Practical Assignment CG - 2016/17

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 below. Each phase must be submitted via blackboard.

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 display the models.

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, made with 2 triangles)
- Box (requires X, Y and Z dimensions, and optionally the number of divisions)
- Sphere (requires radius, slices and stacks)
- Cone (requires bottom radius, height, slices and stacks)

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

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

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.

Afterwards, the engine will receive a configuration file, written in XML. In this phase the XML file will contain only the indication of which previously generated files to load.

Example of a XML configuration file for phase one:

```
<scene>
     <model file="sphere.3d" />
     <model file="plane.3d" />
</scene>
```

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).

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

The demo scenes for this phase are the XML configuration and model files to display each graphical primitive.

Phase 2 – Geometric Transforms

This phase is about creating hierarchical scenes using geometric transforms. 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:

```
<scene>
	<group>
		<translate X=5 Y=0 Z=2 />
		<rotate angle=45 axisX=0 axisY=1 axisZ=0 />
		<models>
			<model file="sphere.3d" />
		</models>
	</group>
</scene>
```

Example of a group with a child group:

```
<scene>
	<group>
		<translate X=1 />
		<models>
			<model file="sphere.3d" />
		</models>
		<group>
			<translate Y=1 />
			<models>
				<model file="cone.3d" />
			</models>
		</group>
	</group>
</scene>
```

In the second example the child group will inherit the geometric transforms from the parent.

Geometric transformations can only exist inside a group, and are applied to all models and subgroups.

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. For 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. The models may have a time dependent transform, as opposed to 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 >
      <point X=1 Y=0 Z=1 />
      <point X=0.707 Y=0.707 Z=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 the minimum number of points is 4.

```
...
<rotate time=10 axisX=0 axisY=1 axisZ=0 />
...
```

In this phase it is also required that models are drawn with VBOs, 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, and read and apply the normals and texture coordinates from the model files.

For instance:

```
	<model file="sphere.3d" texture="earth.jpg"/>
	<model file="bla.3d" diffR=0.8 diffG=0.8 diffB=0.15 />
```

In the first example above we get a textured model, in the second a coloured model (diffuse component). The XML must allow the definition of the diffuse, specular, emissive and ambient colour components.

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

```
<scene>
	<lights>
		<light type="POINT" posX=0 posY=10 posZ=0 />
	</lights>
	<group>
		<translate X=5 Y=0 Z=2 />
		<rotate angle=45 axisX=0 axisY=1 axisZ=0 />
		<models>
			<model file="sphere.3d" />
		</models>
	</group>
</scene>
```

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

Sites providing planet textures:

- NASA/JPL Solar System simulator (http://maps.jpl.nasa.gov/)
- Planetary Pixels Emporium (http://planetpixelemporium.com/planets.html)
- Solar System Scope (http://www.solarsystemscope.com/textures/)

Due dates

- Phase 1: 6th March
- Phase 2: 31th March
- Phase 3: ~~28th April~~ 1st May
- Phase 4: 19th May

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, Visual Studio project, or makefile, or CMake file.
- 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: third person camera motion, complex demo scenes created for the engine, meaningful XML format extensions, view frustum culling, etc...)

XML examples

The following examples were created to assist students in the assignment and contain both valid and invalid syntax.

Valid syntax

Simple group without children

```
<scene>
	<group>
		<translate X=5 Y=0 Z=2 />
		<rotate angle=45 axisX=0 axisY=1 axisZ=0 />
		<scale X=1 Y=2 Z=3 />
		<models>
			<model file=”sphere.3d” />
			<model file=”cone.3d” />
		</models>
	</group>
</scene>
```

Group with a single child

```
<scene>
	<group>
		<translate X=1 />
		<models>
			<model file=”sphere.3d” />
		</models>
		<group>
			<translate Y=1 />
			<models>
				<model file=”cone.3d” />
			</models>
		</group>
	</group>
</scene>
```

Groups may contain only other groups

```
<scene>
	<group>
```

```
          <group>
               <translate Y=1 />
               <models>
                    <model file=”cone.3d” />
                    <model file=”sphere.3d” />
               </models>
          </group>
          <group>
               <translate X=1 />
               <models>
                    <model file=”cone.3d” />
               </models>
          </group>
     </group>
</scene>
```

Hierarchy example with transformations

```
<scene>
     <group>
          <translate X=5 Y=0 Z=2 />
          <rotate angle=45 axisX=0 axisY=1 axisZ=0 />
          <models>
               <model file=”sphere.3d” />
          </models>
          <group>
               <translate Y=1 />
               <models>
                    <model file=”cone.3d” />
               </models>
               <group>
                    <translate X=1 />
                    <models>
                         <model file=”cone.3d” />
                    </models>
               </group>
          </group>
          <group>
               <translate X=1 />
               <models>
                    <model file=”cone.3d” />
               </models>
          </group>
     </group>
</scene>
```

Invalid examples

A group may not have multiple translations (or rotates or scales)

The geometric transformation tags affects all children, hence, should be defined in the beginning of the group.

```
<scene>
     <group>
          <translate X=5 Y=0 Z=2 />
          <group>
               <translate Y=1 />
               <models>
                    <model file=”cone.3d” />
               </models>
          </group>
          <translate X=5 Y=0 Z=2 />
          <group>
               <translate X=1 />
               <models>
                    <model file=”cone.3d” />
               </models>
          </group>
     </group>
</scene>
```

Each group can have only a single “models” tag (inside tag "models" we can have multiple "model" tags).

```
<scene>
     <group>
          <translate X=5 Y=0 Z=2 />
          <models>
               <model file=”cone.3d” />
               <model file=”sphere.3d” />
          </models>
          <rotate X=5 Y=0 Z=2 />
          <models>
               <model file=”cone.3d” />
          </models>
     </group>
</scene>
```