# *DUBLIN CITY UNIVERSITY*

## *SEMESTER TWO SOLUTIONS 2007*

| | |
|---|---|
| ***MODULE:*** ***(Title & Code)*** | 3D Graphics and Visualisation<br>EE563 |
| ***COURSE:*** | M.Eng./Grad. Dip./Grad. Cert. in Electronic Systems<br>M.Eng./Grad. Dip./Grad. Cert. in Telecommunications Engineering<br>ECSA/GDE/GDEI/MEN/MENI/MTC/MTCN |
| ***YEAR:*** | Postgraduate (C) |
| ***EXAMINERS:*** | Prof. Paul Rees<br>Dr Robert Sadleir (x8592)<br>Dr Derek Molloy (x5355) |
| ***TIME ALLOWED:*** | 3 Hours |
| ***INSTRUCTIONS:*** | Please answer FOUR questions.<br>All questions carry equal marks |

***Requirements for this paper***
***Please tick (X) as appropriate***

| | |
|---|---|
| | ***Log Table*** |
| | ***Graph Paper*** |
| | ***Attached Answer Sheet*** |
| | ***Statistical Tables*** |
| | ***Floppy Disk*** |
| | ***Actuarial Tables*** |

**THE USE OF PROGRAMMABLE OR TEXT STORING CALCULATORS IS EXPRESSLY FORBIDDEN**

**Please note that where a candidate answers more than the required number of questions, the examiner will mark all questions attempted and then select the highest scoring ones.**

**PLEASE DO NOT TURN OVER THIS PAGE UNTIL YOU ARE INSTRUCTED TO DO SO**

**Question 1**

(a) Answer all of the following short questions (make sure to keep your answers concise):

(i) What two node component objects does a **`Shape3D`** object maintain references to? Explain what the referenced node components represent.

A **`Shape3D`** object maintains references to an **`Appearance`** node component and at least one **`Geometry`** node component.

The **`Appearance`** node component is a container that maintains references to several appearance components that define how the Shape3D object appears when it is rendered. Appearance components referenced by an **`Appearance`** node component include: **`ColouringAttributes`**, **`PointAttributes`**, **`LineAttributes`**, **`Material`**, **`Texture`** and **`TextureAttributes`**.

The **`Geometry`** node component defines the structure of the shape and typically consists of an array of vertices that represent individual points, lines, triangles or quads. A Geometry node component can also represent a **`Raster`** image or a **`Text3D`** object.

**[2 marks]**

(ii) A sphere has a radius of 1 metre and colouring attributes with a red colour (1.0, 0.0, 0.0). The sphere is in the presence of linear fog with a back distance of 25 metres, a front distance of 5 metres and a colour of mid grey (0.5, 0.5, 0.5). If the sphere is positioned so that its centre is 10 metres away form the viewer then what colour does the closest point on the sphere appear to the viewer?

The distance from the closest point on the sphere to the viewer is 9 metres.
The effect of the fog increases linearly from a distance of 5 metres to a distance of 25 metes.
A 9 metres the effect of the fog is (9 – 5) / (25 – 5) = 4/20 = 20%
The colour of the closest point is therefore:
(1.0, 0.0, 0.0) x 0.80 + (0.5, 0.5, 0.5) x 0.20 =
(0.8, 0.0, 0.0) + (0.1, 0.1, 0.1) =
(0.9, 0.1, 0.1)

**[2 marks]**

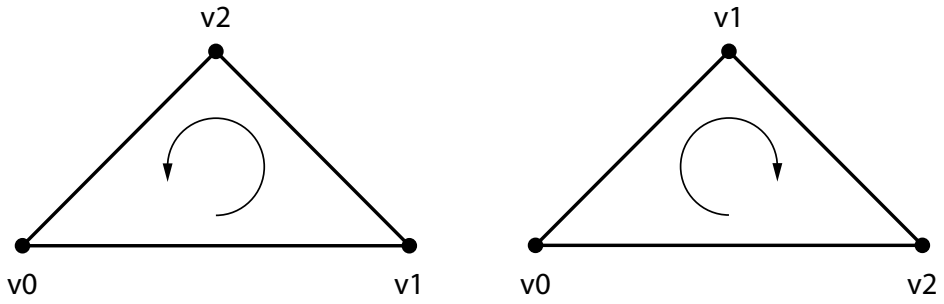
(iii) What are the texture coordinates at the top left hand corner of a texture image with a width of 128 pixels and a height of 64 pixels. Explain your answer.

The horizontal and vertical texture coordinates of a texture image are always in the range [0.0 - 1.0]. This is the case no matter what the dimensions of the image are and no matter what the aspect ration of the image is. Consequently, the top left hand corner of a 128 x 64 pixel texture image will have texture coordinates (1.0, 1.0). Note that it will not have texture coordinates (1.0, 0.5).

**[2 marks]**

(iv) How do you identify the front face of a polygon? Give an example of each. Why does Java 3D differentiate between front facing and back facing polygons?

The front face of a polygon is where the order of the vertices makes a anti-clockwise loop and the back face of a polygon is where the order of the vertices makes a clockwise loop. This is illustrated below.




Java 3D differentiates between back and front facing polygons in order to improve rendering efficiency. In the majority of cases the back face of a polygon is not visible to the viewer and consequently it is not rendered to improve rendering efficiency.

**[2 marks]**

(v) What is the difference between direct volume rendering and indirect volume rendering? Give examples of each approach to support your answer.

Direct volume rendering involves generating a 2D representation of a 3D volume directly from the voxels of the volumetric data set. An example of this would be ray casting.

Indirect volume rendering involves extracting a polygonal mesh representation of an isosurface from a 3D volumetric data set and then rendered the extracted isosurface using conventional 3D surface rendering techniques. The polygonal mesh representation of the isosurface can be extracted using the marching cubes algorithm.

**[2 marks]**

(vi) If an ambient light with a cyan colour (0.18, 0.80, 0.87) illuminates an object with a purple ambient colour (0.65, 0.23, 0.66) what colour will the object appear to be in the rendered scene?

The reflected light will be a product of the ambient colour of the material and the colour of the ambient light.

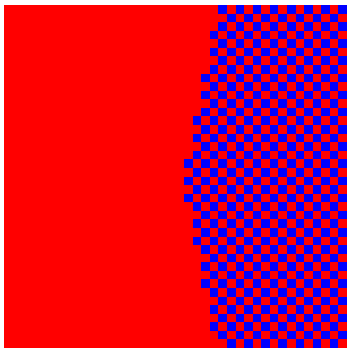
(0.18, 0.80, 0.87) x (0.65, 0.23, 0.66) =
(0.12, 0.18, 0.57) = Dark blue

The reflected colour in specified lighting conditions will be dark blue.

**[2 marks]**

(vii) What is the difference between the **SCREEN_DOOR** and **BLENDED** modes defined by the **TransparencyAttributes** appearance component?

The **SCREEN_DOOR** mode creates gaps in the foreground colour to simulate different levels of transparency. The **BLENDED** mode blends the foreground and background colours to generate a more accurate and visually appealing result. These two modes are illustrated below for a transparency setting of 50%.



**SCREEN_DOOR** – 50%

**BLENDED** - 50%

**[2 marks]**

(b) A line is defined by its two endpoints. Using the equation below rotate the 2D line from (-3, -7) to (2, 2) by 30 degrees about the origin. Note that a positive angle represents anticlockwise rotation about the origin.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Briefly describe two other transformations that can be applied to a 2D image.

cos(30) = 0.866
sin(30) = 0.5

**First point (-3, -7)**

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 0.866 & -0.5 & 0 \\ 0.5 & 0.866 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -3 \\ -7 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} (0.866 \times -3) + (-0.5 \times -7) + (0 \times 1) \\ (0.5 \times -3) + (0.866 \times -7) + (0 \times 1) \\ (0 \times -3) + (0 \times -7) + (1 \times 1) \end{bmatrix}$$

**Second point (2, 2)**

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 0.866 & -0.5 & 0 \\ 0.5 & 0.866 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} (0.866 \times 2) + (-0.5 \times 2) + (0 \times 1) \\ (0.5 \times 2) + (0.866 \times 2) + (0 \times 1) \\ (0 \times 2) + (0 \times 2) + (1 \times 1) \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -2.598+3.5+0 \\ -1.5-6.062+0 \\ 0+0+1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 0.902 \\ -7.562 \\ 1 \end{bmatrix}$$

*(x', y') = (0.902, -7.562)*

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1.732-1+0 \\ 1+1.732+0 \\ 0+0+1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 0.732 \\ 2.732 \\ 1 \end{bmatrix}$$

*(x', y') = (0.732, 2.732)*

The endpoints of the line are located at (0.902, -7.562) and (0.732, 2.732).

Two other examples of transformations that can be applied to a 2D image are **scale** and **translate**. Scaling a 2D image involves multiplying its horizontal and vertical dimensions by a scale factor. A scale factor greater than 1.0 increases the size of the image and a scale factor less than 1.0 decreases the size of the image. Translating an image involves moving the origin by the specified distance in the horizontal and vertical directions.

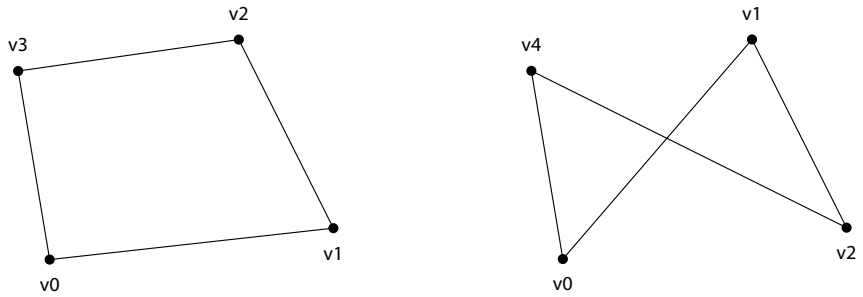
**[5 marks]**

(c) What are the two rules that relate to the definition of polygons in VRML? Describe in detail the operation of the VRML code listed below and provide a wire frame illustration of the expected outcome.

```
#VRML V2.0 utf8

Shape
{
appearance Appearance
       {
              material Material
              {
                     diffuseColor 0 0 1
              }
       }
    geometry IndexedFaceSet
    {
       coord Coordinate
        {
            point [ -1 -1 -1,
                     1 -1 -1,
                    -1  1 -1,
                    -1 -1  1,
                    -1  1  1,
                     1 -1  1,
                     1  1 -1,
                     1  1  1 ]
        }
        coordIndex [ 0, 3, 4, 2, -1,
                     0, 1, 5, 3, -1,
                     0, 2, 6, 1, -1,
                     7, 5, 1, 6, -1,
                     7, 6, 2, 4, -1,
                     7, 4, 3, 5, -1 ]
    }
}
```

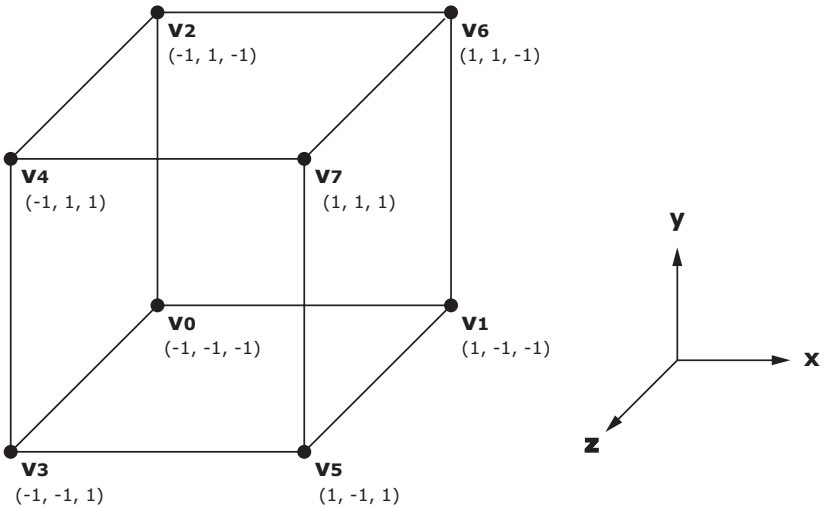
A polygon in VRML must be convex and coplanar. Convex means that there are no bays or convexities in the boundary of the polygon. Examples of convex and non convex polygons are illustrated below.




It should be noted that it is possible to break a convex polygon down into two or more non-convex polygons.

The VRML file defines a single **`Shape`** node. The **`shape`** node has an **`appearance`** and a **`geometry`** associated with it. The **`appearance`** defines a **`Material`** with a diffuse colour of red (1.0, 0.0, 0.0). The geometry is represented by an **`IndexedFaceSet`**. This means that the geometry will be a set of one or more faces where the vertices for a face are defined as indices into a vertex array. In this case, the array of vertices holds 8 points organised in a cubic formation about the origin and the array of vertices defines the six faces of a cube where the vertices are orientated so that the exterior of the cube consists only of front facing polygons.

A wire frame representation of the expected outcome from the VRML file is illustrated below:




**[6 marks]**

**Question 2**

(a) Briefly describe the functionality of the following subclasses of the **`Group`** class:

- **`BranchGroup`**
- **`OrderedGroup`**
- **`TransformGroup`**
- **`ViewSpecificGroup`**

**`BranchGroup:`**

A **`BranchGroup`** object serves as a pointer to the root of a scene graph branch. **`BranchGroup`** objects are the only objects that can be attached to, or removed from, a **`Locale`**. The main method defined by a **`BranchGroup`** is the **`compile()`** method. This causes the branch graph represented by the **`BranchGroup`** object to be converted to an optimised internal representation. Once the **`compile()`** method has been called, only changes that have been explicitly enabled can be made to the scene graph.

**[2 marks]**

**`OrderedGroup:`**

The **`OrderedGroup`** node is a node that ensures its children are rendered in a specific order. In addition to the list of children inherited from the base **`Group`** class, the **`OrderedGroup`** class also maintains an integer array of child indices that indicates the rendering order for its children.

**[2 marks]**

**`TransformGroup:`**

The **`TransformGroup`** class represents a group node that implements a 3D spatial transformation that can position, orient and scale all of its children. The transformation is represented by a **Transform3D** object.

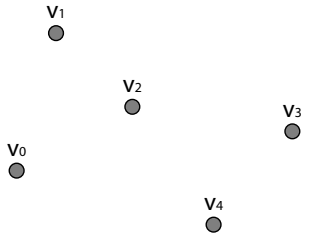
**[2 marks]**

**`ViewSpecificGroup:`**

The **`ViewSpecificGroup`** node is a Group whose descendants are rendered only on a specified set of views. It contains a list of view on which its descendants are rendered. Methods are provided to add views, removes views and enumerate the list of view maintained by this node. The list of views is initially empty. This means that by default, the children of this group will not be rendered on any view.

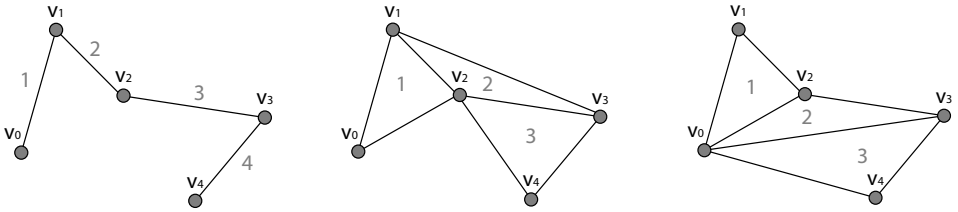
**[2 marks]**

(b) Use the following set of vertices to define a line strip array, triangle strip array and a triangle fan array. A single strip should be generated for each type of geometry and each of the triangles should be numbered in the case of the triangular geometries.




What is the main benefit of using strip geometry over regular geometry?

The following diagrams illustrated the line strip array, triangle strip array and triangle fan array geometries.

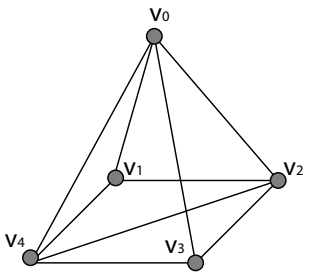



The use of strip geometry reduces the amount of vertex reuse when defining continuous geometry. In the line strip array example above, the definition of separate line segments would require eight vertices, whereas the definition of a strip required only 5 vertices. In the two triangle examples above, the definition of separate triangles would have required 9 vertices, whereas the definition of a triangle fan or strip required only 5 vertices.

**[4 marks]**

(c) Describe, in relation to the piece of geometry illustrated below, the difference between the following subclasses of **`GeometryArray`**:

- **`TriangleArray`**
- **`IndexedTriangleArray`**




In both cases, calculate the number of bytes required to store each version of the geometry. Given that a Java **`float`** primitive required 4 bytes and a Java **`int`** primitive requires 4 bytes. Assume that the vertices are defined using **`Point3f`** object.

A **`TriangleArray`** defines each triangle as a set of three vertices. Therefore the geometry illustrated above will require a total of 3x6 or 18 vertices to be defined. If each vertex consists of three floating point values then the total memory requirement for the definition of this geometry is 18x4x3 or 216 bytes.

An **`IndexedTriangleArray`** defines each triangle as a set of indices into a vertex array. Using this approach each vertex is defined only once and subsequently referenced by an integer index. Therefore the geometry illustrated above will require a total of 3x6 or 18 indices to be defined. The number of unique vertices is 5 therefore 3x5 or 15 floating point values will also have to be defined. The total memory requirement is 15x4 bytes for the indices and 18x4 bytes for the vertices, i.e. a total of 132 bytes.

It is clear that the **`IndexTriangleArray`** is more efficient than the **`TriangleArray`** when there is a high degree of vertex reuse.

**[5 marks]**

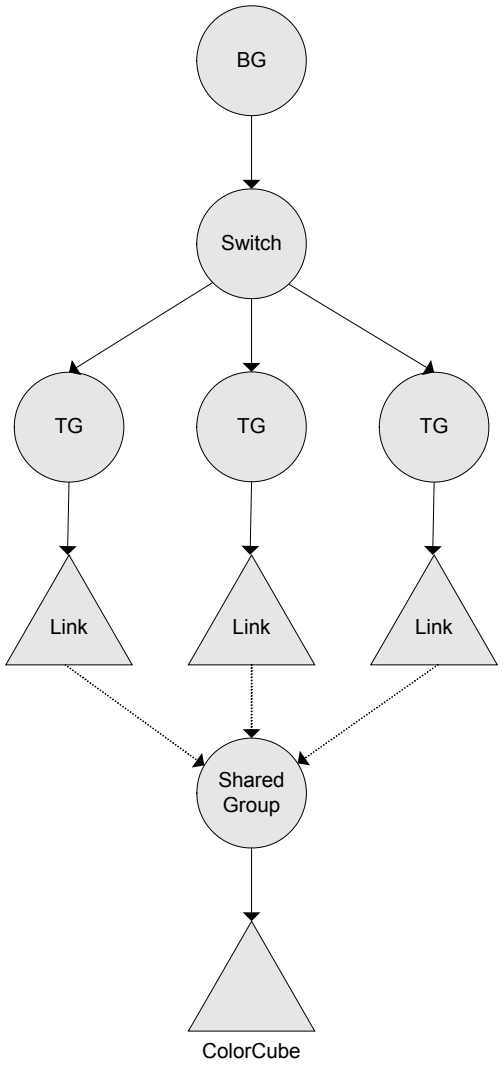
(d) Draw the **`BranchGroup`** rooted scene graph represented by the code listed below. Explain the operation of the code with reference to the scene graph and draw an illustration of the expected outcome when the code is executed.

```
 1. public BranchGroup createContentBranch()
 2. {
 3.   BranchGroup root = new BranchGroup();
 4.
 5.   Switch switchGroup = new Switch(Switch.CHILD_MASK);
 6.   BitSet mask = new BitSet();
 7.   mask.set(0);
 8.   mask.set(2);
 9.   switchGroup.setChildMask(mask);
10.   root.addChild(switchGroup);
11.
12.   SharedGroup sharedGroup = new SharedGroup();
13.   ColorCube cube = new ColorCube(0.2);
14.   sharedGroup.addChild(cube);
15.
16.   Transform3D t1 = new Transform3D();
17.   t1.setTranslation(new Vector3f(-0.25f, -0.25f, 0.0f));
18.   TransformGroup tg1 = new TransformGroup(t1);
19.   Link link1 = new Link();
20.   link1.setSharedGroup(sharedGroup);
21.   tg1.addChild(link1);
22.   switchGroup.addChild(tg1);
23.
24.   Transform3D t2 = new Transform3D();
25.   t2.setTranslation(new Vector3f(0.25f, -0.25f, 0.0f));
26.   TransformGroup tg2 = new TransformGroup(t2);
27.   Link link2 = new Link();
28.   link2.setSharedGroup(sharedGroup);
29.   tg2.addChild(link2);
30.   switchGroup.addChild(tg2);
31.
32.   Transform3D t3 = new Transform3D();
33.   t3.setTranslation(new Vector3f(0.0f, 0.25f, 0.0f));
34.   TransformGroup tg3 = new TransformGroup(t3);
35.   Link link3 = new Link();
36.   link3.setSharedGroup(sharedGroup);
37.   tg3.addChild(link3);
38.   switchGroup.addChild(tg3);
39.
```

```
40.     return root;
41.   }
```

**[8 marks]**

The following illustration represents the BranchGroup rooted scene graph represented by the code listed above.




The program begins by defining a **`BranchGroup`** object that represents the root of the content branch of the scene graph. The first child of the **`BranchGroup`** is a **`Switch`** group. The children of the **Switch** group to be rendered are represented by a 3-bit mask. The specified mask values indicate that only the first and the last of the three children are to be rendered.

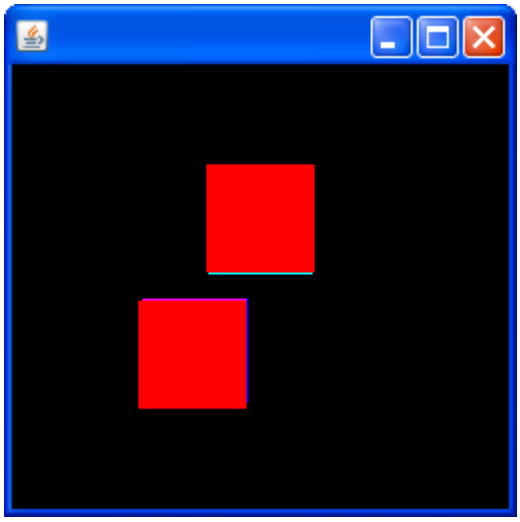
A **`SharedGroup`** is then defined with a single child which is a **`ColorCube`** object with sides 40 cm in length.

Three **`TransformGroup`** objects are then defined. Each one implements a translation that translates its children to the bottom left, bottom right and top centre respectively. The single child of each **`TransformGroup`** is a **`Link`**

leaf node and each **`Link`** references the **`SharedGroup`** defined earlier. These references are indicated by dashed lines in the scene graph.

The child mask specified for the Switch group indicates that only the first and the last child of the Switch group will be rendered, i.e. the **`ColorCube`** at the bottom left and the **`ColorCube`** at the top centre.

The expected outcome of the program is illustrated below:



**Question 3**

(a) Describe the approach to texture mapping implemented by Java 3D.

A texture image has texture coordinates that range from zero to one in the horizontal and vertical directions. The bottom left corner of the texture image has texture coordinates (0, 0) and the top right corner of the texture image has texture coordinates (1.0, 1.0). 2D texture coordinates are also assigned to the vertices of the geometry to indicate how the texture should be applied to the geometry. Texture mapping stretches the texture to make the texture locations specified by the texture coordinates line up with the texture coordinates assigned to the vertices of the geometry being texture mapped.

What is the difference between the following two boundary modes defined by the **`Texture`** class?

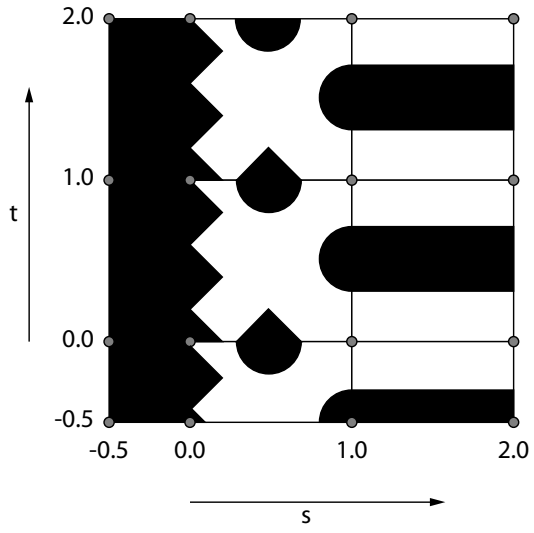
- **`CLAMP`**
- **`WRAP`**

**`CLAMP`** - clamps texture coordinate to be in the range [0, 1]. Texture boundary texels are used for values that fall outside this range.

**`WRAP`** - repeat the texture by wrapping texture coordinates that are outside the range [0, 1]. Only the fractional portion of the texture coordinates will be used here. The integer portion in discarded, e.g. 1.5 would become 0.5.

Illustrate the effect of texture mapping on the following piece of geometry if the horizontal boundary mode is set to **`CLAMP`** and the vertical boundary mode is set to **`WRAP`**. Use the texture image that is provided in your illustration.

Here is an illustration of the texture mapped geometry




**3D geometry with 2D texture coordinates**

Describe one method provided by Java 3D for automatically generating texture coordinates from geometry.

The **`OBJECT_LINEAR`** texture generation mode of the **`TexCoordGeneration`** class can be used to do this - texture coordinates are generated as a linear function of the object coordinates, i.e. in the case of 2D texture coordinates the (s, t) texture coordinates are obtained directly from the (x, y) vertex coordinates.

**[9 marks]**

(b) Describe the operation of the ray casting approach to volume rendering.

The ray casting process involves projecting pixels ray from the viewpoint through the view plane and into the volume. The path is traced back along the pixel ray to the view plane and the colour of the a pixel in the view plane is recursively calculated by combining the colour and opacity values of consecutive voxels along its associated pixel ray.

Calculate the value of the shaded pixel in the 2-D view plane using this equation:

$$c_{out} = c_{in}(1-\alpha) + C\alpha$$

The colour and opacity values for the voxels numbered one to four are:

| Voxel | Colour (c) | Opacity (α) |
|---|---|---|
| 1 | 18 | 0.06 |
| 2 | 15 | 0.03 |
| 3 | 228 | 0.92 |
| 4 | 42 | 0.13 |

- Voxel 4
  - Cin = 0, C = 42, Alpha = 0.13
  - Cout = 42 x 0.13 = 5.46
- Voxel 3
  - Cin = 5.46, C = 228, alpha = 0.92
  - Cout = 5.46 x 0.08 + 228 x 0.92
  - Cout = 0.44 + 209.76 = 210.2
- Voxel 2
  - Cin = 210.2, C = 15, alpha = 0.03
  - Cout = 210.2 x 0.97 + 15 x 0.03
  - Cout = 203.89 + 0.45 = 204.34
- Voxel 1
  - Cin = 204.34, C = 18, alpha = 0.06
  - Cout = 204.34 x 0.94 + 18 x 0.06
  - Cout = 192.08 + 1.08 = 193.16

Therefore, the value of the shaded pixel in the view plane is **193.16**. This value is most heavily influenced by voxel number 3 along the pixel ray.

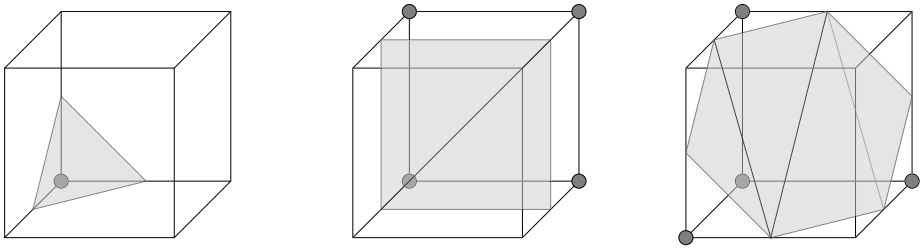
**[8 marks]**

(c) Describe the operation of the marching cubes algorithm.

The marching cubes algorithm begins by thresholding the data set, assigning a 1 to voxels $\geq d_{iso}$ (inside the isosurface) and a 0 to voxels $< d_{iso}$ (outside the isosurface). A cubic mask of size 2 x 2 x 2 is then passed through the volume and at each mask location the configuration of the eight underlying voxels is examined and the relevant surface patches are generated.

Draw the surface patches that correspond to the following voxel configurations according to the marching cubes algorithm. Note that a corner sphere indicates the presence of a voxel inside the isosurface, whereas the omission of a corner sphere indicates the presence of a voxel outside the isosurface.

These are the surface patches for the 3 voxel configurations:



Discuss one of the problems associated with the standard marching cubes algorithm and suggest a possible solution to this problem.

**Problem:** The standard MCA does not generate airtight surfaces. In certain cases holes may be inadvertently introduced into the generated mesh.

**Solution:** The holes are due to ambiguous cases resulting from mismatches between the surface patches of adjoining cubes. The ambiguous cases that result in unwanted holes are a direct result of the use of complementary cases in the standard MCA to reduce the number of core cube configurations that must be specified. By disregarding complementary cases and using only rotation to identify equivalent cube configurations the number of core configurations increases from 15 to 23.

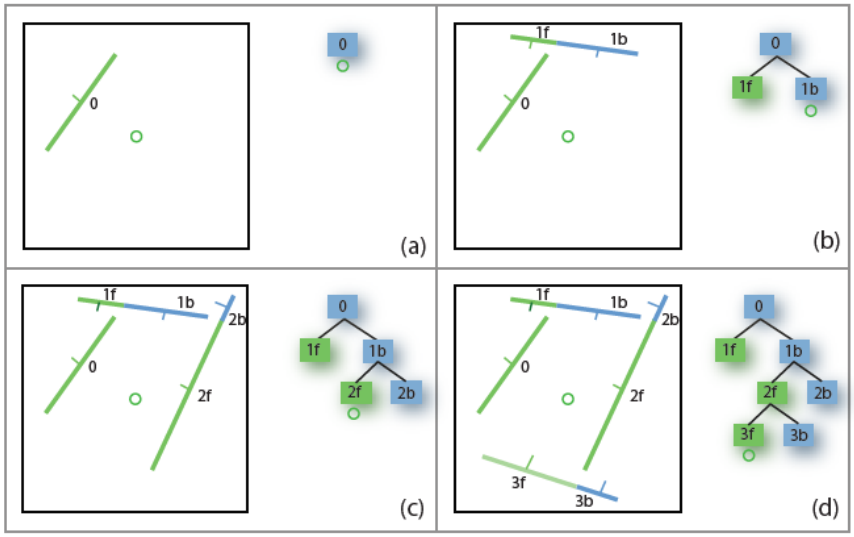
**[8 marks]**

**Question 4**

(a) Describe the Binary Space Partitioning (BSP) approach and the use of BSP Trees as space subdivision management structures.

[6 marks]

Binary Space Partitioning (BSP) is a technique for recursively subdividing a 3-D space into two non-overlapping regions using a plane, referred to as a hyperplane. Any point in 3-D space lies within only one of these regions. BSP is a hierarchal approach where the space that is divided can be further subdivided using the same space partitioning approach until some condition is met, resulting in a space-partitioning tree, which is particularly useful when building techniques for dealing with hidden surface removal. The basic properties of BSPs are that objects on one side of a hyperplane cannot intercept an object on the other side; and given a particular view point objects on the same side of the hyperplane are closer than objects on the other side.




This 2-D example illustrates the sample creation of a BSP Tree using lines to create the tree. Each line represents a partition plane (a) creates the partition plane and thus the root node, (b) through (d) illustrates the addition of further planes, with f representing the front side and b representing the back side.

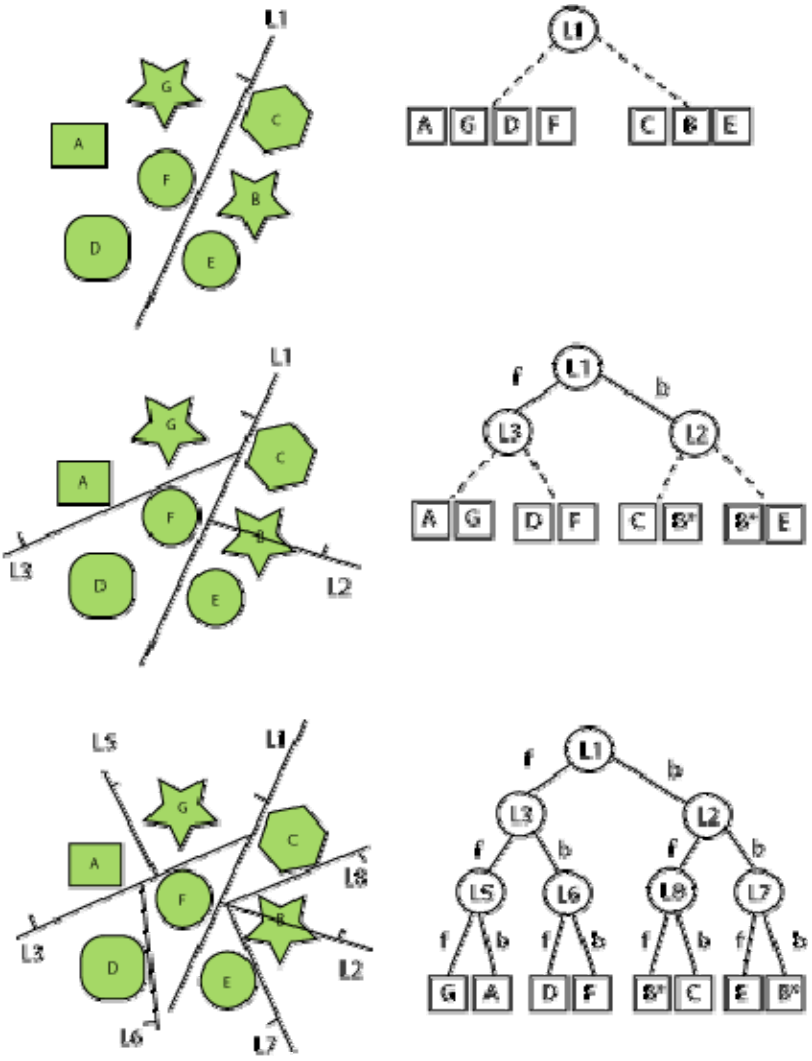
(b) Outline the general algorithm used in building BSP Trees.

[3 marks]

The algorithm to build a BSP tree is:
• Select a partition plane - The choice of planes is application dependent, but often axis aligned. In an ideal situation this will result in a balanced tree, but a poor choice will result in a large number of splits and an increase in the number of polygons. There is usually a trade-off between a well-balanced tree and a large number of splits.
• Segment the current set of polygons using the chosen plane - If a polygon lies entirely to one side or other of the plane then it is not modified and is added to the partition set for the side that it is on. If the polygon spans the partition plane then it is split into two pieces, which are added to the set on the correct side of the plane.
• Repeat again using the new sets of polygons - The termination condition is a application specific, often based on a maximum number of nodes in a leaf node, or maximum tree depth.

(c) Apply the algorithm outlined in (b) to step-by-step subdivide the scene as illustrated in Figure 4.1(a) and to build a representative BSP Tree.

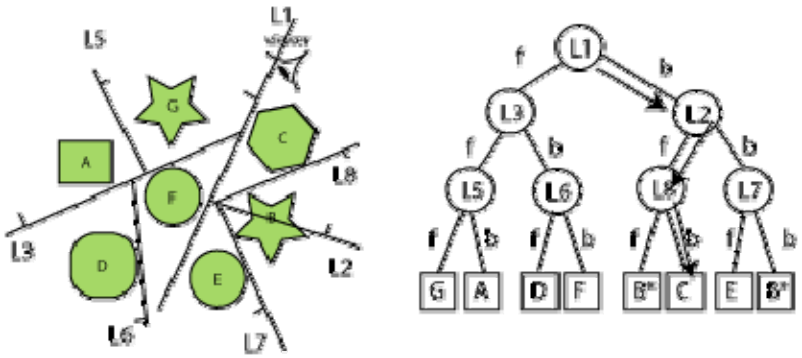
[6 marks]




(d) Why are BSP Trees useful when building hidden surface removal algorithms? Using the BSP Tree that you built in (c) illustrate how you could perform a search to find the visibility order of all of the objects in the scene if the observer was placed in the top right-hand corner, facing towards the centre of the object cluster (as illustrated in Figure 4.1(b)).

One reason that BSP trees are appropriate for hidden surface removal algorithms is that splitting difficult polygons can be an automated part of tree construction. To draw the contents of the tree you can perform a back to front tree traversal, which begins at the root node and classifies the eye point with respect to its partition plane. We draw the subtree at the far child from the eye, then the polygons in this node, followed by the near subtree. This is repeated recursively for each subtree.

We can do this by traversing the tree (rather than calculating the Euclidean distance to every object in the scene). Starting at the root node, is the observer in front of, or behind L1? In this case the observer is behind L1, and so we continue down the tree.

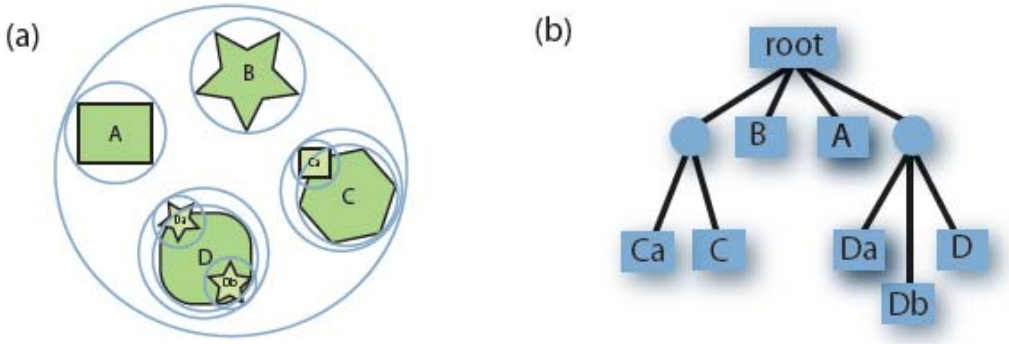



In this case we traverse to L2, where the viewer is in front of L2, so then to L8 and repeating on to object C, where the viewer is behind L8. We can then traverse backwards through the tree to give a visibility order of: C B* B* E G A F D

(e) Compare and contrast the BSP approach to the Bounding Volume Hierarchy (BVH) approach. How does the use of these approaches differ?

[4 marks]

A Bounding Volume Hierarchy (BVH) is a tree of bounding volumes where the root node includes every object in the scene and at the leaf nodes each bounding volume is just large enough to contain each scene object. The tree takes on the same hierarchical shape as the scene graph. Once again, we can quickly determine if an object is in a particular region of space using its bounding volume, but the hierarchy also allows us to determine if the segmented objects' bounding volumes contained in the child nodes are also within this region of space. The tree like structure allows all these tests to be performed very quickly. It is common practice for us to use the same object-oriented tree structure for the scene graph and also for the bounding volume hierarchy. This figure illustrates the BVH approach.




The BVH Approach is a hierarchial approach, which may require a priori knowledge of the structure of the objects in the scene. BSP does not rely on any higher a priori knowledge, rather it partitions space entirely. BSP can segment individual objects into collections of polygons depending on the way that the space is segmented.

## Question 5

(a) OpenGL has two matrix modes GL_MODELVIEW and GL_PROJECTION. Describe the use of these two matrix modes.

[4 marks]

The GL PROJECTION is a matrix transformation that is applied to every vertex that comes after it and GL MODELVIEW is a matrix transformation that is applied to every vertex on a particular model. The GL PROJECTION matrix should contain only the projection transformation calls it needs to transform eye space coordinates into clip coordinates - i.e.

think of the projection matrix as describing the attributes of the camera, such as the focal length, field of view etc. The GL MODELVIEW matrix, as its name implies, should contain modelling and viewing transformations, which transform object space coordinates into eye space coordinates.

(b) Describe the OpenGL *retained mode* and *immediate mode* with reference to the OpenGL client/server model. Discuss the advantages and disadvantages of each mode. Use a short segment of pseudo-code to outline how you would define and use a display list in OpenGL.

[8 marks]

OpenGL has an immediate mode where as soon as our C++ program executes a statement that defines a primitive (or indeed vertices, attributes, viewing information etc.), the primitive is sent immediately to the graphics card server for display. When the scene needs to be redrawn, as in our rotating sphere example, then the vertices defining the sphere must be resent to the server. Clearly, this will involve sending large amounts of data between your C++ client application and the 3-D graphics server.

OpenGL also provides retained mode graphics, which provides us with display lists. As discussed, we define the object once and place it in a display lists. Since display lists are part of the server state, therefore residing on the 3-D graphics server, the cost of repeatedly sending vertex information is dramatically reduced. Some graphics hardware may store display lists in dedicated memory or may store the data in an optimised form that is more compatible with the graphics hardware. There are some disadvantages with display lists; display lists require memory on the server and there is a small overhead in creating the display lists.

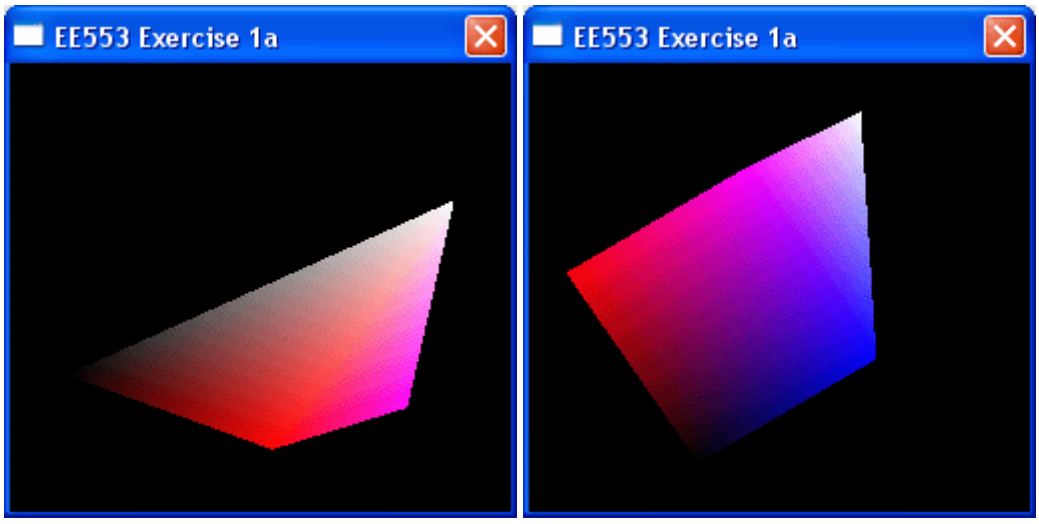
```
#define SPHERE 1 // An identifier for a sphere

void defineGLSphere(GLfloat radius, GLfloat step)
{
        glNewList(SPHERE, GL COMPILE); // define a sphere
                for (GLfloat phi=−80.0f; phi<80.0; phi+=step)
                { ...
                        glBegin(GL QUAD STRIP);
                ...
                glEnd();
                }
                // Close one end
                glBegin(GL TRIANGLE FAN);
                …
                glEnd();
                // Close the other
                glBegin(GL TRIANGLE FAN);
                ...
                glEnd();
        glEndList(); // end sphere definition
}

// Called to draw the scene
int drawGLScene(float theta)
{
        ...
        glCallList (SPHERE); // actually draw the sphere
        ...
}
```

(c) Draw and describe the output of this segment of C++ OpenGL code:

[8 marks]




The code will generate a pyramid, with a square base and an apex that is offset to be directly above one of the vertices of the underlying base. The colour of the apex should be white and blended to the base where colours will blend from red, to magenta, to blue to black.

(d) What functions does the OpenGL stack perform? Use a short segment of pseudo-code to describe how you would use this stack to place objects in a scene.

[5 marks]

Because we need to apply many transformations to different objects and vertices in using OpenGL, we have to be careful that we only apply these transformations to the correct objects and vertices. This can be a difficult problem, but fortunately OpenGL has provided us with the matrix and attribute stacks. These stacks allow us to store the current state, by pushing it onto the stack; change the state to some other value, perform some operations and finally to restore the original state. We can do this by pushing and popping them from the stack. It is good practice to push both the current matrices and attributes onto the their correct stacks when we enter a display list, and to pop them off when we are exiting the list. For example, this piece of code draws a blue sphere at (1,0,0) and a red sphere at (-1,0,0).

```
{
        glPushMatrix();
                glColor3f(0.0 f , 0.0 f , 1.0 f ); // blue
                glTranslatef (1.0 f , 0.0 f , 0.0 f );
                glCallList (SPHERE);
        glPopMatrix();

        glPushMatrix();
                glColor3f(1.0 f , 0.0 f , 0.0 f ); // red
                glTranslatef (−1.0f, 0.0 f , 0.0 f );
                glCallList (SPHERE);
        glPopMatrix();
}
```

**Question 6**

(a) Describe the following terms, which are used to describe shaded surfaces: *Specular*, *Translucent*, and *Diffuse*.

[3 marks]

- Specular surfaces - These surfaces appear shiny as the light that is reflected is maintained within a narrow range of angles, close to the angle of reflection. Mirrors are perfect specular surfaces.
- Translucent surfaces - These surfaces allow some of the light to penetrate the surface and to emerge from some other location on the object. For example, refraction in glass or water would cause the light to emerge from another location on the object.
- Diffuse surfaces - These surfaces are characterised by having light scattered in all directions; for example, walls painted with a matte paint are diffuse reflectors.

(b) Describe the Phong Reflection Model. What is the Lambertian Surface Model and how does the intensity of reflected light vary as the angle to the light and viewer changes with respect to the surface normal?

[7 marks]

The Phong Reflection model provides a good approximation to physical reality, producing good renderings under varying lighting conditions and materials. The Phong model uses four vectors to calculate the colour at a particular point P on a surface; these are **n**, the normal vector at that point on the surface; **v**, which is in the direction from point P to the viewer (or centre of projection); **l**, the direction of a line from P to a point light source; and **r** is the direction that a perfectly reflected ray from **l** would take. The Phong model supports the three types of material-light interaction of ambient, diffuse and specular. OpenGL works by assuming that if there is a set of point sources that each source can have separate red, green and blue ambient, diffuse and specular components.

Diffuse relections are characterised by rough surfaces, where rays of light that strike the surface are reflected back at quite different angles. Perfectly diffuse surfaces are called Lambertian Surfaces and can be modelled by Lambert's law, which states that: $R_d \propto \cos\theta$, where theta is the angle between the normal at the point of interest **n** and the direction of the light source **l**. If only a fraction of incoming light is reflected we can add in a reflection coefficient $k_d$ (where $0 < k_d < 1$) we can write: $R_d = k_d L_d \cos$ theta.

(c) Using C++ pseudo-code, write a generic container class for a Scene Graph Tree that is capable of storing and identifying scene graph elements, such as lights, objects, cameras etc.

[10 marks]

```
#include<vector>  // Use the STL Vector as our container

enum RTTI_OBJECT_TYPE
{
        RTTI_CAMERA,     //does not exist yet
        RTTI_LIGHT,      //does not exist yet
        RTTI_DUMMY,
};

class SceneObject
```

```
{

protected:
        SceneObject*                                    parentObject;
        std::vector<SceneObject*>               childrenObjects;

public:
        SceneObject();
        virtual ~SceneObject();

        // assessors/mutators
        void setParent(SceneObject* parent)     { parentObject = parent; }
        void addChild(SceneObject* child);
        std::vector<SceneObject*>* getChildrenObjects() { return &childrenObjects; }
        virtual RTTI_OBJECT_TYPE getType() const = 0;

        // Force every child to have a render and update methods
        virtual void render(float timeElapsed) = 0;
        virtual void update(float timeElapsed) = 0;
};

void SceneObject::addChild(SceneObject *child)
{
        if (!child) { std::cerr << "Attempt to add invalid child to scene graph."; }

        child->setParent(this);
        childrenObjects.push_back(child);
}
```

(d) Using C++ pseudo-code, write an algorithm for traversing your scene graph tree containers from part (c) in a recursive manner.

[5 marks]

```
class SceneObject;  //avoid a circular definition

class SceneGraph
{
  public:

        SceneGraph();
        virtual ~SceneGraph();

        void updateScene(SceneObject* sceneObject, float timeElapsed);
};


void SceneGraph::updateScene(SceneObject* sceneObject, float timeElapsed)
{
        if (!sceneObject)
        {
                std::cerr << "Attempt update of object not on the scene graph.";
                return;
        }
        else
        {
                sceneObject->update(timeElapsed);
                // and call all the children
```

```
		std::vector<SceneObject*>::iterator it = sceneObject->getChildrenObjects()->begin();
		for (; it!=sceneObject->getChildrenObjects()->end(); ++it)
		{
			updateScene(*it, timeElapsed);
		}
	}
}
```