DUBLIN CITY UNIVERSITY

SEMESTER TWO SOLUTIONS 2011

MODULE: EE563 Graphics and Visualisation

COURSE: MEN – M.Eng. in Electronic Systems
MTC – M.Eng. in Telecommunications Engineering

YEAR: Postgraduate (C)

EXAMINERS: Prof. Peter Ashburn (External Examiner)
Dr Robert Sadleir, Section A, Ext no. 8592
Dr Derek Molloy, Section B, Ext no. 5355

TIME ALLOWED: 3 Hours

INSTRUCTIONS: Please answer any TWO questions from Section A and any TWO questions from Section B. All questions carry equal marks.

Please do not turn over this page until you are instructed to do so

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

Section A

Q1 (a)

(i) **`AxisAngle4f:`** A four-element axis angle represented by single-precision floating point x,y,z,angle components. An axis angle is a single precision rotation of angle (radians) about the vector (x,y,z)
`Vector3d:` A 3-element vector that is represented by double-precision floating point x,y,z coordinates. If this value represents a normal, then it should be normalized.
[2 marks]

(ii) The ViewPlatform is moved back along Z so that objects at the origin spanning the normalized X range of -1.0 to +1.0 can be fully viewed across the width of the window. This is done by setting a translation of 1/(tan(fieldOfView/2)) in the ViewPlatform transform.
[2 marks]

(iii) **`Gouraud`**: This shading model interpolates the colour at each vertex across the primitive (e.g. line, polygon, quad) and consequently the primitive is drawn with many different colours.
`Flat:` This shading model does not interpolate color across the primitive. The primitive is drawn with a single color and the color of one vertex of the primitive is duplicated across all the vertices of the primitive.
[2 marks]

(iv) **By Copying:** The existing methods for setting positional coordinates, colors, normals, texture coordinates, and vertex attributes (such as setCoordinate, setColors, etc.) copy the data into this GeometryArray. This is appropriate for many applications and offers an application much flexibility in organizing its data. This is the default mode.
By Reference: A new set of methods in Java 3D version 1.2 allows data to be accessed by reference, directly from the user's arrays. To use this feature, set the BY_REFERENCE bit in the vertexFormat field of the constructor for this GeometryArray.
[2 marks]

(v) The ObjectFile class implements the Loader interface for the Wavefront .obj file format, a standard 3D object file format created for use with Wavefront's Advanced Visualizer.
RESIZE: Flag sent to constructor. The object's vertices will be changed so that the object is centered at (0,0,0) and the coordinate positions are all in the range of (-1,-1,-1) to (1,1,1).
REVERSE: Flag sent to constructor. Use if the vertices in your .obj file were specified with clockwise winding (Java 3D wants counter-clockwise) so you see the back of the polygons and not the front. Calls GeometryInfo.reverse().
[2 marks]

(vi) **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.
[2 marks]

(vii) Holes can be introduced in the generated mesh due to discontinuities between certain adjacent sets of surface patches. This can be corrected by removing complementary case and only using rotation to populate lookup table (requires 8 extra base configurations to be specifed)
[2 marks]

Q1(b)

Capability bits are used to specify what changes can be made to the scene graph after the scene has gone live. The following methods are provided by the **`SceneGraphObject`** class to deal with capability bits:

- **`void setCapability(int bit)`** Sets the capability indicated by the specified capability bit.
- **`boolean getCapability(int bit)`** Returns the status of the capability indicated by the specified capability bit.
- **`void clearCapability(int bit)`** Turns off the capability indicated by the specified capability bit.

[3 marks]

If the required capability bit has not been set, then any attempt to read or write the related property after the scene graph has gone live will result in a CapabilityNotSetException being thrown.
[1 mark]

The capability bits for a SceneGraphObject can only be changed when a scene graph is not live. Any attempt to change the capability bits after a scene graph has gone live will result in a RestrictedAccessException.
[1 mark]

Q1(c)

Scale => $\begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Translate => $\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix}$

Rotate => $\begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$

[2 marks each]

Q2(a)

The child index order array specifies which order the children of an **`OrderedGroup`** are rendered in. For example:

```
int[] childOrder = new int[5]
childOrder[0] = 4;
childOrder[1] = 3;
childOrder[2] = 2;
childOrder[3] = 1;
childOrder[4] = 0;
```

Would render the children of the OrderedGroup in the opposite order to the order in which they were added to the group.
[2marks for desc] - [2 marks for example]

Q2(b)

(i) TriangleArray – define 3 vertices for each triangle in the structure

```
float[] coordinates = {
                      -0.5f, 0.0f, 0.5f, //v1
                      0.5f, 0.0f, 0.5f, //v4
                      0.0f, 1.0f, 0.0f, //v0

                      0.5f, 0.0f, 0.5f, //v4
                      0.5f, 0.0f, -0.5f, //v3
                      0.0f, 1.0f, 0.0f, //v0

                      0.5f, 0.0f, -0.5f, //v3
                      0.0f, 1.0f, 0.0f, //v0
                      -0.5f, 0.0f, -0.5f, //v2

                      -0.5f, 0.0f, -0.5f, //v2
                      -0.5f, 0.0f, 0.5f, //v1
                      0.0f, 1.0f, 0.0f, //v0

                      -0.5f, 0.0f, 0.5f, //v1
                      0.5f, 0.0f, 0.5f, //v4
                      0.0f, -1.0f, 0.0f, //v5
```

```
                        0.5f, 0.0f, 0.5f, //v4
                        0.5f, 0.0f, -0.5f, //v3
                        0.0f, -1.0f, 0.0f, //v5

                        0.5f, 0.0f, -0.5f, //v3
                        0.0f, -1.0f, 0.0f, //v5
                        -0.5f, 0.0f, -0.5f, //v2

                        -0.5f, 0.0f, -0.5f, //v2
                        -0.5f, 0.0f, 0.5f, //v1
                        0.0f, -1.0f, 0.0f, //v5
            };
```

[3 marks]

(ii) IndexedTriangleArray – define the vertices needed, then specify indices into the vertex array to create the geometry:

```
float[] coordinates = {
                0.0f, 1.0f, 0.0f, //v0
                -0.5f, 0.0f, 0.5f, //v1
                -0.5f, 0.0f, -0.5f, //v2
                0.5f, 0.0f, -0.5f, //v3
                0.5f, 0.0f, 0.5f, //v4
                0.0f, -1.0f, 0.0f, //v5
  };

        int[] indices = {
                        1, 4, 0,
                        4, 3, 0,
                        3, 0, 2,
                        2, 1, 0,

                        1, 4, 5,
                        4, 3, 5,
                        3, 5, 2,
                        2, 1, 5,
            };
```

[4 marks]

(iii) IndexedTriangleFanArray – define the vertices needed, then specify the fan indices for the top strip and the bottom strips. Each fan is initialised using three vertices, then each additional triangle in the fan is specified using one additional vertex which connects the first vertex and the last vertex in the fan to form a new triangle. The strips counts are then specified, in this case there are two strips of six indices each.

```
float[] coordinates = {
                0.0f, 1.0f, 0.0f, //v0
                -0.5f, 0.0f, 0.5f, //v1
                -0.5f, 0.0f, -0.5f, //v2
                0.5f, 0.0f, -0.5f, //v3
                0.5f, 0.0f, 0.5f, //v4
                0.0f, -1.0f, 0.0f, //v5
      };

       int[] indices = {
                0, 1, 4,
```

```
                3, 2, 1,

                5, 4, 1,
                2, 3, 4
            };

    int[] stripIndexCounts = {6, 6};
```

[4 marks]

Q2(c)

- Ambient colour - the ambient RGB colour reflected off the surface of the material. The range of values is 0.0 to 1.0. The default ambient colour is (0.2, 0.2, 0.2).
- Diffuse colour - the RGB colour (i.e. the true colour) of the material when illuminated. The range of values is 0.0 to 1.0. The default diffuse colour is (1.0, 1.0, 1.0).
- Specular colour - the RGB specular colour of the material (highlights). The range of values is 0.0 to 1.0. The default specular colour is (1.0, 1.0, 1.0).
- Emissive colour - the RGB colour of the light the material emits, if any. The range of values is 0.0 to 1.0. The default emissive colour is (0.0, 0.0, 0.0).
- Shininess - the material's shininess, in the range [1.0, 128.0] with 1.0 being not shiny and 128.0 being very shiny. Values outside this range are clamped. The default value for the material's shininess is 64.
- Colour target - the material colour target for per-vertex colours, one of: AMBIENT, EMISSIVE, DIFFUSE, SPECULAR, or AMBIENT_AND_DIFFUSE. The default target is DIFFUSE.

[1 mark each]

Q2(d)

(i) Specify the texture coorindates:

```
float[] texCoords = {   0.0f, 0.0f,
                        1.0f, 0.0f,
                        1.0f,  1.0f,
                        0.0f, 1.0f};
```

[2 marks]

(ii) Add the necessary flag to the constructor:

```
GeometryArray geometryArray = new QuadArray(4,
GeometryArray.COORDINATES|GeometryArray.TEXTURE_COORDINATE_2);
```

[1 mark]

(iii) Associate the texture coordinates with the geometry:

```
geometryArray.setTextureCoordinates(0,0,texCoords);
```

[1 mark]

Q3(a)

(i) MouseZoom is a Java3D behavior object that lets users control the Z axis translation of an object via a mouse drag motion with the second mouse button. To use this utility, first create a transform group that this rotate behaviour will operate on. Then,

```
MouseZoom behavior = new MouseZoom();
behavior.setTransformGroup(objTrans);
objTrans.addChild(behavior);
behavior.setSchedulingBounds(bounds);
```

The above code will add the rotate behavior to the transform group. The user can rotate any object attached to the objTrans.

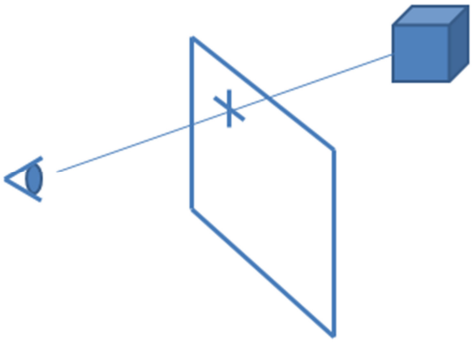
(ii) Position interpolator behavior. This class defines a behavior that modifies the translational component of its target TransformGroup by linearly interpolating between a pair of specified positions (using the value generated by the specified Alpha object). The interpolated position is used to generate a translation transform along the local X-axis of this interpolator.

(iii) The Billboard behavior node operates on the TransformGroup node to cause the local +z axis of the TransformGroup to point at the viewer's eye position. This is done regardless of the transforms above the specified TransformGroup node in the scene graph.

If the alignment mode is ROTATE_ABOUT_AXIS, the rotation will be around the specified axis. If the alignment mode is ROTATE_ABOUT_POINT, the rotation will be about the specified point, with an additional rotation to align the +y axis of the TransformGroup with the +y axis in the View.

[2 marks each]

Q3(b) Picking is essentially the opposite operation to viewing. It enables the selection of a specific shape by projecting 2D screen coordinates into the virtual world and identifying the shape associated with the coordinates. The PickCanvas class is used to turn the mouse coordinates into an area of space or a PickShape, that projects from the viewer through the mouse location into the virtual world. The PickCanvas class extends a more general PickTool class that defines basic picking operations. When a pick is requested, Java 3D figures out the pickable shapes that intersect with the PickShape. These shapes are stored in a list of PickResult objects. The picking process is illustrated below '+' represents the location of the mouse click and the ray projected from the viewer, through the mouse coordinates into the scene intersects with the cube i.e. the cube has been picked.

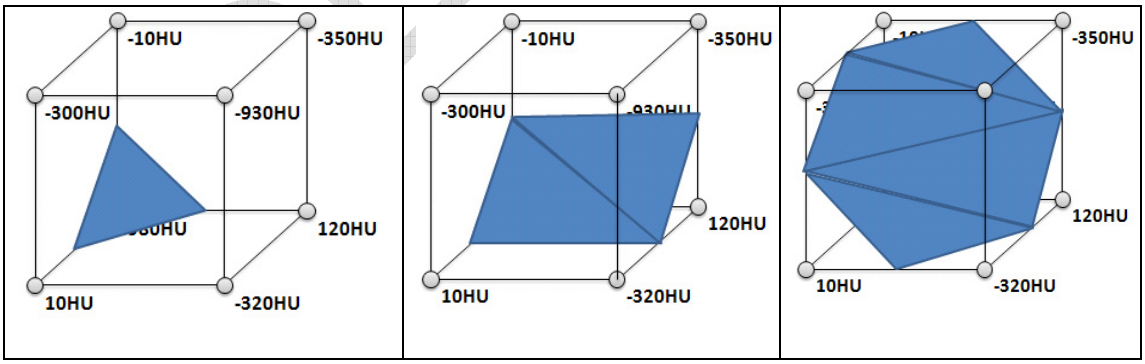


[4 marks]

Q3(c)

The marching cubes algorithm begins by thresholding the data set, assigning a 1 to voxels ≥ 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. These surface patches can be rendered using conventional computer graphics techniques.

There are 256 (2^8) possible configurations of eight binary voxels and although possible, the task of manually specifying the surface patches associated with each configuration is both tedious and prone to error. This task can be greatly simplified by considering all complementary cases (& rotations) for each configuration. Once specified the surface patches can be stored in a lookup table for easy access during the surface extraction process.
[3 marks]



[3 marks]

In order to fit the extracted surface more accurately to the actual isosurface identified by d_{iso}, each vertex must be interpolated between v_a & v_b based on the relationship between their densities d_a & d_b and the isosurface density d_{iso}. This is achieved by calculating the normalised distance (δ_{iso}) between the voxel that is closest to the origin (either v_a or v_b) and the isosurface (see Equation 3.1).

$$\delta_{iso} = \frac{d_{iso} - d_a}{d_b - d_a} \tag{3.1}$$

The value for δ_{iso} represents the intersection location relative to the reference voxel in terms of the intervoxel spacing (see Figure 3.5).

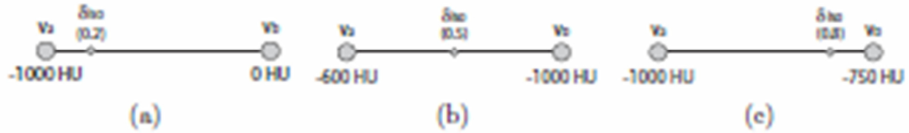


Figure 3.5: The calculation of δ_{iso} at three sample boundaries. In each case, the value of δ_{iso} is calculated using linear interpolation and the density at the point represented by d_{iso} is -800 HU, i.e. the isosurface density d_{iso}.

The closest voxel to the origin is used as the reference point v_a to ensure δ_{iso} has a positive value and to standardise the interpolation process throughout the surface extraction algorithm. The δ_{iso} value can then be used to calculate an interpolated vertex location that is more representative of the actual isosurface. Assuming that v_a is the closest voxel to the origin, the interpolated vertex location (x_i, y_i, z_i) located between v_a and v_b is calculated using:

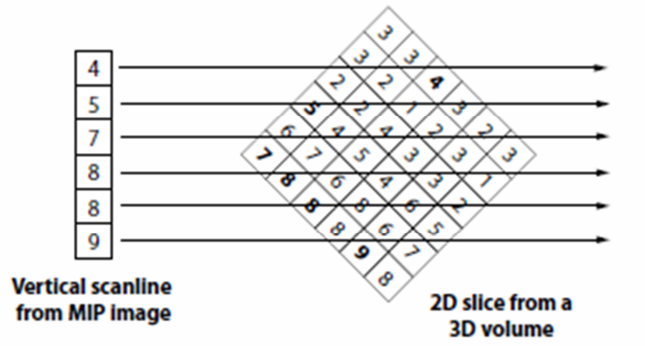
$$\begin{aligned} x_i &= x_a + \delta_{iso}(x_b - x_a) \\ y_i &= y_a + \delta_{iso}(y_b - y_a) \\ z_i &= z_a + \delta_{iso}(z_b - z_a) \end{aligned} \tag{3.2}$$

where (x_a, y_a, z_a) and (x_b, y_b, z_b) are the locations of v_a and v_b respectively.

[3 marks]

Q3(d)

A maximum intensity projection (MIP) is a simple technique for transforming 3D data sets to a 2D image. It involves identifying the maximum valued pixel along each ray projected from the pixels of the 2D view plane. The resulting maximum value is ultimately assigned to the associated pixel.



Maximum intensity projection is particularly useful in medical imaging applications where contrast enhanced material has been used to highlight a abnormal region of anatomy.

[4 marks]

One example of this is in a medical imaging technique known as magnetic resonance cholangiopancreatography. This involves the use of magnetic resonance imaging to visualise the biliary tree and pancreatic ducts in a non-invasive manner.

[2 marks]

Section B

4(a)

To create such a coordinate system, we consider a basis $\vec{v}_1, \vec{v}_2, ..., \vec{v}_n$ (a subset of vectors in vector space V), where a vector can be written as $\vec{v} = \alpha_1\vec{v}_1 + \alpha_2\vec{v}_2 + ... + \alpha_n\vec{v}_n$. The list of scalars $\alpha_1, \alpha_2, ..., \alpha_n$ is the representation of $\vec{v}$ with respect to this basis, and we can write it as $\begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 \end{bmatrix}^T$. A basis vector for a vector space is a set of linearly independent vectors where every object in the vector space can be described as a linear combination of the basis vectors.

In other words, a co-ordinate system is insufficient to represent points. If we work in an affine space, we can add an origin to our basis vectors to form a frame, as in figure 1.21(c). In the longitude and latitude example, this origin is the intersection of the prime meridian and the equator, thus providing us with a reference by which we can describe any point on the Earth. Mathematically, a frame is determined by $(P_0, \vec{v}_1, \vec{v}_2, \vec{v}_3)$, where within this frame every vector can be written as $\vec{v} = \alpha_1\vec{v}_1 + \alpha_2\vec{v}_2 + ... + \alpha_n\vec{v}_n$ and every point can be written as $P = P_0 + \beta_1\vec{v}_1 + \beta_2\vec{v}_2 + ... + \beta_n\vec{v}_n$. It is important not to confuse points and vectors; In this example we could write $\vec{v} = \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 \end{bmatrix}^T$ and $P = \begin{bmatrix} \beta_1 & \beta_2 & \beta_3 \end{bmatrix}^T$, but remember that a vector has no position, so the point is at a fixed location $(\beta_1, \beta_2, \beta_3)$, but the vector could be anywhere.

Homogeneous co-ordinates allow affine transformations to be easily represented by a matrix. Homogeneous co-ordinate systems are key to computer graphic systems as all standard transformations can be implemented with matrix multiplications using 4x4 matrices, taking advantage of an accelerated hardware pipeline.

We can write a vector $\vec{v}$ and point P as:

$$\vec{v} = \alpha_1\vec{v}_1 + \alpha_2\vec{v}_2 + \alpha_3\vec{v}_3 = \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 & 0 \end{bmatrix} \begin{bmatrix} v_1 & v_2 & v_3 & P_0 \end{bmatrix}^T \quad (1.28)$$

$$P = P_0 + \beta_1\vec{v}_1 + \beta_2\vec{v}_2 + \beta_3\vec{v}_3 = \begin{bmatrix} \beta_1 & \beta_2 & \beta_3 & 1 \end{bmatrix} \begin{bmatrix} v_1 & v_2 & v_3 & P_0 \end{bmatrix}^T \quad (1.29)$$

Therefore, a four-dimensional homogeneous representation is $\vec{v} = \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 & 0 \end{bmatrix}^T$ and $P = \begin{bmatrix} \beta_1 & \beta_2 & \beta_3 & 1 \end{bmatrix}^T$. The homogeneous co-ordinate form for a 3-D point $\begin{bmatrix} x & y & z \end{bmatrix}$ is given by $P = \begin{bmatrix} x' & y' & z' & w \end{bmatrix}^T = \begin{bmatrix} xw & yw & zw & w \end{bmatrix}^T$. If we have a point (i.e. $w \neq 0$, where if $w = 0$ the representation is that of a vector) we can return to a 3-D point by the transformations $x'/w \mapsto x$, $y'/w \mapsto y$ and $z'/w \mapsto z$. For $w = 1$, the representation of a point is $\begin{bmatrix} x & y & z & 1 \end{bmatrix}$. Homogeneous coordinates are frequently used in computer graphics because they solve the problem of representing a translation as a matrix operation.

If we consider two representations $a = \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 \end{bmatrix}$ and $b = \begin{bmatrix} \beta_1 & \beta_2 & \beta_3 \end{bmatrix}$ of the same vector $\vec{v}$ with respect to two different bases (as in figure 1.22), where:

$$\vec{v} = \alpha_1\vec{v}_1 + \alpha_2\vec{v}_2 + \alpha_3\vec{v}_3 = \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 \end{bmatrix} \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix}^T \quad (1.30)$$

$$\vec{v} = \beta_1\vec{u}_1 + \beta_2\vec{u}_2 + \beta_3\vec{u}_3 = \begin{bmatrix} \beta_1 & \beta_2 & \beta_3 \end{bmatrix} \begin{bmatrix} u_1 & u_2 & u_3 \end{bmatrix}^T \quad (1.31)$$

...

So, within two frames any vector or point has a representation $a = \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 & \alpha_4 \end{bmatrix}$ in the first frame and $b = \begin{bmatrix} \beta_1 & \beta_2 & \beta_3 & \beta_4 \end{bmatrix}$ in the second frame, where $\alpha_4 = \beta_4 = 0$ for vectors and $\alpha_4 = \beta_4 = 1$ for a point, then we can write $a = M^T b$, as before, where:

$$M = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} & 0 \\ \gamma_{21} & \gamma_{22} & \gamma_{23} & 0 \\ \gamma_{31} & \gamma_{32} & \gamma_{33} & 0 \\ \gamma_{41} & \gamma_{42} & \gamma_{43} & 1 \end{bmatrix}$$

[This is a screen grab of a description from the course notes. Each student must explain these concepts in their own words, but get across the core points – approx 2-3 marks per point]

[8 marks total]

4(b) The following design decisions are valid:

• Operator Overloading - For both ease of use and efficiency we have allowed the user to apply unary and binary operations on vectors and indeed vectors with scalars, eg. a ·b can be called using “a*b” and the cross product a×b can be called using aˆb (It is not possible to treat the character ‘x’ as an operator in C++). The use of the ˆ for the cross-product is not an ideal choice and sometimes draws criticism when used.

• The inline keyword - In C++ inline methods (identified by the inline keyword) are inserted by the compiler directly at the location where they are called. This provides significant performance improvement over standard methods, as there is no overhead in jumping the program counter in the compiled code. You should reserve this for short methods, otherwise your program size will be significant.

• Why the float type? - It depends on the precision of our application as to whether we should use float or double types9; floats will provide the level of accuracy that we will require in our systems. It is possible to use C++ templates to remove the type from the Vec3 class developed here, to allow us to substitute doubles if required.

• Using const methods - By placing the keyword const after a method name, we are undertaking that this method will not modify the object, allowing us to pass a constant vector to the method.

• Passing by const reference - Passing an object by value copies the data at that memory location to create a new copy of that value, allowing us to protect the original value from changing. However, this is inefficient, especially when we are passing an object, as the constructor must be called to create the object copy. Passing by constant reference passes

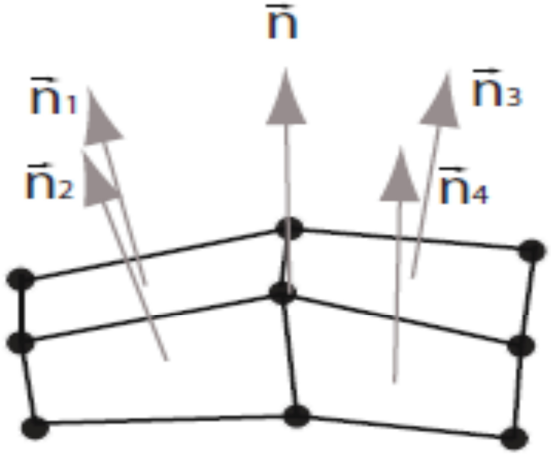
the address directly using the reference (like a pointer), but the const keyword prevents the value from being modified - providing all the benefits of pass-by-value, but with the efficiency of pass-by-reference.
• No virtual method? - Virtual methods are useful when our solution involves inheritance and dynamic binding; however, as a design decision the Vec3 class will not involve the creation of child classes. Virtual methods require that additional instructions are inserted and that checks are put in place at runtime, slowing performance, which would be problematic for this class. In particular, the use of the inline keyword would be affected by the compiler optimisation.
• No encapsulation? - Yes, if you completed an object-oriented module then you would know that it is generally poor practice to expose the states of the class (i.e. make them public). However, in this case the use of accessor and mutator methods would detrimentally affect performance. It is not clear that there would be any benefit either in checking the three values of the vector - as they are just floats of any value.

[9 marks total]

4(c)
With smooth surfaces the vector normal to the surface exists at every point and gives the local orientation of the surface. A surface normal, or just normal to a flat surface is a three-dimensional vector which is perpendicular to that surface (orthogonal). For a polygon, the surface normal can be calculated as the vector cross product of two non-parallel) vector edges of the polygon. For a plane given by the equation ax+by+cz+d = 0; this equation can be rewritten in terms of the normal to the plane as: ~n · (P − P0) = 0 where P is any point on the plane. If we have three non-collinear points P0, P1 and P2 (and therefore a plane) then the vectors P1 −P0 and P2 −P0 can be used to calculate the normal as: ~n = (P2 − P0) × (P1 − P0)

glNormal3f(Nx, Ny, Nz);
glNormal3fv(pointerToNormal);

Gouraud proposed the use of the normals around a mesh vertex. Gouraud shading finds the average normal at each vertex and applies the modified Phong model at each vertex. It then interpolates vertex shades across each polygon. Phong shading finds the vertex normals; interpolates the vertex normals across the edges; interpolates edge normals across the polygon and then applies the modified Phong model at each fragment. If the polygon mesh approximates a surface with a high curvature then Phong shading tends to look much smoother, while Gouraud shading tends to show up edges. Phong shading requires much more computational effort than Gouraud shading - It was not until recently that Phong shading was available in real-time, by using fragment shaders. Both of these shading schemes need data structures to represent the mesh, by which shading calculations can be performed.



Mathematically speaking, the calculation of a normal at a vertex should cause concern, as it is discontinuous at this point. However, Gouraud showed that by defining a normal at a vertex that smoother shading could be achieved over an object by using interpolation. In this figure

we can calculate the normal ~n at this vertex by using: ~n = (~n1 + ~n2 + ~n3 + ~n4)/(|~n1 + ~n2 + ~n3 + ~n4)

[8 marks total]

5(a)
Constructive Solid Geometry (CSG) is a technique whereby a complex model can be created by using Boolean operations to combine a set of primitive shapes (spheres, cubes, cylinders etc.). It is generally used for rigid CAD type models of machinery and equipment, where it is possible that these complex models will be manufactured from such primitives. The figure illustrates the main concepts behind CSG - In this example we have the equation (A − B) union (C intersection D), which is represented as a CSG tree in the same figure.

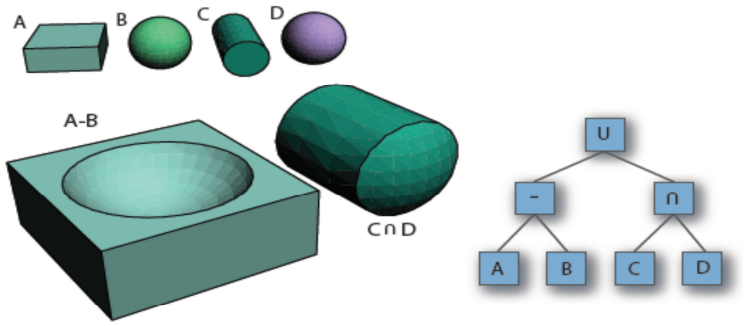


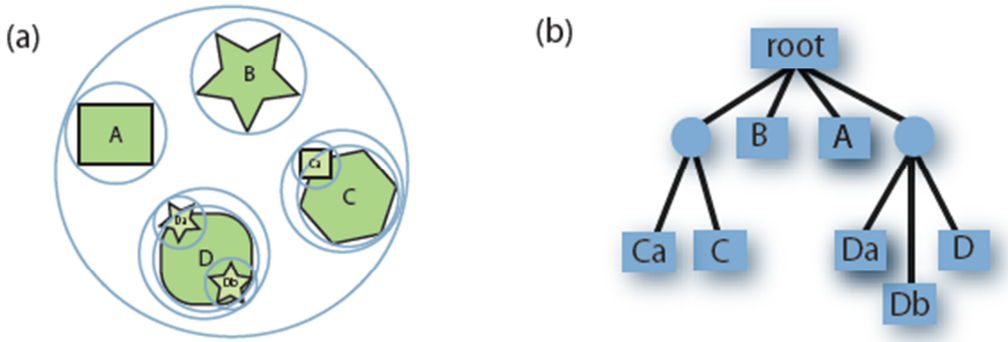
Figure 2.5: CSG Example illustrating the shapes A, B, C, D; the result of $A - B$ and of $C \cap D$; and the associated CSG Tree.

[4 marks]

5(b) 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.

[3 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 hierarchical 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.

[3 marks]
[6 marks total]

5(c)
The Level of Detail concept is that we can use simpler version of an object to represent that object when it is distant from the camera/viewer. Why would we do this? Well, one reason would be to preserve our frame rate... more triangles generally equates to a lower frame rate. For example, if we were representing a person in a scene who was made up of 50,000 triangles, it would be possible to represent that person with a lot less triangles if they were only occupying 20-30 pixels of screen space - perhaps we could represent the distant person by 1,000 triangles or less. A dodecahedron that we used before looks like a sphere from a large enough distance and so can be used to replace it so long as it is viewed from this or a greater distance. However, if it must ever be viewed more closely, it will look like a dodecahedron. If we decide to use the dodecahedron to stand in for a sphere then as the dodecahedron travels closer to the camera we must swap the dodecahedron with a higher detailed sphere. It is also possible to use other techniques such as fog to limit the depth of field and to allow us to ignore objects when we know that they will not be visible due to the foggy conditions. It is also more representative of the real-world, where a slight fog/haze limits our viewing depth. In general LOD algorithms consist of three separate parts, which are generation, selection and switching.
Generation: One mechanism to generate different LOD models of the same object would be to do it manually by hand. However, the quality of the different models could lead to noise when the different models are switched. Automated polygonal simplification tools can be used for levels of detail generation. They remove primitives from an original mesh in order to produce simpler models which retain the key visual characteristics of the original object. The result is typically a series of simplifications (as shown in 3.6), which can be used in various conditions, such as in order to maintain a constant frame rate. A good balance can be determined between the richness of the models and the time it takes to display them.
An alternative to LOD generation in certain cases is to use procedural modelling, which allows real-time detail to be added to a model as it is approached. One example of this is the use of fractals to model terrain, trees, plants within a scene. Such models are generated on-the-fly according to our procedural algorithm.
Selection is when we select which LOD is appropriate based on some criteria, such as distance from the object or the area that the object will appear on the screen. A metric, called the benefit function, is evaluated for the current viewpoint and the currently location of the object and the value of this function allows us to choose an appropriate LOD. There are different benefit functions, such as range-based that use a user-defined distance value to the object to decide if the object is visible and to what level of detail. An alternative approach is to use a project area-based LOD selection that often uses a bounding box to determine the screen space coverage.
Switching is when we change from one LOD to another and is prone to sudden changes (called popping). The most straightforward switching mechanism is Discrete Geometry LOD where the different representations of the same object contain different numbers of primitives.

The different LODs can be stored as indexed triangle strips and when required the higher/lower level LOD is simply swapped into the next rendered frame. An alternative approach is to use a Blend LOD which performs a blend operation over a number of frames to smooth the transition. It is however much more computationally expensive.
One LOD implementation could be as follows:

```
class Range
{
        float range;
        SceneObject_ child;
        public:
        Range(SceneObject_ s, float r) { child = s; range = r; }
        float getRange() { return range; }
        SceneObject_ getChild() { return child; }
};

class LOD : public SceneObject
{
        private:
        std :: vector<Range> _rangeList;
        SceneObject _selectedChild;

        public:
        LOD();
        virtual ~LOD();
        virtual RTTI OBJECT TYPE getType() const { return RTTI LOD; }
        void addChild(SceneObject_ child, float range);

        // Force every child to have a render and update methods
        virtual void render(float timeElapsed);
        virtual void postRender(float timeElapsed);
        virtual void update(float timeElapsed);
        virtual void switchChild(float distance);
};

void LOD::addChild(SceneObject_ child, float range)
{
        Range r(child, range);
        rangeList->push back(r);
        selectedChild = child;
}

void LOD::switchChild(float distance)
{
        float curDist = 999999.0f;
        for(int i=0; i<rangeList->size(); i++)
        {
        // find object with smallest r that is greater than distance
                Range range = rangeList->at(i);
                float r = range.getRange();
                SceneObject _o = range.getChild();

                if ((r <= curDist) && ( r >= distance))
                {
                        curDist = r;
                        selectedChild = o;
                }
        }
}
```

[10 marks]

5(d) Clipping against the view volume removes objects which are not within the view volume, where typically polygons outside of the view volume are discarded and polygons that intersect the view volume boundary are 'clipped'. Culling is a similar operation that takes account of the viewer's position to determine if part of an object cannot be seen; for example, in 3-D (the real world!) most objects are only 50% visible from any one point, as the back side of the object is occluded by the front side. A very simple test to see if we are looking at the front-side or back-side of a polygon is to use the view vector ~n and the polygon's normal vector ~np, where a particular polygon is visible if: $\sim np \cdot \sim n > 0$. Similarly if the polygon is outside the bounding box of the view volume then it can be clipped.

[5 marks]

6(a)

• 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.

[3 marks]

6(b)

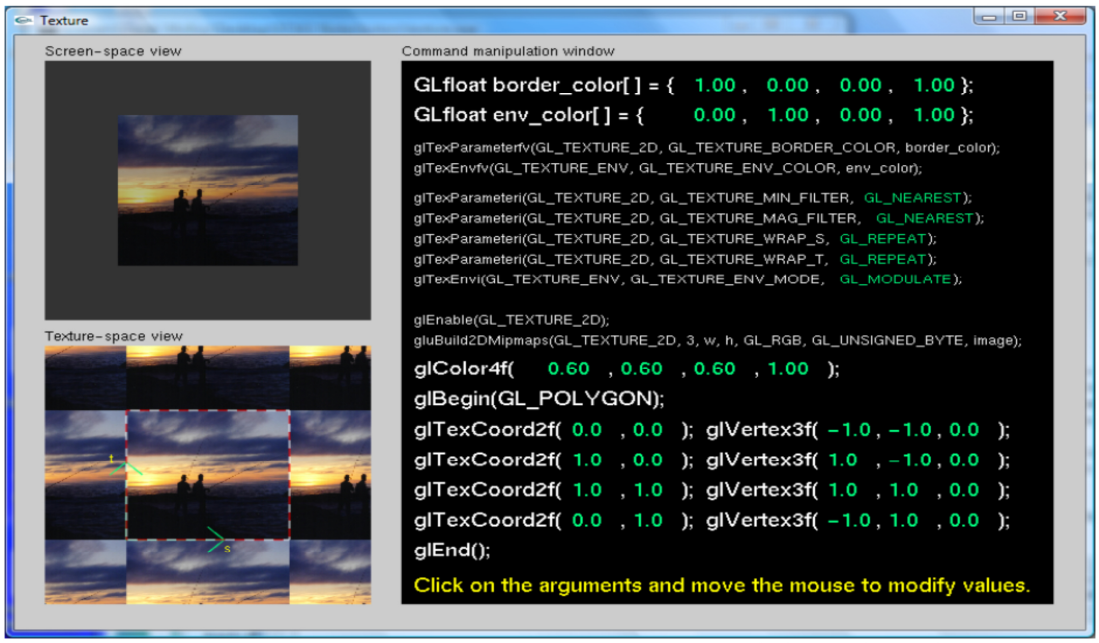
The Phong Reflection model provides a good approximation to physical reality, producing good renderings under varying lighting conditions and materials. The Phong model uses four vectors to calculate the colour at a particular point P on a surface; these are n, the normal vector at that point on the 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 reflections 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: , 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 kd (where $0 < kd < 1$) we can write: $Rd = kdLd \cos theta$.

[7 marks]

6(c)

(i) The output should look like the image below. In particular there should be a full description of the image on the left-hand side.

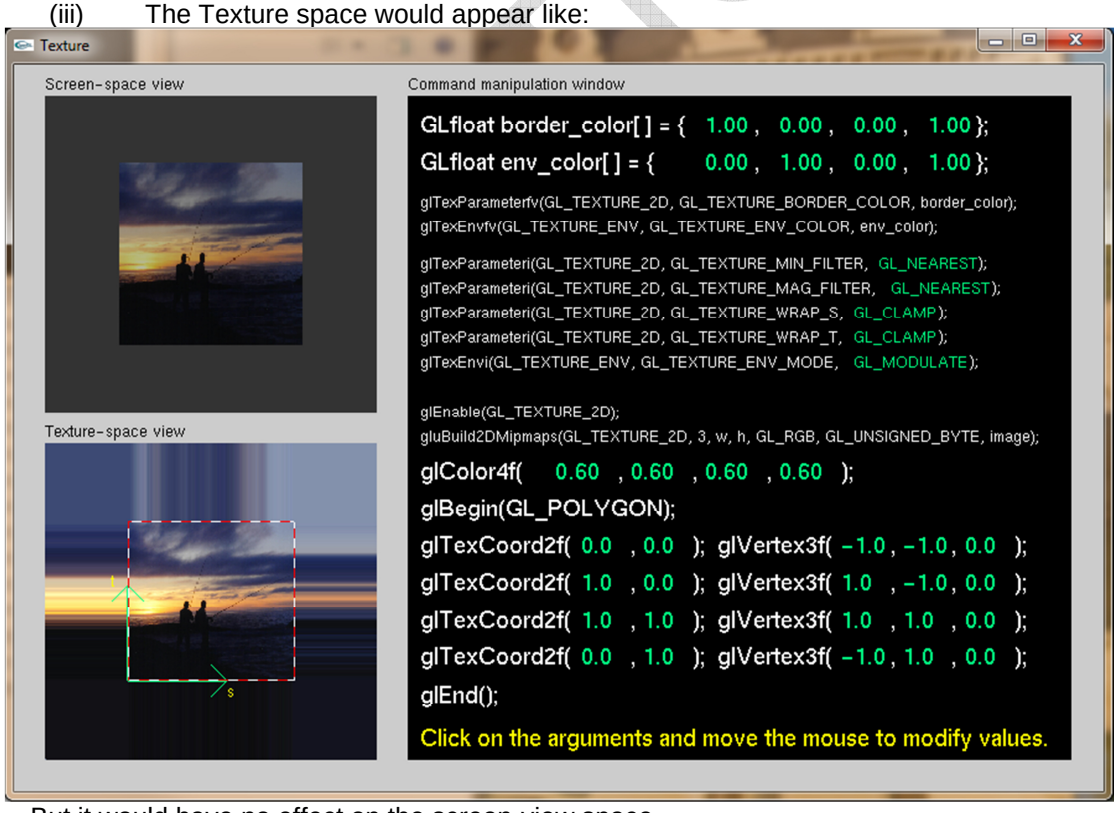


[6 marks]

(ii) It would give the image a red hue in the screen-space view. However, when the texture environment value is changed to GL_REPLACE then the effects of the red hue will disappear as the texture is completely replacing the underlying material properties.

[3 marks]

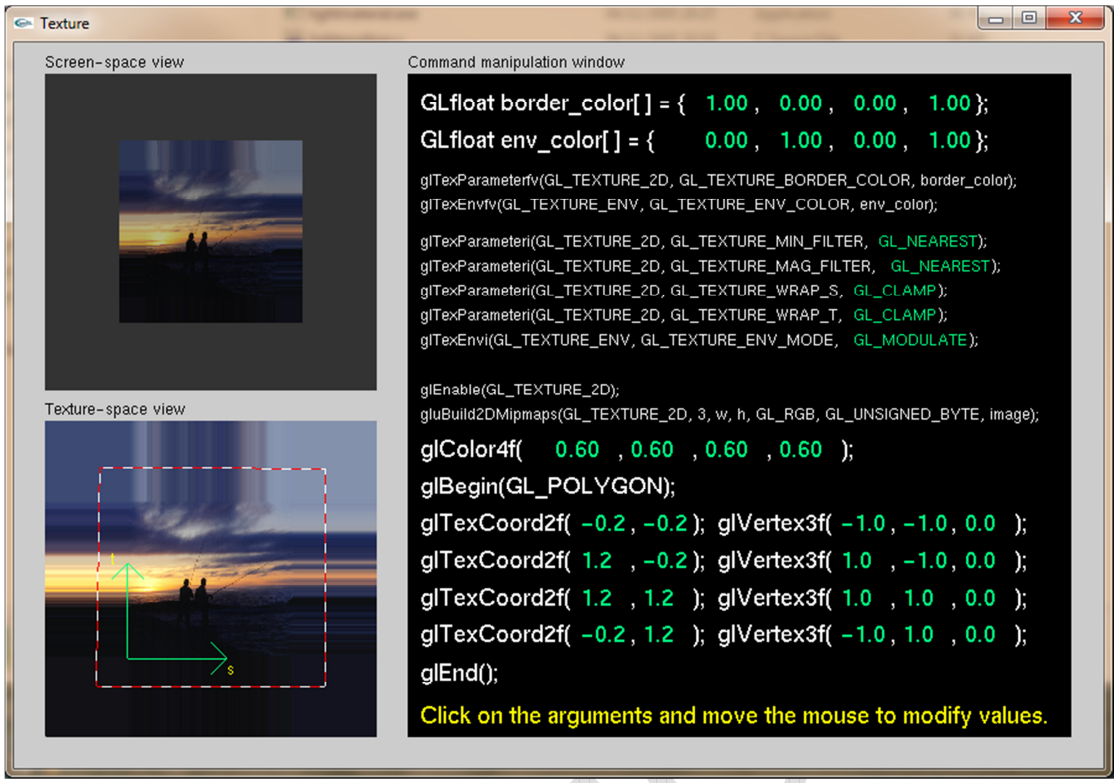
(iii) The Texture space would appear like:



But it would have no effect on the screen-view space.

[3 marks]

(iv)



It will expand the texture mapping in the texture space and will mean that screen-space view will see clamped values at the edges of the polygon.

[3 marks]
[15 marks total]