Practical Assignment CG - 2022/23

MIEI & LCC DI - UM

The goal of this assignment is to develop a mini scene graph based 3D engine and provide usage examples that show its potential. The assignment is split in four phases, each with a due date as specified in the course page.

For each phase a set of configuration XML files will be provided for testing and evaluation purposes. Each configuration is accompanied by the respective visual output. Students should verify that their output is identical to the sample images provided for each configuration file.

Phase 1 – Graphical primitives

This phase requires two applications: one to generate files with the models information (in this phase only generate the vertices for the model) and the engine itself which will read a configuration file, written in XML, and displays the models.

Generator

To create the model files an application (independent from the engine) will receive as parameters the graphical primitive's type, other parameters required for the model creation, and the destination file where the vertices will be stored.

In this phase the following graphical primitives are required:

- Plane (a square in the XZ plane, centred in the origin, subdivided in both X and Z directions)
- Box (requires dimension, and the number of divisions per edge, centred in the origin)
- Sphere (requires radius, slices and stacks, , centred in the origin)
- Cone (requires bottom radius, height, slices and stacks, the bottom of the cone should be on the XZ plane)

Both the box and sphere should be centred on the origin. The cone should have its base in the XZ plane.

For instance, if we wanted to create a sphere with radius 1, 10 slices, 10 stacks, and store the resulting vertices in a file named sphere.3d, and assuming our application is called *generator*, we could write on a terminal:

```
C:\>generator sphere 1 10 10 sphere.3d
```

Instruction to generate a box with a length of 2 units, where each side is divided in a grid 3x3:

```
C:\>generator box 2 3 box.3d
```

To generate a cone with radius 1, height 2, 4 slices and 3 stacks, we could write:

```
C:\>generator cone 1 2 4 3 cone.3d
```

Finally, here is an example to create a plane with 1 unit in length, and 3 divisions along each axis:

```
C:\>generator plane 1 3 plane.3d
```

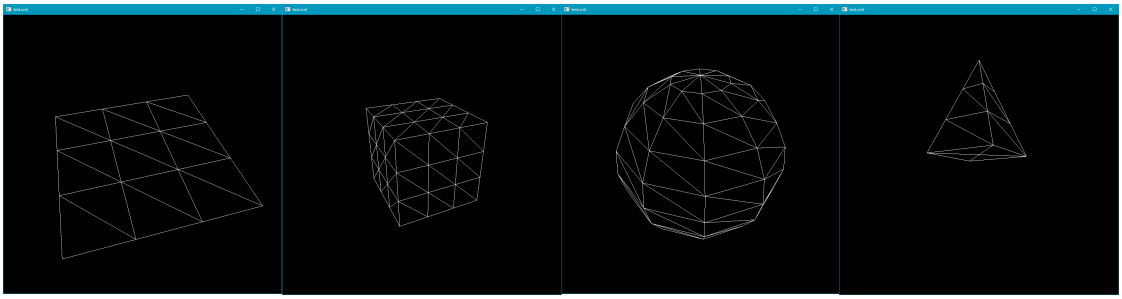
After the above commands are complete there should be a set of files with “3d” extension, each file containing all the vertices and faces required to define the respective primitive.

The file format should be defined by the students and can support additional information to assist the reading process, for example, the file may contain a line at the beginning stating the number of vertices it contains.

Engine

The engine application will receive a configuration file, written in XML. In this phase the XML file will contain the camera settings and the indication of which previously generated files to load.

The screenshots below show an example of the result for each model, created with the commands provided in the generator section.



Example of a XML configuration file for phase one:

```
<world>
    <window width="512" height="512" />
    <camera>
        <position x="3" y="2" z="1" />
        <lookAt x="0" y="0" z="0" />
        <up x="0" y="1" z="0" />
        <projection fov="60" near="1" far="1000" />
    </camera>
    <group>
        <models>
            <model file="plane.3d" />
            <model file="cone.3d" />
        </models>
    </group>
</world>
```

Note: in the above example it is assumed that the files “cone.3d” and “plane.3d” have been previously created with the generator application.

The XML file should be read only once when the engine starts. Students should define appropriate data structures to store the information of the model files in memory.

Several alternatives are available to assist the XML Reading, such as tinyXML (http://www.grinninglizard.com/tinyxml/). For other alternatives check out this discussion at stackoverflow (http://stackoverflow.com/questions/170686/what-is-the-best-open-xml-parser-for-c).

Phase 2 – Geometric Transforms

This phase is about creating hierarchical scenes using geometric transforms. Only the engine application needs to be updated. A scene is defined as a tree where each node contains a set of geometric transforms (translate, rotate and scale) and optionally a set of models. Each node can also have children nodes.

Example of a configuration XML file with a single group:

```
<world>
    <window width="512" height="512" />
    <camera>
        <position x="10" y="10" z="10" />
        <lookAt x="0" y="0" z="0" />
        <up x="0" y="1" z="0" />
        <projection fov="60" near="1" far="1000" />
    </camera>
    <group>
        <transform>
            <translate x="4"  y="0" z="0" />
            <rotate angle="30" x="0" y="1" z="0" />
            <scale x="2" y="0.3" z="1" />
        </transform>
        <models>
            <model file="cone.3d" />
            <model file="plane.3d" />
        </models>
    </group>
</world>
```

Example of a hierarchical group definition:

```
<world>
    <window width="512" height="512" />
    <camera>
        <position x="10" y="10" z="10" />
        <lookAt x="0" y="0" z="0" />
        <up x="0" y="1" z="0" />
        <projection fov="60" near="1" far="1000" />
    </camera>
    <group>
        <transform>
```

```
            <translate x="4"  y="0" z="0" />
        </transform>
        <models>
            <model file="cone.3d" />
        </models>
        <group>
            <transform>
                <translate x="4"  y="0" z="0"/>
            </transform>
            <models>
                <model file="sphere.3d" />
            </models>
            <group>
                <transform>
                    <translate x="0"  y="4" z="0"/>
                </transform>
                <models>
                    <model file="cone.3d" />
                </models>
            </group>
        </group>
    </group>
</world>
```

In the second example the children will inherit the parent's geometric transforms.

Geometric transformations can only exist inside a group, and are applied to all models and subgroups. There can only be one transformation of each type.

Note: the order of the geometric transforms is relevant.

The required demo scene for this phase is a static model of the solar system, including the sun, planets and moons defined in a hierarchy.

Phase 3 – Curves, Cubic Surfaces and VBOs

In this phase the **generator** application must be able to create a new type of model based on Bezier patches. The generator will receive as parameters the name of a file where the Bezier control points are defined as well as the required tessellation level. The resulting file will contain a list of the triangles to draw the surface.

Regarding the **engine**, we want to extend the *translate* and *rotate* elements. Considering the translation, a set of points will be provided to define a Catmull-Rom cubic curve, as well as the number of seconds to run the whole curve. The goal is to perform animations based on these curves. This transform also includes a filed "align" to specify if the object is to be aligned with the curve. The models may have either a time dependent transform, or a static one as in the previous phases. In the rotation node, the angle can be replaced with time, meaning the number of seconds to perform a full 360 degrees rotation around the specified axis.

To measure time the function `glutGet(GLUT_ELAPSED_TIME)` can be used.

```
...
<translate time="10" align="True" >
    <point x="1" y="0" z="1" />
    <point x="0.707" y="0.707" <="1”"/>
    <point x="0" y="1" z="1" />
    ...
    <point x="-1" y="0" z="1" />
</translate>
...
```

Note. Due to Catmull-Rom's curve definition it is always required an initial point before the initial curve segment and another point after the last segment. The minimum number of points is 4.

```
...
<rotate time="10" x="0" y="1" z="0" />
...
```

In this phase it is also required that models are drawn with **VBO**s, as opposed to immediate mode used in the previous phases.

The demo scene is a dynamic solar system, including a comet with a trajectory defined using a Catmull-Rom curve. The comet must be built using Bezier patches, for instance with the provided control points for the teapot.

Phase 4 – Normals and Texture Coordinates

In this phase the geometry generator application should generate texture coordinates and normals for each vertex.

In the 3D engine we must activate the lighting and texturing functionalities, as well as read and apply the normals and texture coordinates from the model files.

For instance, in the XML file we may define a textured model as:

```
  <o<models>

        <model file="sphere.3d" >
            <!-- if texture is not specified then the
                  model is rendered with color only-->
            <texture file="earth.jpg" />
            <!-- if color is not specified, use the values
                  below as default -->
            <color>
                <diffuse R="200" G="200" B="200" />
                <ambient R="50" G="50" B="50" />
                <specular R="0" G="0" B="0" />
                <emissive R="0" G="0" B="0" />
                <shininess value="0" />
            </color>
```

```
            </model>
        </models>
```

The XML must allow the definition of the diffuse, specular, emissive, and ambient colour components, as well as shininess.

In the XML file we must also be able to define the light sources. For instance:

```
era> <lights>

        <light type="point" posX="0" posY="10" posZ="0" />
        <light type="directional" dirX="1" dirY="1" dirZ="1"/>
        <light type="spotlight" posX="0" posY="10" posZ="0"
                                dirX="1" dirY="1" dirZ="1"
                                cutoff="45" />
    </lights>
```

In the above example, type can be “point”, “directional” or “spot”. Note that for the latter two types different arguments are be required. For instance a directional light requires a direction, not a position.

The required demo scene for this phase is the animated solar system with texturing and lighting.

Sites providing planet textures:

- Planetary Pixels Emporium (http://planetpixelemporium.com/planets.html)
- Solar System Scope (http://www.solarsystemscope.com/textures/)

Due dates

- See blackboard

The deadline is always at midnight. Students may submit work late, except in the last phase, with a penalty of 10% per day. Weekends count as one day.

In all phases, students are required to deliver:

- The source code, plus Visual Studio project, or makefile, or CMake file.
- All required libraries and include files from third party APIs
- Demo scenes
- A written report detailing the decisions and approaches taken during the phase development.

Assessment: Each phase: 22.5% (17.5% application, 5% written report)

Extras: 10% (examples: view frustum culling, new geometrical primitives, third person camera motion, complex demo scenes created for the engine, meaningful XML format extensions, etc...)