on UABUtility open do
```

```
(
	--Introducid aquí el código de inicialización de la utility
)
--Esta función se llama al pulsarse sobre el btón ExportMeshButton
on ExportMeshButton pressed do
(
	--Implementad aquí el código de pulsado de este botón
)
--Para introducir una lista de elementos dentro de DropDownList utilizaremos el siguiente código
local levels=#("","level 1","level 2", "level 3","level 4", "main menu", "select player")
LevelList.items=levels
--Para escribir texto sobre el label del Path
PathLabel.text="C:/UABEngine/Data"
--Para comprobar si un checkbox está checked
ExportMeshCheck.checked
--Para saber el elemento seleccionado en un DropDownList
LevelList.selection
--Para extraer el texto del elemento seleccionado en el DropDownList
LevelList.items[LevelList.selection]
```

UAB SetProperties

A continuación vamos a crear un diálogo que va a permitir a los artistas modificar las propiedades de exportación de los objetos modificando las propiedades de usuario.

```
rollout UABSetProperties "UABSetProperties" width:264 height:406
(
    -- Contenido del rollout
)
createDialog UABSetProperties
```

Para ello vamos a crear un rollout similar al siguiente.

En este rollout encontramos los siguientes apartados de propiedades de exportación.

- *Render,* en este apartado introduciremos las propiedades referentes al renderizado, dónde podemos encontrar las siguientes propiedades.
    - *Export*, el tipo de objeto que queramos exportar, podremos exportar instancias de mallas estáticas o de mallas animadas.
    - *Visible,* propiedad de objeto visible o invisible.
    - *Layer*, encontraremos la lista de capas dónde querremos que se renderice dicho elemento, un ejemplo sería: solid, alpha_objects, alpha_blend_objects, particles

    - *Anim. Model*, especifica la core del modelo animado que estamos introduciendo

- *Physics*, en este apartado introduciremos las propiedades referentes al sistema físico, dónde podemos encontrar las siguientes propiedades.
    - *Generate*, si queremos que un modelo genere elemento colisionable dónde puede ser de tipo: malla triangular, forma convexa, forma caja, forma esfera, forma plano, forma cápsula
    - *Layer*, para poder gestionar después los grupos de colisión
- *Logic*, en este apartado introduciremos los elementos de lógica que querremos exportar, podremos incluir elementos cajas de eventos, esferas de eventos, escaleras, filos agarrables, filos dónde dejarse caer, …