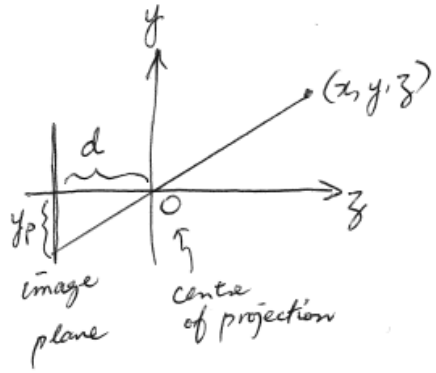
Blue colour text – questions
Black colour text – sample answers
Red colour text – further explanation or references for the sample answers

Question 1. (12 marks)

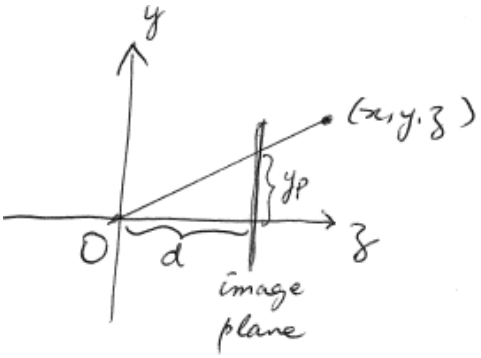
a) (5 marks) Explain the OpenGL *synthetic camera* model, including how it relates to a *pinhole camera* and how it relates 3D space to 2D images.

The synthetic camera used in OpenGL is based on the pinhole camera model in which the rays of all the scene points pass through the centre of projection before projecting onto the image plane.



Projection of 3D space to 2D image is governed by the equations:

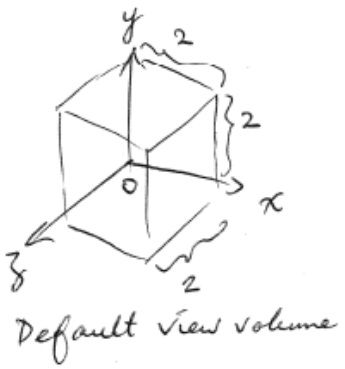
$$\frac{x_p}{z_p} = \frac{x}{z}$$
$$\frac{y_p}{z_p} = \frac{y}{z}$$
$$z_p = -d$$



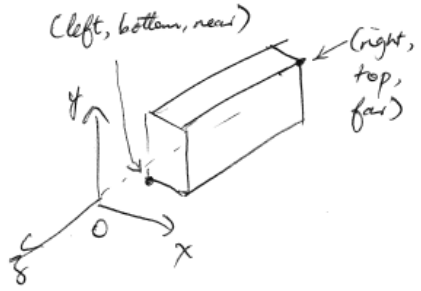
Because the pinhole camera model produces upside down and left-right reversed images, the synthetic camera model of OpenGL modifies this by moving the image plane to the same side as the scene, giving the diagram shown on the right.
The projected image is now the right way up and no longer left-right reversed.

b) (3 marks) Describe the shape of the OpenGL default view volume. Describe what you would do if you want to use a view volume other than the default one (**Hint:** think about other shapes of the view volume).

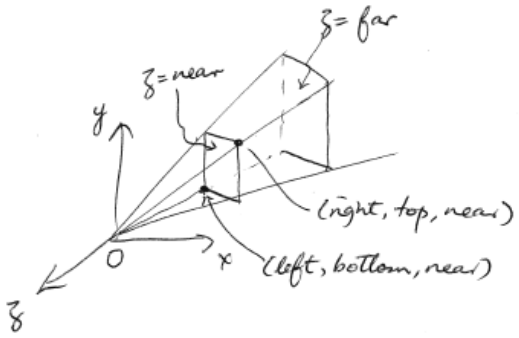
The default view volume of OpenGL is a cube centred at the origin, having sides of length 2 units.



Other view volumes can be defined by specifying the bottom, top, left, right, near, and far clipping planes. For instance, we can use the *Ortho* function to define a rectangular view volume for orthographic projection.



We can have a view volume taking the shape of a truncated pyramid for perspective projection, e.g., by calling the *Perspective* or the *Frustum* function.



For clarity, I have put some words in italics in the sample answer above. In the exam, you don't need to worry about doing the same. You can underline words if they are special terms (or names) that you want to distinguish from the rest of the sentence.

c) (2 marks) What is clipping in the context of OpenGL, and where does it fit in the steps of the OpenGL pipeline architecture?

OpenGL has a default view volume defined. Only objects falling within the volume would be processed further. This is similar to cameras having a finite field of view. In the OpenGL pipeline architecture, the clipping step takes place after the vertex processing step. After the vertices of the object have been transformed by the vertex processor, vertices are grouped together to form primitives. Primitives falling outside the view volume are clipped away. Clipping takes place near the beginning of the pipeline for an obvious reason — only primitives survive the clipping need to be processed further.

d) (2 marks) What is rasterization in the context of the OpenGL pipeline architecture, and what are the inputs and outputs to the rasterization step?

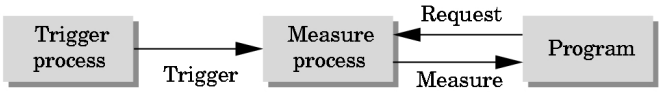
The rasterizer's job is to produce a set of fragments from each primitive that is passed to it. In the OpenGL pipeline architecture, the rasterizer step is located after the "primitive assembler and clipping" step but before the "fragment processor" step. The inputs to the rasterizer are therefore those primitives that survive the clipping. The output produced by the rasterizer

are fragments. Each fragment is a potential pixel carrying the pixel location, but it usually contains additional information such as depth, colour, alpha (opacity), etc. If normals, view vectors, etc., are passed down to the rasterizer, then they would be interpolated and output to the fragment processor also.

Question 2. **(12 marks)**

a) (6 marks) Describe *request mode* and *event mode*. What is the difference between them? Use appropriate diagrams in your explanation.

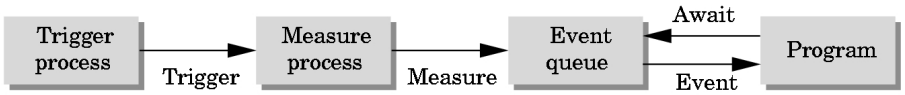
Request mode:



The program issues an input request from an input device (e.g., mouse click) at a particular point of the program. The program just hangs there until the Trigger Process detects the input (the user clicks the mouse). A trigger is sent to the Measure Process which returns a measure (in the mouse click example, the measure would include the mouse click position, which mouse button is clicked, the status of the mouse button (up or down)) to the program.

In request mode, the program is hard coded to process events in a pre-defined order.

Event mode:



In event mode, the program specifies a number of input events that are of interest. The program gets into an event handling loop. Each trigger of an event generates a measure for that event. The measures of all the events are put into an event queue by the windows manager. Each time in the event handling loop, the windows manager goes through each event and passes its measure to the appropriate callback function.

In event mode, the program does not need to know what events would be invoked by the user at what time during execution.

b) (2 marks) What is the purpose of the *idle callback* in OpenGL? Include an example of what it can be used for.

The purpose of an idle callback function is to specify what the program should do when no windows events are received. At each pass of the event handling loop, if no events are received, the windows manager would automatically call the idle callback. Because of that, idle callback functions are suitable when we want to have continuous animation in the program, e.g., a cube continuously rotating.

c) (2 marks) What is *double buffering*? When would it be suitable to use double buffering?

Double buffering refers to using two buffers for rendering and displaying. While the front buffer is being displayed, the scene is rendered and stored in the back buffer for the next frame. Double buffering would be suitable when we want to have smooth animation (i.e., avoid flickering).

d) (2 marks) Give three common applications of vertex shaders in OpenGL.

Three common applications (See topic 5, slides 18-19):

(a) Geometric transformation – change relative location, rotation, and scale of objects; apply 3D perspective transformation (make far objects appear smaller).

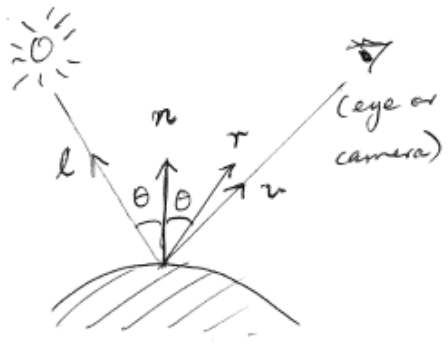
(b) Morphing vertices, e.g., compute wave motion and deformation, simulate particle effects such as fire, smoke, rain.

(c) Lighting/shading – calculate the shading colour using light and surface properties (per-vertex shading).

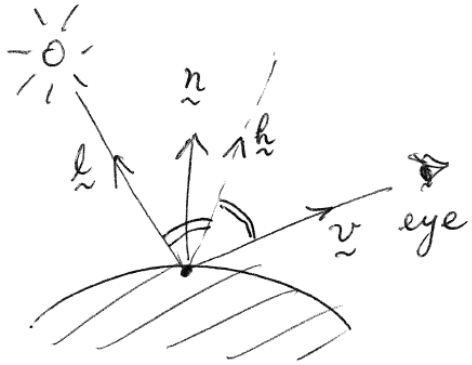
Question 3. **(12 marks)**

a) (6 marks) What is the difference between the Phong model of shading and the modified Phong/Blinn-Phong model, and what are their relative advantages?

Both models are shading formulae involving the diffuse, specular, and ambient terms for each colour (R, G, and B) component.



(Phong model)



(Blinn-Phong model)

Phong model:

$$I = k_d I_d(\mathbf{l} \cdot \mathbf{n}) + k_s I_s(\mathbf{v} \cdot \mathbf{r})^\alpha + k_a I_a$$

Blinn-Phong model:

$$I = k_d I_d(\mathbf{l} \cdot \mathbf{n}) + k_s I_s(\mathbf{n} \cdot \mathbf{h})^\beta + k_a I_a$$

where
k_d, k_s, k_a: diffuse, specular, and ambient coefficients for the material

I_d, I_s, I_a: similar coefficients but for the light source
$\mathbf{l}$: light source vector
$\mathbf{n}$: normal vector
$\mathbf{v}$: view (or eye) vector
$\mathbf{r}$: perfect mirror reflection vector of $\mathbf{l}$ w.r.t. $\mathbf{n}$
$\mathbf{h}$: half-way vector between $\mathbf{l}$ and $\mathbf{v}$
α: shininess coefficient
β: shininess coefficient modified from α

Phong model:

Advantage: gives better rendering result, but requires the reflected vector $\mathbf{r}$ to be computed.

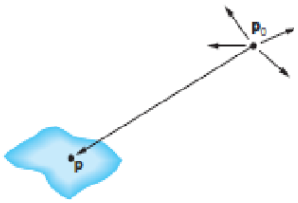
Blinn-Phong:

Advantage: the half-way vector $\mathbf{h}$ is cheaper to compute; rendering result is only slightly worse than the Phong model; is the default shading model used in OpenGL.

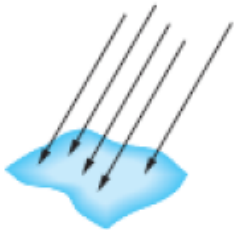
In the exam, you don't need to darken a symbol to boldface in order to denote that it is a vector. You can just write it the same as scalar symbols. Alternatively, you can use one of the following common notations for denoting vectors in hand-writing: (1) put an arrow on top of the symbol (such as $\vec{n}$, $\vec{l}$); or (2) put a tilde below the symbol (such as $\underset{\sim}{n}$, $\underset{\sim}{l}$).

b) (2 marks) Explain the difference between a directional light and a point source light in the context of OpenGL.

A point source is modelled with a position $\mathbf{p}_0 = (x, y, z)$ and an intensity $\mathbf{I} = (r, g, b)$. It illuminates equally in all directions. A distance term can be incorporated to attenuate $\mathbf{I}$.



A directional light illuminates objects with parallel rays of light. It is modelled with a direction and an intensity $\mathbf{I} = (r, g, b)$.



c) (4 marks) Why is a fourth component w added to the usual three dimensional coordinates in computer graphics systems like OpenGL? Give two common uses of the extra entries in the corresponding 4×4 matrices (those in the last row and column).

If 3×3 matrices are used to transform 3D points and vectors, we have to treat translation differently.

For 3D rotation and scaling, we can define a 3×3 transformation matrix M, and transform a 3D point $\mathbf{p}$ to $\mathbf{p}'$ by matrix-vector multiplication: $\mathbf{p}' = M\mathbf{p}$.

However, for translation, we would have to do vector addition: $\mathbf{p}' = \mathbf{p} + \mathbf{d}$, where $\mathbf{d} = (d_x, d_y, d_z)$ is a 3D vector.

By adding a fourth component w so that 3D points (and 3D vectors also) are represented as homogeneous coordinates, all the transformation can be done via matrix-vector multiplication.

The 3×3 matrix M earlier becomes a 4×4 matrix:

$$\begin{bmatrix} & & & 0 \\ & M & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

For translation, the vector $\mathbf{d}$ can now appear in the last column of the matrix:

$$\begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Not only so, the last (4th) row of the 4×4 matrix can be used for perspective projection. For affine transformation, the last row of the 4×4 matrix is $(0, 0, 0, 1)$. For perspective transformation, the last row is $(a, b, c, 1)$ where a, b, and c are some non-zero numbers.

Question 4. **(12 marks)**

a) (5 marks) Explain clearly what each function call does in the code below:

```
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
glutInitContextVersion( 3, 2 );
glutInitContextProfile( GLUT_CORE_PROFILE );
glutCreateWindow("Example");
glutMouseFunc(mouse);
glutReshapeFunc(reshape);
glutDisplayFunc(display);
init();
glutMainLoop();
```

Line glutInit:
initializes GLUT using the supplied command line arguments.

Line glutInitDisplayMode:
specifies that the program requires double and colour (including alpha) buffers.

Line glutInitContextVersion:
specifies the GLUT context version number needed by the program.

Line glutInitContextProfile:
specifies that GLUT Core profile is needed.

Line glutCreateWindow:
asks to create a window with heading = "Example".

Line glutMouseFunc:
registers function *mouse* as the mouse callback function.

Line glutReshapeFunc:
registers function *reshape* as the reshape callback function.

Line glutDisplayFunc:
registers function *display* as the display callback function.

Line init:
calls the user-defined *init* function.

Line glutMainLoop:
program enters the event handling loop (never returns).

b) (3 marks) Suppose you want to build a program using OpenGL that draws two kinds of objects - shiny billiard balls, and dull wooden balls. How would you implement these in a way that realistically represents the difference in shininess between the two kinds of objects?

For the shiny billiard balls, I would look at the colour of the shiny billiard balls and set their diffuse coefficients $(k_{d,r}, k_{d,g}, k_{d,b})$ appropriately. For the specular coefficients $(k_{s,r}, k_{s,g}, k_{s,b})$ I would adjust them to larger numbers than the diffuse coefficients. So, when the Phong (or modified Phong) shading formula is applied, the specular term dominates. I would also set the shininess coefficient α to be large (between 100 and 200).

For the dull wooden balls, I would set the diffuse coefficients appropriately for the colour of wood. For the specular coefficients, I would set them smaller in proportion, so the diffuse term dominates in the shading computation. I would also set α to a small number (e.g., 5 to 10).

c) (2 marks) Suppose you have a situation where both the viewer and the only light source are always very distant from the objects in a scene. How could you take advantage of this to perform the lighting calculations more efficiently?

Since the objects would be quite small, I would avoid doing per-fragment shading completely. I would program the vertex shader or the application program to compute the colour and let the colour pass through the pipeline to the fragment shader. If the rendering result looks acceptable, I would use flat shading (one shading computation per polygon) to further reduce computation.

d) (2 marks) What are Euler angles? Briefly explain how you would represent a rotation using these angles.

Euler angles are three angles for describing the orientation of an object with respect to a fixed coordinate system. The Euler angles are the three rotation angles α, β, and γ, about the x, y, and z axes respectively. Any orientation can be described by a rotation matrix R computed as the product $R_x(\alpha)R_y(\beta)R_z(\gamma)$, where $R_x(\alpha)$ denotes rotating an angle α about the x-axis and so on.

Question 5. **(12 marks)**

a) (3 marks) Consider a horizontal row of fragments that form part of a triangle during rasterization. Suppose the coordinates, normals and texture coordinates of the first and last fragment in the row have already been calculated. How can we quickly calculate the coordinates of the other fragments as we scan along the row from the first fragment to the last fragment?

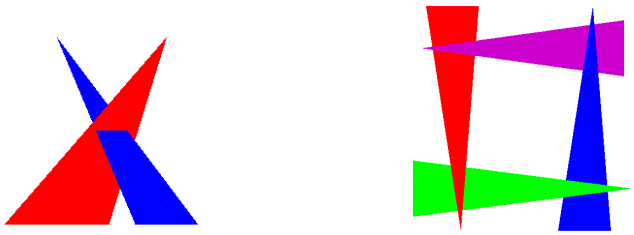
We can calculate the coordinates of other fragments by interpolating the coordinates of the first and last fragments. We firstly compute the distance D (in pixel units) of the two end fragments. Each time we move along the row one pixel to the right, the distance of the current pixel to the first pixel is increased by 1. Let d be this distance value. Then the coordinates of each pixel along the row can be computed as:

(pseudo code)
for $d = 1$ to D {
$\quad \mathbf{x} = \frac{D-d}{D}\mathbf{x}_1 + \frac{d}{D}\mathbf{x}_2$
}
where $\mathbf{x}_1$ and $\mathbf{x}_2$ denote the coordinates of the two end fragments.

The normals and texture coordinates can be interpolated similarly.

b) (2 marks) The *painter's algorithm* for hidden surface removal is an alternative to the *z-buffer* algorithm that instead sorts scene objects by distance and then draws them starting from the furthest away, with each overwriting (or "overpainting") what has previously been drawn. Describe two situations where this algorithm will not work well.

The painter's algorithm fails under two situations:



1. when the triangles intersect.
2. when the triangles form a cycle of overlapping depth.

In the exam, you can draw simpler diagrams and no need to put in colours.

c) (2 marks) What is *mipmapping*? What is the advantage of using mipmapping?

Mipmapping is a technique where an original high-resolution texture map is scaled and filtered into multiple lower resolution ones. Mipmapping lessens interpolation error for smaller textured objects.

d) (2 marks) Describe the two options for texture sampling in OpenGL. Contrast these options in terms of their rendering outputs and computation times.

OpenGL supports two options for sampling textures:

1. Point sampling. This sampling option uses the value of the texture element that is closest to the texture coordinates output by the rasterizer.
2. Linear filtering. This sampling option computes the weighted average of texture elements in the neighbourhood of the texture coordinates output by the rasterizer.

Option 1 is faster but produces poorer rendering results. Option 2 is slower but produces better results.

(Topic 16 "OpenGL Texture Mapping", slide 13)

e) (3 marks) What is aliasing? Describe a way to ease the aliasing problem in texture mapping.

Aliasing refers to the process by which smooth curves or lines become jagged due to the resolution of the graphics device or insufficient data to represent the curves.

In texture mapping, we can ease the aliasing problem by considering that each pixel is not just a point but but it occupies a finite area (region) on the computer screen. With such consideration, we should extract the texture area corresponding to the four corners of each pixel and assign the average value of the texture region to the pixel. Mapping area to area like so helps to ease the aliasing problem.

(Topic 15 "Texture Mapping", slides 17-18.)

Question 6. **(12 marks)**

a) (3 marks) What kinds of objects are generally suitable for modelling using a subdivision surface technique? Include an example of where using a subdivision surface technique would be appropriate, and another of where it wouldn't.

Objects that

- are locally smooth – having continuous 1st derivatives everywhere and continuous 2nd derivatives almost everywhere; and
- can be recursively generated from a simple polygonal mesh acting as the control cage

are suitable for modelling using subdivision surface techniques.

Examples:

- objects suitable for modelling using subdivision surface techniques include: articulated figures, cartoon characters.
- objects not suitable: industrial engineering parts. For such objects, precision is important. As subdivision surface techniques will smooth the object in the subdivision process, they are not suitable for modelling this type of objects.

In *subdivision surfaces* terminology, A *control cage* is a coarse polygonal mesh where the vertices of the mesh are used as control points for subdivision.

Some references refer to *subdivision surface techniques* loosely as *subdivision surfaces*. Sometimes the term *subdivision surfaces* is used to referred to the mesh being subdivided.

b) (3 marks) Suppose we apply the Catmull-Clark technique to subdivide surfaces starting with a box shape. What happens to the resulting surface if we add additional vertices near one corner without altering the shape of the box?

The Catmull-Clark technique uses an *approximation scheme* to subdivide surfaces. Under this scheme, the subdivided mesh does not necessarily pass through the vertices on the control cage. However, since all control points have the same weight on the subdivision surfaces, if we add additional vertices (control points) near one corner of the box shape (the control cage), then with the extra weights from the added control points, the subdivision surfaces will be pulled toward these control points. We effectively create a sharp corner at that region.

In general, if we want a sharp corner, we put several control points close together; if we want a smooth surface, we put the control points further apart.

Some other subdivision surface techniques use the *interpolation scheme*.

c) (3 marks) What are bone weights in the context of animation?

In animation of articulated figures, vertices on the mesh should follow the movement of the underlying bones smoothly. In particular, for vertices that are near the joint of two or more bones, their movements should be affected by all of these bones. In animation, we therefore need a small number (4 is often sufficient) of *bone weights* and *bone IDs* attached to each vertex. These bone weights should be non-negative and should sum to 1. Bones that are close to a vertex should have more influence and their weights should be higher (close to 1). When animating the articulated figure, each vertex is then transformed by the weighted sum of the 4 bone transformations.

d) (3 marks) In a typical implementation of skinning in OpenGL that takes advantage of shader programs, like that in lectures, say what the main static data is that's passed to the shaders (i.e., not changing between frames) and what the main dynamic data is that's passed to the shaders (i.e., changing each frame).

Please inspect the vertex shader in part 2 of your project. What static data are passed from the application program to the vertex shader? vertices of the mesh? vertex normals? bone ids? bone weights? anything else?
What main dynamic data is passed to the shader? bone transforms? anything else?

END OF PAPER