COSC422 Advanced Computer Graphics
Programming Exercise 11
Introduction to Asset Import Library (Assimp)

This programming exercise aims to familiarise you with the scene graph structure consisting of a hierarchy of transformations used by model loading libraries such as Assimp.

Assimp:

Please download and install Assimp (version 3.3 or 4.0 or 4.1) from

http://www.assimp.org/index.php/downloads

A documentation on the definitions of classes and their structures can be found at

http://sir-kimmi.de/assimp/lib_html/data.html

ModelLoader.cpp:

The program `ModelLoader.cpp` provides a simple implementation of a 3D model loader using Assimp. The file `assimp_extras.h`. contains a set of helper functions useful in such implementations.

The function `loadModel()` creates the scene object for the input model. The name of the input model file is specified in the `initialize()` function.

The function `render()` is the core of the model loader. This recursive function is used to traverse the scene graph from its root node. Each node (`nd`) of the scene graph stores an array of mesh indices given by

`meshIndex = nd->mMeshes[n],`   n = 0 … (`nd->mNumMeshes)-1.`

Using these indices, the mesh objects are retrieved from the scene object:

`mesh = scene->mMeshes[meshIndex]`   (see Slide [7]-18).

Each mesh contains a single material index given by `mesh->mMaterialIndex.` Some mesh models may have colour values associated with each vertex (`mesh->HasVertexColors(0)).` `Assimp` supports multiple channels of vertex colours. Most commonly, only the first channel with index 0 is used.

After drawing all polygons of the current mesh, the next mesh in the current node is processed. After processing the current node, the `render()` function is recursively called to descend to the child nodes. The transformation hierarchy in the scene graph is directly translated into a nested structure of glPushMatrix()-glPopMatrix() blocks.

The `display()` function renders the model as generated by the `render()` function. The bounding box of the model is used to scale the model to fit within the display window. Most mesh model definitions use the *z*-axis as the primary axis for modelling, and require a −90 degs rotation about the *x*-axis. By default, the program displays models after this transformation. Use the keyboard input ‘1’ to turn this rotation on or off. The print functions in `loadModel()` may be uncommented to get the output of scene and node parameters. Note: The models provided with this exercise do not contain any animation data.

A set of mesh models are provided in a zip file, and the outputs are shown below in Fig. 1. You may replace the default colour value used for rendering a model with your own colour value by setting “replaceCol” to ‘true’, and assigning the colour values to the variable “materialCol”.

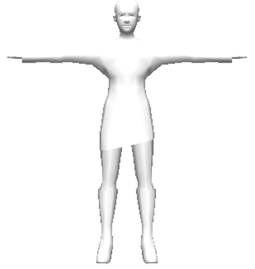		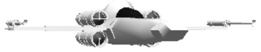	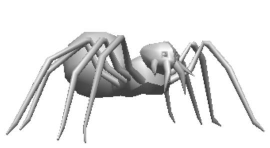
Ana.x	Truck.3ds	SpaceShip.3ds	Spider.obj

Fig. 1.

Texture Mapping:

The program `ModelLoader.cpp` includes a function `loadGLTextures()` to load images stored in various formats as textures. The Developer’s Imaging Library (DevIL/OpenIL) is used for this purpose.

If a model file uses textures, it will have the relative path and file names of textures stored in material definitions. All four models in Fig. 1 use textures. The function `loadGLTextures()` visits all material objects of the scene and checks if any of them has any texture file information stored in them. If a material object contains a texture file name, the function loads that texture and associates the texture id with the material id using a hash map (`texIdMap`).

We will now implement texture mapping inside the `render()` function of the program. Uncomment the statement `//loadGLTextures(scene);` in the initialize() function.

Inside the loop for mesh objects in the `render()` function, check if the current mesh has texture coordinates: `if (mesh->HasTextureCoords(0)) { ... }`

If the current mesh has texture coordinates, then enable OpenGL texturing, obtain the texture id using the material id attached to the mesh from the hash map `texIdMap`, and call glBindTexture() with that texture id. Inside the glBegin-glEnd block, specify the current vertex’s texture coordinates as

```
glTexCoord2f  (mesh->mTextureCoords[0][vertexIndex].x ,
               mesh->mTextureCoords[0][vertexIndex].y   );
```

You should now be able to get the display of texture mapped 3D models as shown in Fig. 2.

		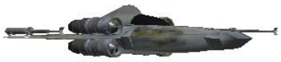	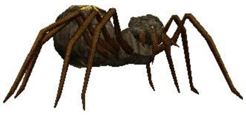
Ana.x	Truck.3ds	SpaceShip.3ds	Spider.obj

Fig. 2.

[7]: COSC422 Lecture Slides: Lec07_MeshFiles.pdf