Cs 602 Imp

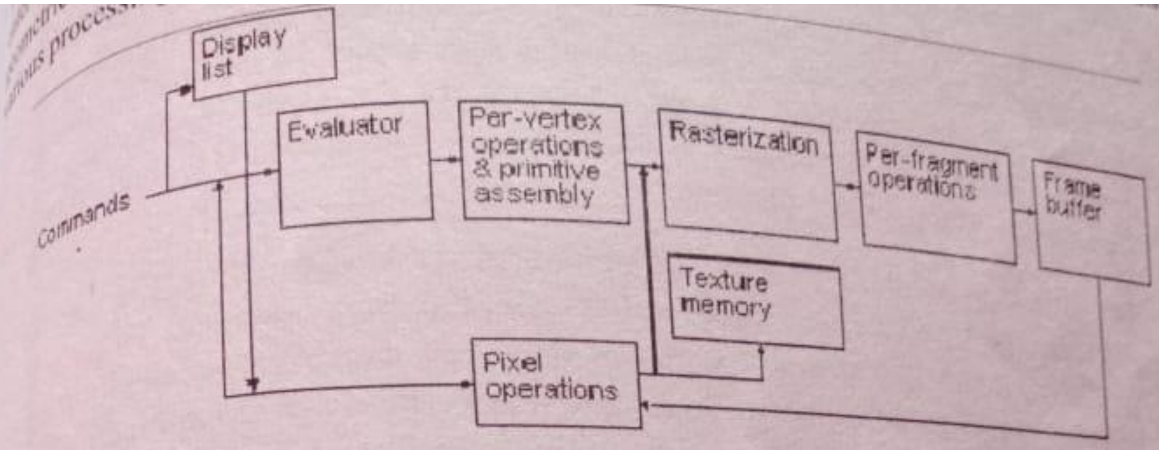
Topic for

Final Term

By

senior

student



The processing stages in basic OpenGL operation are as follows:

✓ **Display list** Rather than having all commands proceed immediately through the pipeline, you can choose to accumulate some of them in a display list for processing later.

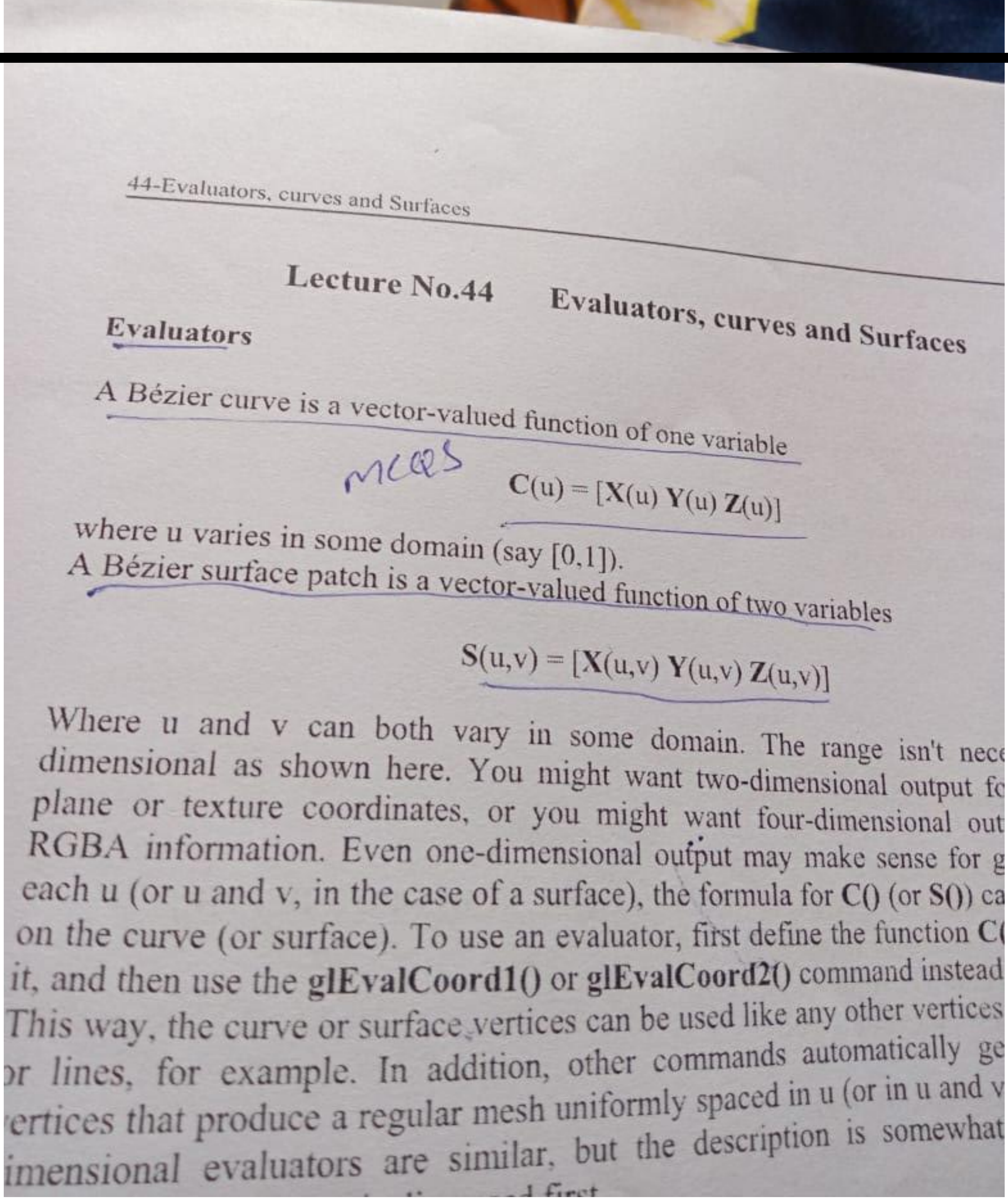
✓ **Evaluator** The evaluator stage of processing provides an efficient way to approximate curve and surface geometry by evaluating polynomial commands of input values.

✓ **Per-vertex operations and primitive assembly** OpenGL processes geometric primitives—points, line segments, and polygons—all of which are described by vertices. Vertices are transformed and lit, and primitives are clipped to the view port in preparation for rasterization.

✓ **Rasterization** The rasterization stage produces a series of frame-buffer addresses and associated values using a two-dimensional description of a point, line segment, or polygon. Each fragment so produced is fed into the last stage, per-fragment operations.

✓ **Per-fragment operations** these are the final operations performed on the data before it's stored as pixels in the frame buffer.

Per-fragment operations include conditional updates to the frame buffer based on incoming and previously stored z values (for z buffering) and blending of



Lecture No.44 Evaluators, curves and Surfaces

Evaluators

A Bézier curve is a vector-valued function of one variable

MCQS

$$\mathbf{C}(u) = [\mathbf{X}(u)\ \mathbf{Y}(u)\ \mathbf{Z}(u)]$$

where u varies in some domain (say [0,1]).
A Bézier surface patch is a vector-valued function of two variables

$$S(u,v) = [\mathbf{X}(u,v)\ \mathbf{Y}(u,v)\ \mathbf{Z}(u,v)]$$

Where u and v can both vary in some domain. The range isn't nece
dimensional as shown here. You might want two-dimensional output fc
plane or texture coordinates, or you might want four-dimensional out
RGBA information. Even one-dimensional output may make sense for g
each u (or u and v, in the case of a surface), the formula for **C**() (or **S**()) ca
on the curve (or surface). To use an evaluator, first define the function **C**(
it, and then use the **glEvalCoord1()** or **glEvalCoord2()** command instead
This way, the curve or surface vertices can be used like any other vertices
or lines, for example. In addition, other commands automatically ge
ertices that produce a regular mesh uniformly spaced in u (or in u and v
imensional evaluators are similar, but the description is somewhat

Figure 7: A soybean field showing differing reflection properties.

how the backscattering image shows a near uniform diffuse illumination, whereas ard scattering image shows a uniform dull diffuse illumination. Also note that we specular highlights and more color variation because of the shadows due to the rface whereas the backscattered image washes out the detail. In an effort to better ough surfaces, Oren and Nayar [OREN 1992] came up with a generalized version nbertian diffuse shading model that tries to account for the roughness of the They applied the Torrance—Sparrow model for rough surfaces with isotropic s and provided parameters to account for the various surface structures found in ance—Sparrow model. By comparing their model with actual data, they d their model to the terms that had the most significant impact. The Oren– ffuse shading model looks like this.

$$i_d = \frac{\rho}{\pi} E_0 \cos(\theta_i)(A + B\max[0, \cos(\phi_r - \phi_i)]\sin(\alpha)\tan(\beta))$$

$$A = 1 - 0.5\frac{\sigma^2}{\sigma^2 + 0.33}$$

$$B = 0.45\frac{\sigma^2}{\sigma^2 + 0.09}$$

may look daunting, but it can be simplified to something we can appreciate if e the original notation with the notation we've already been using. ρ/π is a flectivity property, which we can replace with our surface diffuse color. E0 is a t energy term, which we can replace with our light diffuse color. And the θi st our familiar angle between the vertex normal and the light direction. Making anges gives us

$$i_d = (m_d \otimes s_d)(\hat{n} \bullet \hat{l})(A + B\max[0, \cos(\phi_r - \phi_i)]\sin(\alpha)\tan(\beta))$$

(Oren-Nayer)

ks a lot more like the equations we've used, there are still some parameters to

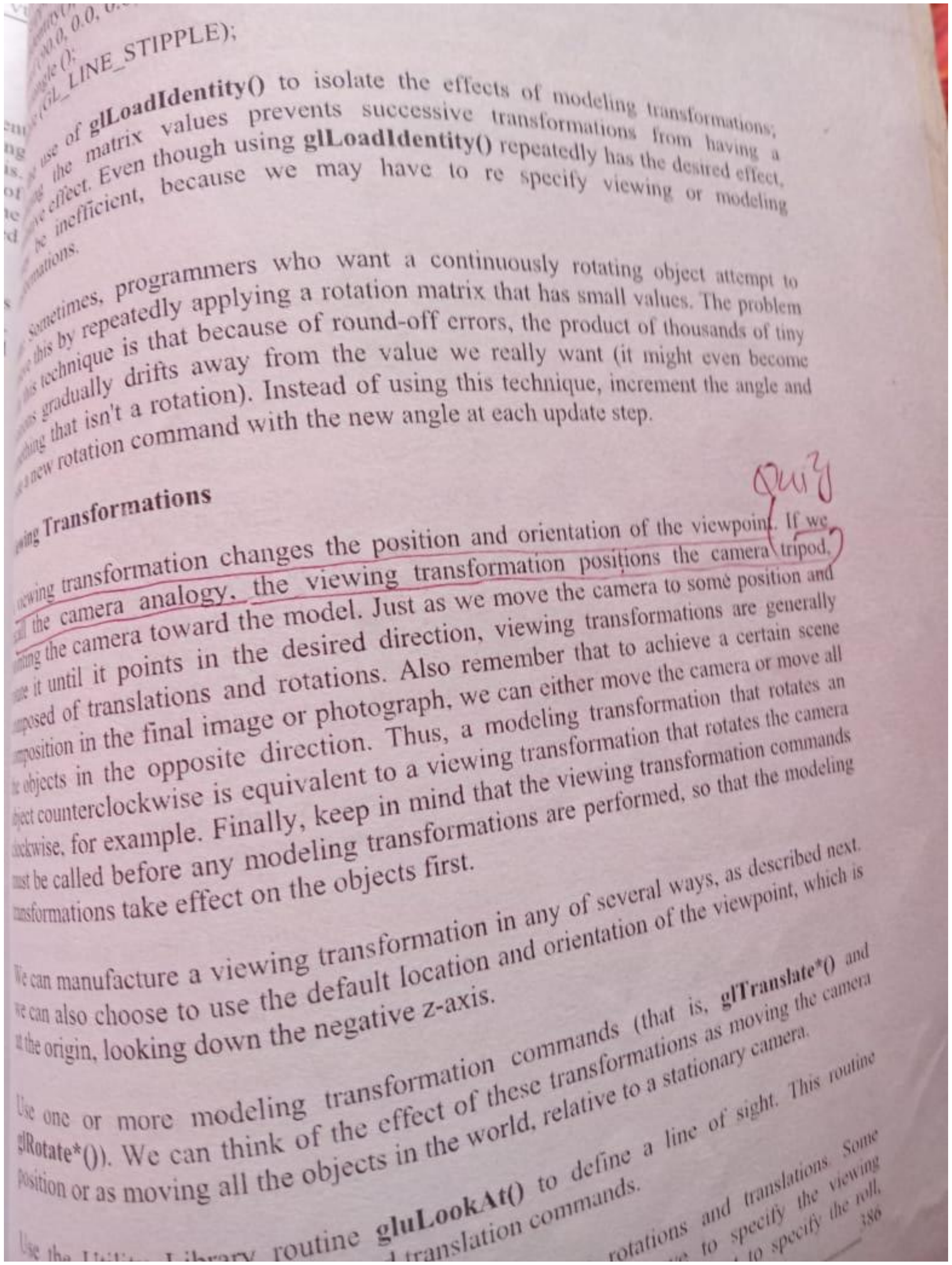
α is

β is

N

m

0



0.0, 0.0,

le ();

(GL_LINE_STIPPLE);

use of **glLoadIdentity()** to isolate the effects of modeling transformations; the matrix values prevents successive transformations from having a effect. Even though using **glLoadIdentity()** repeatedly has the desired effect, be inefficient, because we may have to re specify viewing or modeling ations.

Sometimes, programmers who want a continuously rotating object attempt to this by repeatedly applying a rotation matrix that has small values. The problem technique is that because of round-off errors, the product of thousands of tiny gradually drifts away from the value we really want (it might even become thing that isn't a rotation). Instead of using this technique, increment the angle and a new rotation command with the new angle at each update step.

ing Transformations

Quiz

viewing transformation changes the position and orientation of the viewpoint. If we the camera analogy, the viewing transformation positions the camera tripod, ting the camera toward the model. Just as we move the camera to some position and e it until it points in the desired direction, viewing transformations are generally posed of translations and rotations. Also remember that to achieve a certain scene position in the final image or photograph, we can either move the camera or move all e objects in the opposite direction. Thus, a modeling transformation that rotates an ject counterclockwise is equivalent to a viewing transformation that rotates the camera ckwise, for example. Finally, keep in mind that the viewing transformation commands st be called before any modeling transformations are performed, so that the modeling nsformations take effect on the objects first.

We can manufacture a viewing transformation in any of several ways, as described next. We can also choose to use the default location and orientation of the viewpoint, which is at the origin, looking down the negative z-axis.

Use one or more modeling transformation commands (that is, **glTranslate*()** and **glRotate*()**). We can think of the effect of these transformations as moving the camera position or as moving all the objects in the world, relative to a stationary camera.

Use the Utility Library routine **gluLookAt()** to define a line of sight. This routine d translation commands.

rotations and translations. Some to specify the viewing to specify the roll, Look at ... single geometric description of a building block ... program that produced ... contained a single ... Each block was positioned in the scene: Some blocks were scattered on the floor, some were top of each other on the table, and some were assembled to make the globe. particular viewpoint had to be chosen. Obviously, we wanted to look at the corner containing the globe. But how far away from the scene - and where exactly - viewer be? We wanted to make sure that the final image of the scene contained view out the window, that a portion of the floor was visible, and that all the the scene were not only visible but presented in an interesting arrangement. and use OpenGL to accomplish these tasks: how to position and orient models in dimensional space and how to establish the location - also in three-dimensional of the viewpoint. All of these factors help determine exactly what image appears screen. We want to remember that the point of computer graphics is to create a dimensional image of three-dimensional objects (it has to be two-dimensional it's drawn on a flat screen), but we need to think in three-dimensional coordinates making many of the decisions that determine what gets drawn on the screen. A mistake people make when creating three-dimensional graphics is to start too soon that the final image appears on a flat, two-dimensional screen. Avoid about which pixels need to be drawn, and instead try to visualize three-dimensional space. Create your models in some three-dimensional universe that lies deep your computer, and let the computer do its job of calculating which pixels to color.

es of three computer operations convert an object's three-dimensional coordinates to positions on the screen. Transformations, which are represented by matrix Paper cation, include modeling, viewing, and projection operations. Such operations rotation, translation, scaling, reflecting, orthographic projection, and perspective ection. Generally, we use a combination of several transformations to draw a scene. the scene is rendered on a rectangular window, objects (or parts of objects) that lie side the window must be clipped. In three-dimensional Computer graphics, clipping urs by throwing out objects on one side of a clipping plane.

Mcqs

Finally, a correspondence must be established between the transformed coordinates and screen pixels. This is known as a *viewport* transformation.

glMatrixMode

The **glMatrixMode** function specifies which matrix is the current matrix.

```
void glMatrixMode(
  GLenum mode
);
```

Parameters

mode

The matrix stack that is the target for subsequent matrix operations. The *mode* parameter can assume one of three values:

Left to Right

Value	Meaning
GL_MODELVIEW	Applies subsequent matrix operations to the modelview matrix stack.
GL_PROJECTION	Applies subsequent matrix operations to the projection matrix stack.
GL_TEXTURE	Applies subsequent matrix operations to the texture matrix stack.

Remarks

MCQ The **glMatrixMode** function sets the current matrix mode.

The following function retrieves information related to **glMatrixMode**:

Error Codes

The following are the error codes generated and their conditions.

Error code	Condition
GL_INVALID_ENUM	*mode* was not an accepted value.
GL_INVALID_OPERATION	MCQ **glMatrixMode** was called between a cal **glBegin** and the corresponding call to **glEnd**

glLoadIdentity

The **glLoadIdentity** function replaces the current matrix with the identity matrix.

```
void glLoadIdentity(
  void
);
```

Remarks

The **glLoadIdentity** function replaces the current matrix with the identity ma semantically equivalent to calling **glLoadMatrix** with the identity matrix cannot be changed (except within very narrow limits) without destroying that effect. In this case, the overall timing of long sections of the film is governed entirely by the dialogue. (There could be, however, considerable flexibility for more detailed timing within this fixed overall length.)

The director has room to maneuver sections. So, if the total timing for all the recorded dialogue is subtracted from the required length for the whole film, this gives the amount of time that is available without dialogue. This can then be split up in the normal way and distributed throughout the film to give the best effect.

Limited animation

MCQ) With limited animation as many repeats as possible are used within the 24 frames per second. A hold is also lengthened to reduce the number of drawings. As a rule not more than 6 drawings are produced for one second of animation. Limited animation requires almost as much skill on the part of the animator as full animation, since he must create an illusion of action with the greatest sense of economy.

Full animation

MCQs Full animation implies a large number of drawings per second of action. Some action may require that every single frame of the 24 frames within the second is animated in order to achieve an illusion of fluidity on the screen. Neither time nor money is spared on animation. As a rule only, TV commercials and feature-length animated films can afford this luxury.

MCQs Animation is expensive and time-consuming. It is not economically possible to animate more than is needed and edit the scenes later, as it is in live-action films. In cartoons the director carefully pre-times every action so that the animator works within exact limits and does no more drawings than necessary.

Ideally, the director should be able to view line test loops of the film as it progresses and so have a chance to make adjustments. But often there is no time to make corrections in limited animation and the aim is to make the animation work the first time.

Timing for Animation in general

MCQs Timing in animation is an elusive subject. It only exists whilst the film is being projected, in the same way that a melody only exists when it is being played. A melody is more easily appreciated by listening to it than by trying to explain it in words. So with cartoon timing, it is difficult to avoid using a lot of words to explain what may seem fairly simple when seen on the screen.

Timing is also a dangerous factor to try to formulate—something which works in one situation or in one mood may not work at all in another situation or mood. The only real criterion for timing is: if it works effectively on the screen it is good, if it doesn't, it isn't.

d through the following pages you can see a better way to

Overview: The Camera Analogy

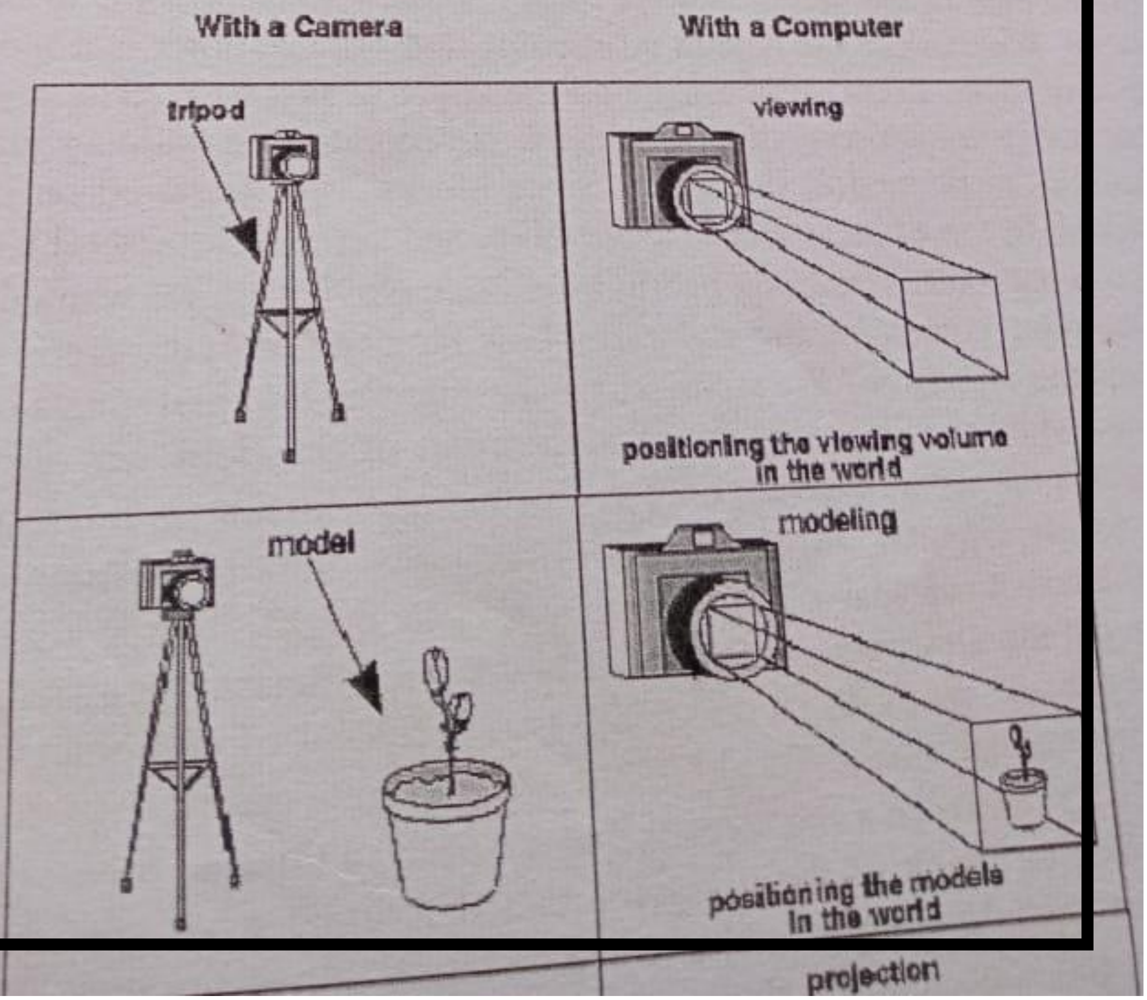
The transformation process to produce the desired scene for viewing is analogous taking a photograph with a camera. As shown in Figure 1, the steps with a camera (o computer) might be the following. Set up your tripod and pointing the camera at the sce (viewing transformation).

Arrange the scene to be photographed into the desired composition (mode transformation).

Choose a camera lens or adjust the zoom (projection transformation).

Determine how large we want the final photograph to be - for example, we might w enlarged (viewport transformation).

After these steps are performed, the picture can be snapped or the scene can be draw


With a Camera
With a Computer
tripod
viewing
positioning the viewing volume in the world
model
modeling
positioning the models in the world
projection

aggerated to be made 'animatable' and to express ideas, feelings ects? The timing mainly described is that which is used in so- MCQs or 'full' animation. To cover all possible kinds of timing in all nimation would be quite impossible.

rough

in image of false god with worshipping it.

loms "God saw how corrupt th had become, – "

k so long that corrupt- ity.

ll the people on earth ted their ways."

ck to Noah. (Ark ll in background)

to Noah – "

s going up ramp

the ark two of tures, male and

nimals going up

alive with you".

Nevertheless we hope to provide a basic understanding of how timing in animation is ultimately based on timing in nature and how, from this starting point, it is possible to apply such a difficult and invisible concept to the maximum advantage in film animation.

Animation Principles

What is good timing?

MCQs Timing is the part of animation which gives *meaning* to movement. Movement can easily be achieved by drawing the same thing in two different positions and inserting a number of other drawings between the two. The result on the screen will be movement, but it will not be animation. In nature, things do not just *move*. Newton's first law of motion stated that things do not move unless a force acts upon them. So in animation the movement itself is of secondary importance; the vital factor is how the action expresses the underlying causes of the movement. With inanimate objects these causes may be natural forces, mainly gravity. With xternal forces can

...software inter... ...software with which ...frame buffer. Op... ...programmers can create high-quality still and animated three-dimensional color images.

Red, green, blue

...here Applicable:

...penGL is built for compatibility across hardware and operating systems. This ...tecture makes it easy to port OpenGL programs from one system to another. While ...h operating system has unique requirements, the OpenGL code in many programs can ...e used as is.

Developer Audience:

...esigned for use by C/C++ programmers

Run-time Requirements:

OpenGL can run on Linux and all versions of 32 bit Microsoft Windows.

Most Widely Adopted Graphics Standard

OpenGL is the premier environment for developing portable, interactive 2D and 3D graphics applications. Since its introduction in 1992, OpenGL has become the industry's most widely used and supported 2D and 3D graphics application programming interface (API), bringing thousands of applications to a wide variety of computer platforms. OpenGL fosters innovation and speeds application development by incorporating a broad set of rendering, texture mapping, special effects, and other powerful visualization functions. Developers can leverage the power of OpenGL across all popular desktop and workstation platforms, ensuring wide application deployment.

High Visual Quality and Performance

Any visual computing application requiring maximum performance-from 3D animation to CAD to visual simulation-can exploit high-quality, high-performance OpenGL capabilities. These capabilities allow developers in diverse markets such as broadcasting, CAD/CAM/CAE, entertainment, medical imaging, and virtual reality to produce and display incredibly compelling 2D and 3D graphics.

Developer-Driven Advantages

✓ **Industry standard**

An independent consortium, the OpenGL Architecture Review Board, guides the OpenGL specification. With broad industry support, OpenGL is the only truly open, vendor-neutral, multiplatform graphics standard.

✓ **Stable**

OpenGL implementations have been available for more than seven years on a wide variety of platforms. Additions to the specification are well controlled, and proposed updates are announced in time for developers to adopt changes

Backward compatibility requirements ensure that existing applications do not become obsolete.

Reliable and portable

All OpenGL applications produce consistent visual display results on any OpenGL API-compliant hardware, regardless of operating system or windowing system.

Evolving

Because of its thorough and forward-looking design, OpenGL allows ne hardware innovations to be accessible through the API via the OpenGL extensi mechanism. In this way, innovations appear in the API in a timely fashion, lett application developers and hardware vendors incorporate new features into normal product release cycles.

Scalable

OpenGL API-based applications can run on systems ranging from co electronics to PCs, workstations, and supercomputers. As a result, applicat scale to any class of machine that the developer chooses to target.

Easy to use

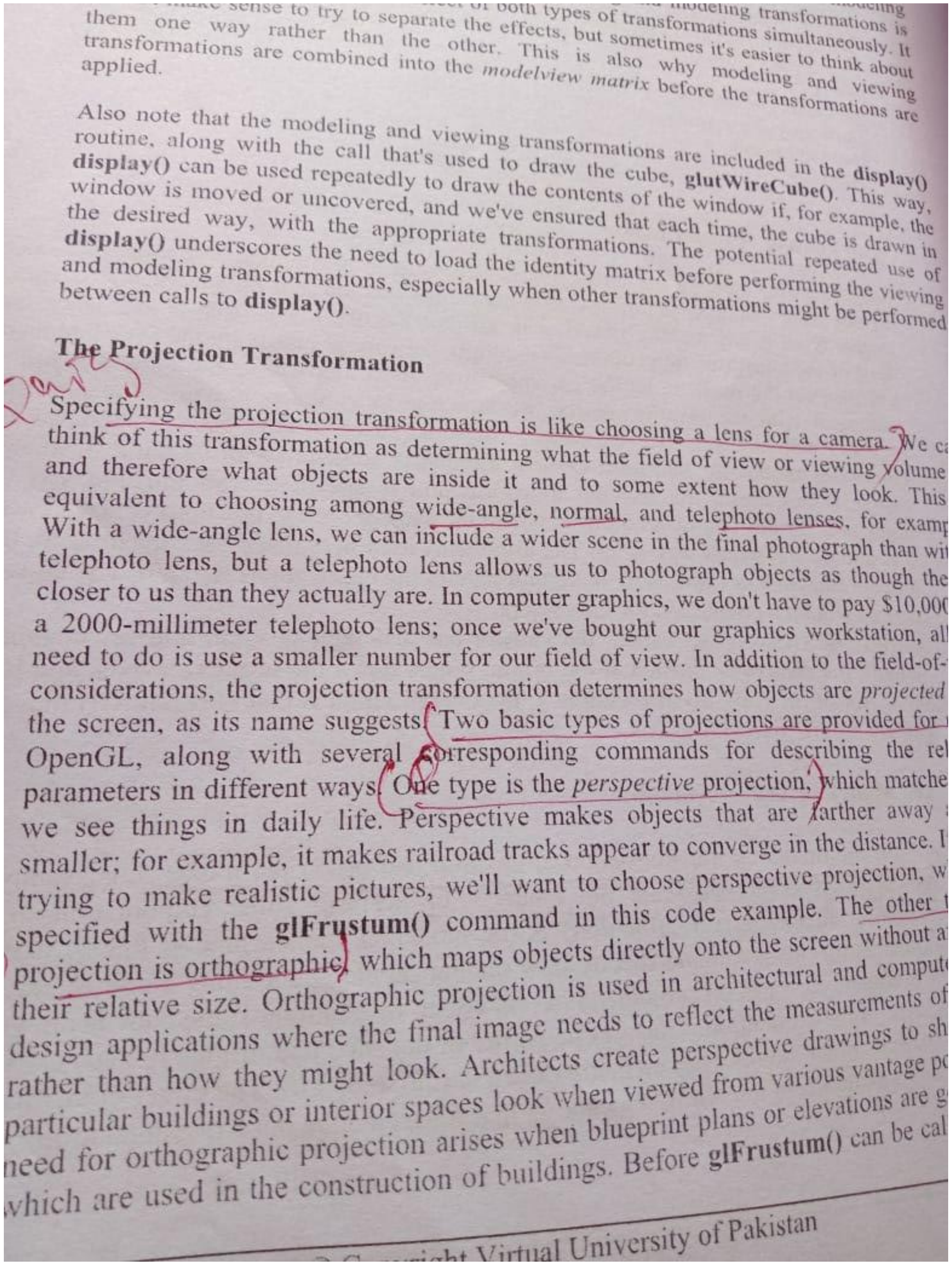
OpenGL is well structured with an intuitive design and logical Efficient OpenGL routines typically result in applications with fewer than those that make up programs generated using other graphic packages. In addition, OpenGL drivers encapsulate informati underlying hardware, freeing the application developer from havin specific hardware features.

Well-documented

Numerous books have been published about OpenGL, and a gre code is readily available, making information about OpenGL easy to obtain.

Simplifies Software Development, Speeds Time-to-Market

GL routines simplify the development of graphics softwar
polygon to the creation of the



… of both types of transformations simultaneously. It … make sense to try to separate the effects, but sometimes it's easier to think about them one way rather than the other. This is also why modeling and viewing transformations are combined into the *modelview matrix* before the transformations are applied.

Also note that the modeling and viewing transformations are included in the **display()** routine, along with the call that's used to draw the cube, **glutWireCube()**. This way, **display()** can be used repeatedly to draw the contents of the window if, for example, the window is moved or uncovered, and we've ensured that each time, the cube is drawn in the desired way, with the appropriate transformations. The potential repeated use of **display()** underscores the need to load the identity matrix before performing the viewing and modeling transformations, especially when other transformations might be performed between calls to **display()**.

The Projection Transformation

Specifying the projection transformation is like choosing a lens for a camera. We ca think of this transformation as determining what the field of view or viewing volume and therefore what objects are inside it and to some extent how they look. This equivalent to choosing among wide-angle, normal, and telephoto lenses, for examp With a wide-angle lens, we can include a wider scene in the final photograph than wi telephoto lens, but a telephoto lens allows us to photograph objects as though the closer to us than they actually are. In computer graphics, we don't have to pay $10,000 a 2000-millimeter telephoto lens; once we've bought our graphics workstation, al need to do is use a smaller number for our field of view. In addition to the field-of- considerations, the projection transformation determines how objects are *projected* the screen, as its name suggests. Two basic types of projections are provided for OpenGL, along with several corresponding commands for describing the rel parameters in different ways. One type is the *perspective* projection, which matche we see things in daily life. Perspective makes objects that are farther away smaller; for example, it makes railroad tracks appear to converge in the distance. I trying to make realistic pictures, we'll want to choose perspective projection, w specified with the **glFrustum()** command in this code example. The other t projection is orthographic, which maps objects directly onto the screen without a their relative size. Orthographic projection is used in architectural and comput design applications where the final image needs to reflect the measurements of rather than how they might look. Architects create perspective drawings to sh particular buildings or interior spaces look when viewed from various vantage p need for orthographic projection arises when blueprint plans or elevations are g which are used in the construction of buildings. Before **glFrustum()** can be cal

Overview: The Camera Analogy

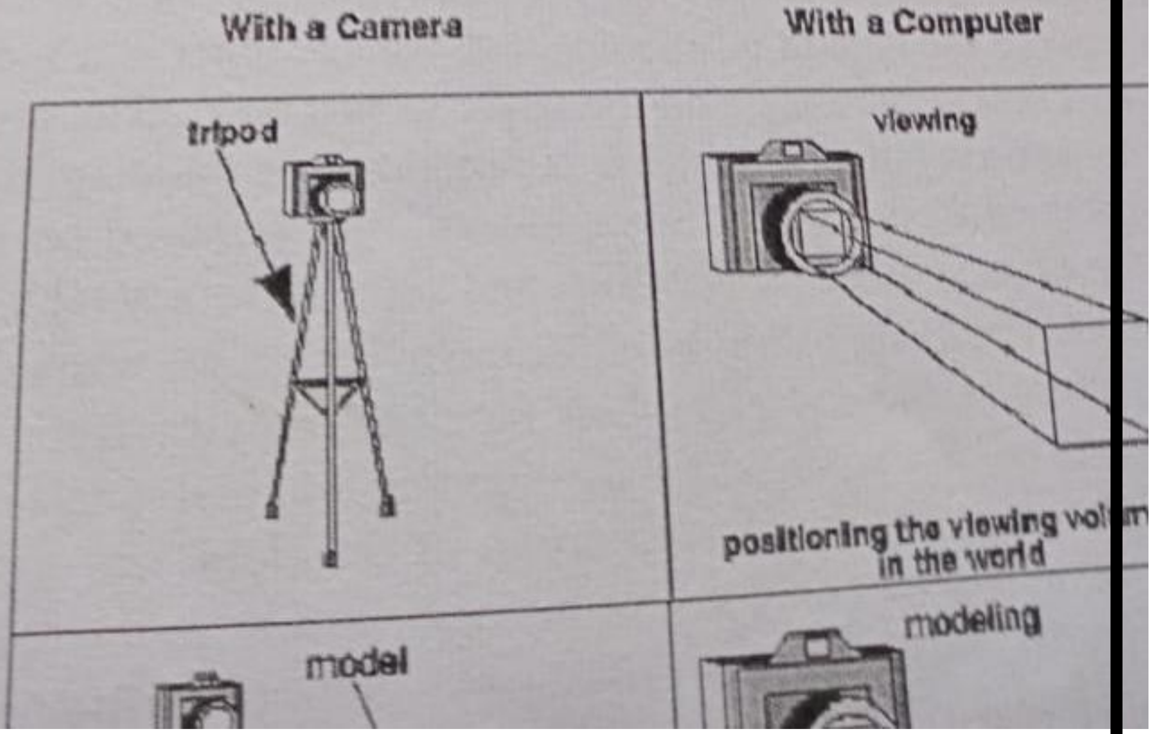
The transformation process to produce the desired scene for viewing is a taking a photograph with a camera. As shown in Figure 1, the steps with computer) might be the following. Set up your tripod and pointing the cam (viewing transformation).

Arrange the scene to be photographed into the desired composi transformation).

Choose a camera lens or adjust the zoom (projection transformation).

Determine how large we want the final photograph to be - for example, w enlarged (viewport transformation).

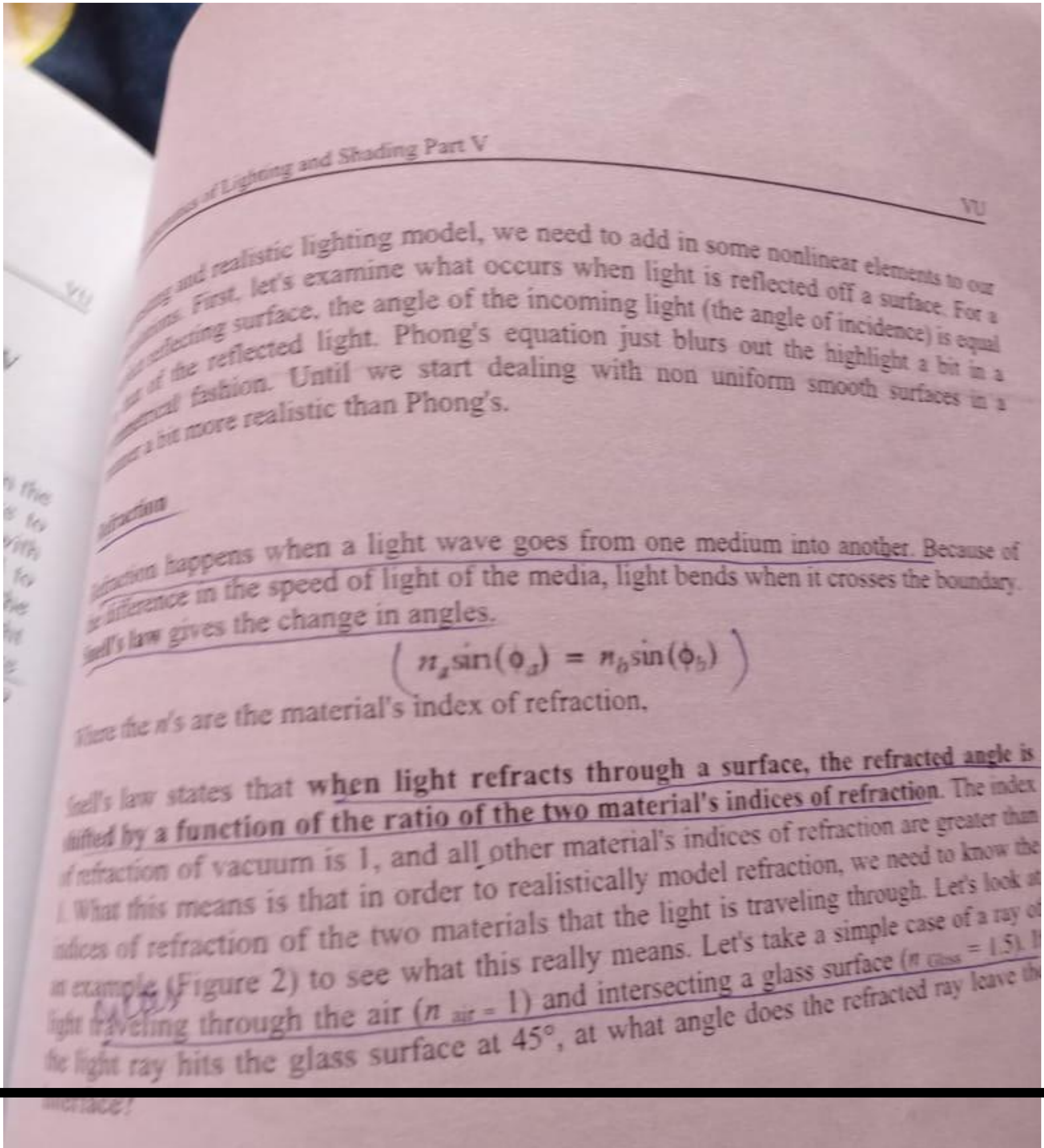
After these steps are performed, the picture can be snapped or the scene



As a ... ure, the main purpose of OpenGL is to render two- and three-dimensional objects into a frame buffer. These objects are described as sequences of vertices (that define geometric objects) or pixels (that define images). OpenGL performs several processes on this data to convert it to pixels to form the final desired image in the frame buffer.

The following topics present a global view of how OpenGL works:

- **Primitives and Commands** discusses points, line segments, and polygons as the basic units of drawing; and the processing of commands.
- **OpenGL Graphic Control** describes which graphic operations OpenGL controls and which it does not control.
- **Execution Model** discusses the client/server model for interpreting OpenGL commands.



and realistic lighting model, we need to add in some nonlinear elements to our First, let's examine what occurs when light is reflected off a surface. For a reflecting surface, the angle of the incoming light (the angle of incidence) is equal of the reflected light. Phong's equation just blurs out the highlight a bit in a fashion. Until we start dealing with non uniform smooth surfaces in a a bit more realistic than Phong's.

Refraction

Refraction happens when a light wave goes from one medium into another. Because of the difference in the speed of light of the media, light bends when it crosses the boundary. Snell's law gives the change in angles.

$$\left(n_a \sin(\phi_a) = n_b \sin(\phi_b) \right)$$

Where the n's are the material's index of refraction.

Snell's law states that **when light refracts through a surface, the refracted angle is shifted by a function of the ratio of the two material's indices of refraction**. The index of refraction of vacuum is 1, and all other material's indices of refraction are greater than 1. What this means is that in order to realistically model refraction, we need to know the indices of refraction of the two materials that the light is traveling through. Let's look at an example (Figure 2) to see what this really means. Let's take a simple case of a ray of light traveling through the air ($n_{air} = 1$) and intersecting a glass surface ($n_{Glass} = 1.5$). If the light ray hits the glass surface at 45°, at what angle does the refracted ray leave the interface?

Lecture No.38 Bezier Curves

MCQs

The Bezier curve is an important part of almost every computer-graphics illust program and computer-aided design system in use today. It is used in many ways designing the curves and surfaces of automobiles to defining the shape of letters i fonts. And because it is numerically the most stable of all the polynomial-based used in these applications, the Bezier curve is the ideal standard for representing th complex piecewise polynomial curves. Quiz

MCQ's

In the early 1960s, Peter Bezier began looking for a better way to define cu surfaces one that would be useful to a design engineer. He was familiar with the Ferguson and Coons and their parametric cubic curves and bicubic surfaces. I hese did not offer an intuitive way to alter and control shape. The results of esearch led to the curves and surfaces that bear his name and became pa NISURF system. The French automobile manufacturer, Renault used UN esign the sculptured surfaces of many of its products.

Geometric Construction

can draw a Bezier curve using a simple recursive geometric construction. constructing a second-degree curve. We select three points A, B, C so that ent to the curve at A, and BC is tangent at C. The curve begins at A an any ratio, u_i, we construct points D and E so that

```
    ...rge, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize (500, 500);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);
    init ();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}
```

Try This

Try adding a moon to the planet. Or try several moons and additional planets. Hint: Use **glPushMatrix()** and **glPopMatrix()** to save and restore the position and orientation of the coordinate system at appropriate moments. If we're going to draw several moons around a planet, we need to save the coordinate system prior to positioning each moon and restore the coordinate system after each moon is drawn.

Try tilting the planet's axis.

Building an Articulated Robot Arm

This section discusses a program that creates an articulated robot arm with two or more segments. The arm should be connected with pivot points at the shoulder, elbow, or other joints. Figure 25 shows a single joint of such an arm.

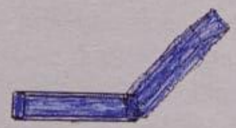
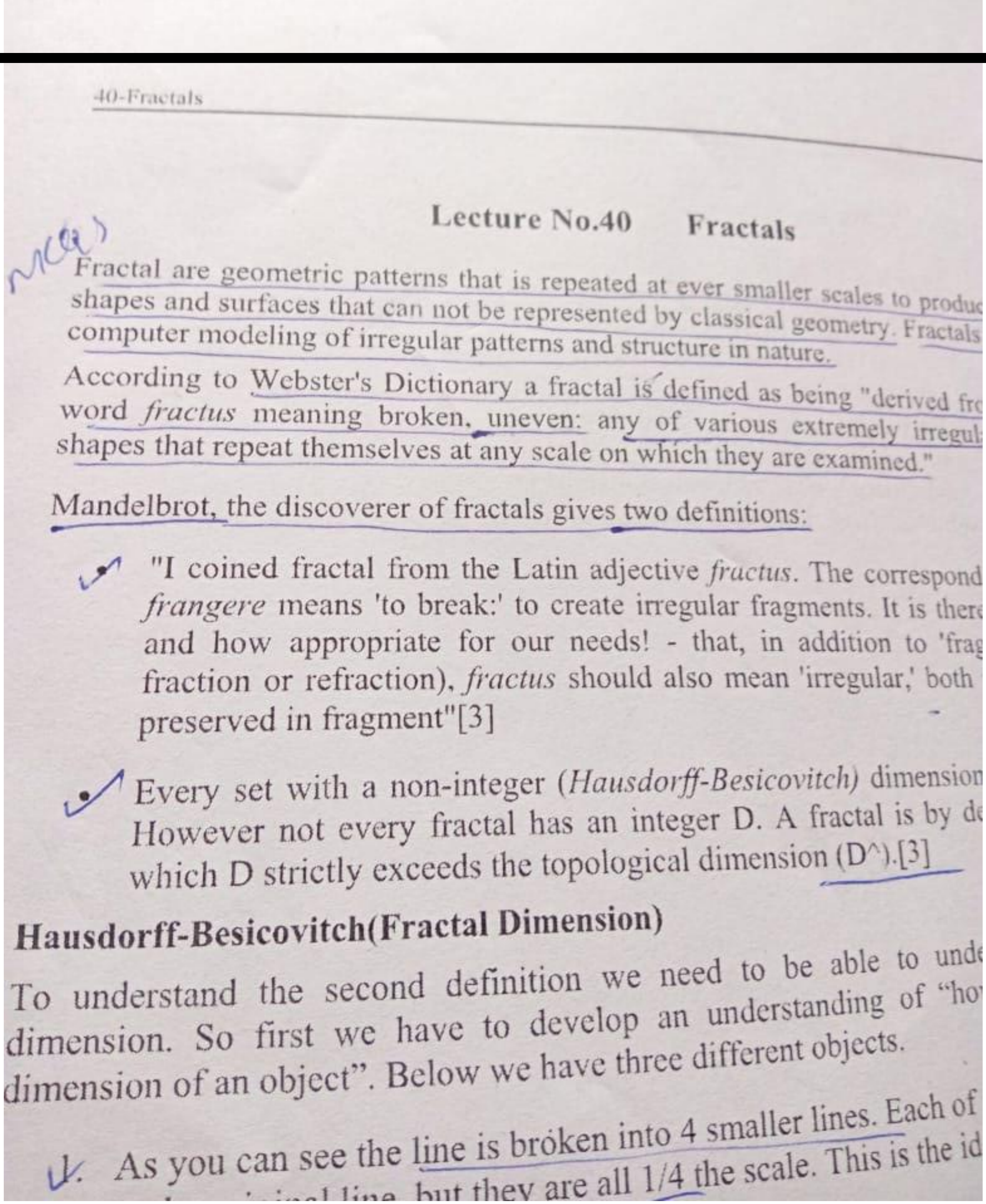


Figure 25 : Robot Arm

can use a scaled cube as a segment of the robot arm, but first we must call the ropriate modeling transformations to orient each segment. Since the origin of the local dinate system is initially at the center of the cube, we need to move the local



Lecture No.40 Fractals

MCQ's)

Fractal are geometric patterns that is repeated at ever smaller scales to produc shapes and surfaces that can not be represented by classical geometry. Fractals computer modeling of irregular patterns and structure in nature.

According to Webster's Dictionary a fractal is defined as being "derived fr word *fractus* meaning broken, uneven: any of various extremely irregul shapes that repeat themselves at any scale on which they are examined."

Mandelbrot, the discoverer of fractals gives two definitions:

- "I coined fractal from the Latin adjective *fractus*. The correspond *frangere* means 'to break:' to create irregular fragments. It is there and how appropriate for our needs! - that, in addition to 'frag fraction or refraction), *fractus* should also mean 'irregular,' both preserved in fragment"[3]
- Every set with a non-integer (*Hausdorff-Besicovitch*) dimension However not every fractal has an integer D. A fractal is by de which D strictly exceeds the topological dimension (D^).[3]

Hausdorff-Besicovitch(Fractal Dimension)

To understand the second definition we need to be able to unde dimension. So first we have to develop an understanding of "ho dimension of an object". Below we have three different objects.

1. As you can see the line is broken into 4 smaller lines. Each of inal line, but they are all 1/4 the scale. This is the id

where the kc, kl, and kq parameters are the constant, linear, and quadratic attenuation constants respectively; to get the "real" attenuation factor, we can set kc to one and the others to zero. The attenuation factor is multiplied by the light diffuse and specular values. Typically, each light will have a set of these parameters for itself. The lighting equation with the attenuation factor looks like this.

M(25) $$i_{total} = i_a + \sum f_{atten}(i_d + i_s)$$

Figure 2 shows a sample of what attenuation looks like. This image is the same as the one shown in Figure 1, but with light attenuation added.

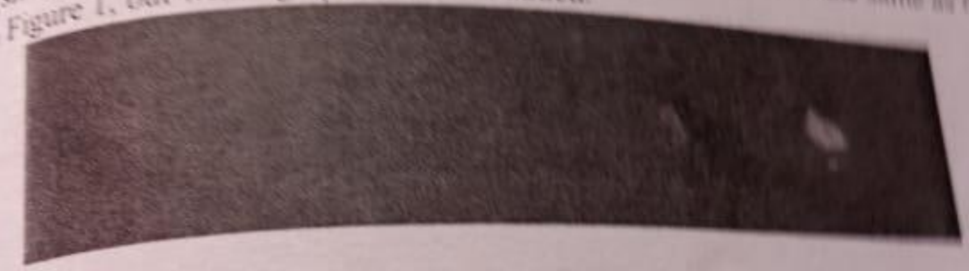


Figure 2: A scene with light attenuation. The white sphere is the light position.

Schlick's simplification for the Specular Exponential Term

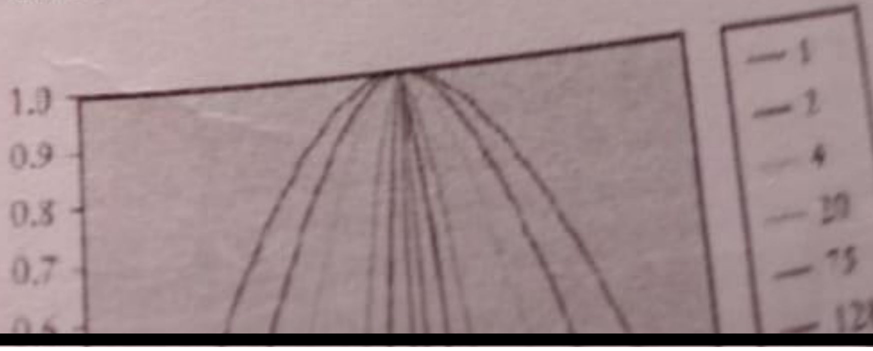
Real-time graphics programmers are always looking for simplifications. We've probably gathered that there's no such thing as the "correct" lighting equation, just a series of hacks to make things look right with as little computational effort as possible. Schlick [SCHLICK 1994] suggested a replacement for the exponential term since that's a fairly expensive operation. If we define part of our specular light term as follows,

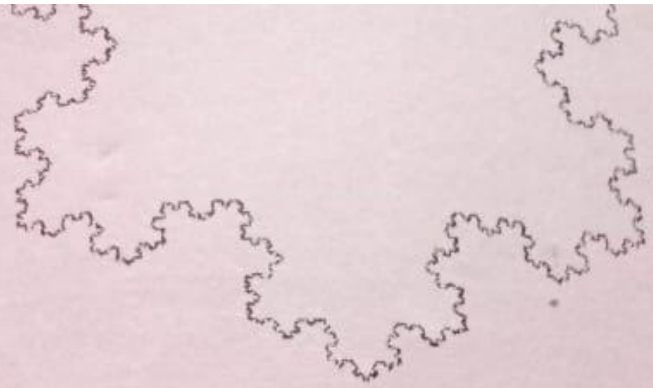
$$(S)^{m_s}$$

where S is either the Phong or Blinn-Phong flavor of the specular lighting equation, then Schlick's simplification is to replace the preceding part of the specular equation with

M(25) ✓ $$\frac{S}{m_s - m_s S + S}$$

Which eliminates the need for an exponential term, At first glance, a plot of Schlick's function looks very similar to the exponential equation (Figure 3).





Length = 12.6420

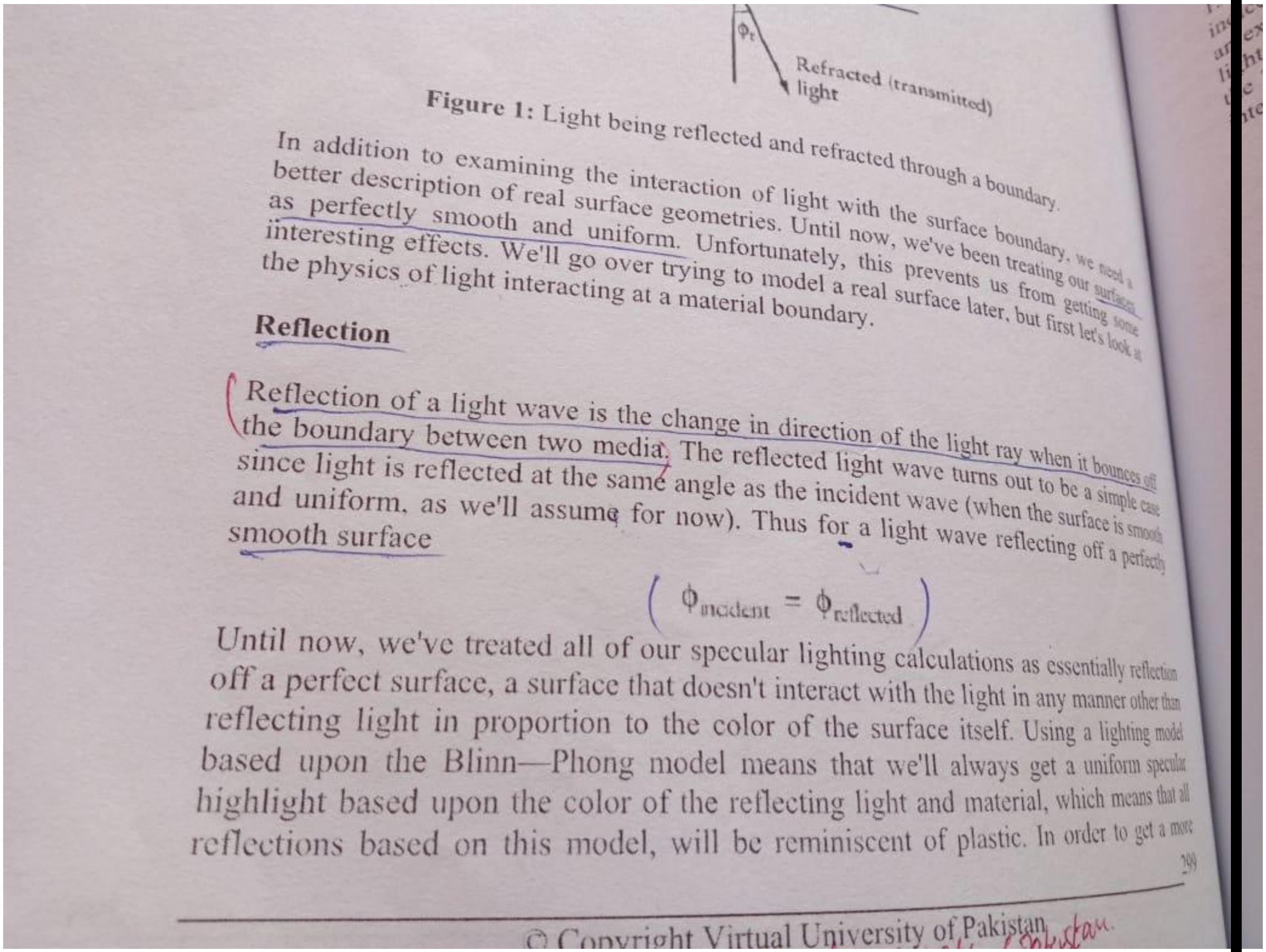
MCQ

As we continue adding edges, the length of the curve increases. If we add edges forever, the length of the curve reaches infinity, but the whole curve nevertheless covers a finite area. The curve is infinitely detailed. No matter how closely we zoom into the image, it always shows up more detail.

Self Similarity

So what do these mathematical curiosities have to do with the real world? Well, everything as it turns out. Such objects turn up all the time in the natural world. Animals, MCQs plants, rocks, crystals and liquids all exhibit fractal properties and self similarity. Let's take a look at a common plant, the fern. The fern is typical of many plants in that it exhibits self similarity. A fern consists of a leaf, which is made up from many similar, but smaller leaves, each of which, in turn, is made from even smaller leaves. The closer we look the more detail we see.

The following figure is a standard fern, which we may well find while being dragged on long walks in the country by your parents long before we are able to fully appreciate the beauty of nature. We will see the overall theme of repeating leaves. Each smaller leaf looks similar to the larger leaf.



ϕ_t
Refracted (transmitted) light

Figure 1: Light being reflected and refracted through a boundary.

In addition to examining the interaction of light with the surface boundary, we need a better description of real surface geometries. Until now, we've been treating our surfaces as perfectly smooth and uniform. Unfortunately, this prevents us from getting some interesting effects. We'll go over trying to model a real surface later, but first let's look at the physics of light interacting at a material boundary.

Reflection

Reflection of a light wave is the change in direction of the light ray when it bounces off the boundary between two media. The reflected light wave turns out to be a simple case since light is reflected at the same angle as the incident wave (when the surface is smooth and uniform, as we'll assume for now). Thus for a light wave reflecting off a perfectly smooth surface

$$\phi_{\text{incident}} = \phi_{\text{reflected}}$$

Until now, we've treated all of our specular lighting calculations as essentially reflection off a perfect surface, a surface that doesn't interact with the light in any manner other than reflecting light in proportion to the color of the surface itself. Using a lighting model based upon the Blinn—Phong model means that we'll always get a uniform specular highlight based upon the color of the reflecting light and material, which means that all reflections based on this model, will be reminiscent of plastic. In order to get a more

diffuse reflectance, ambient reflectance isn't affected by the position of the viewpo
real-world objects, diffuse and ambient reflectance are normally the same color.
reason, OpenGL provides we with a convenient way of assigning the same valu
simultaneously with **glMaterial*()**:

```
GLfloat mat_amb_diff[] = { 0.1, 0.5, 0.8, 1.0 };
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, mat_am
```

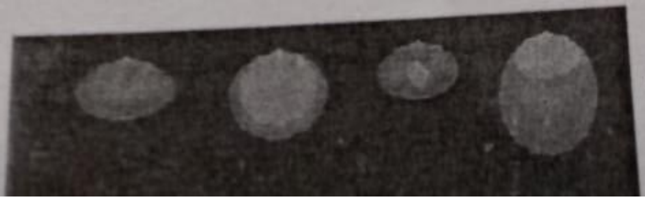
In this example, the RGBA color (0.1, 0.5, 0.8, 1.0) - a deep blue color - rep
current ambient and diffuse reflectance for both the front- and
back-facing polygons.

Specular Reflection

MCQ

Specular reflection from an object produces highlights. Unlike ambient
reflection, the amount of specular reflection seen by a viewer does depend o
of the viewpoint - it's brightest along the direct angle of reflection. To see
looking at a metallic ball outdoors in the sunlight. As we move our head
created by the sunlight moves with us to some extent. However, if we mov
much, we lose the highlight entirely.

OpenGL allows us to set the effect that the material has on reflec
GL_SPECULAR) and control the size and brightness of the h
GL_SHININESS). We can assign a number in the range of [
GL_SHININESS - the higher the value, the smaller and brighter (mo
highlight.



No ambient reflection.

glTranslated, glTranslatef

The glTranslated and glTranslatef functions multiply the current matrix by a translation matrix.

```
void glTranslated(
GLdouble x,
GLdouble y,
GLdouble z
);

void glTranslatef(
GLfloat x,
GLfloat y,
GLfloat z
);
```

Parameters *x*, *y*, *z*

The *x*, *y*, and *z* coordinates of a translation vector.

Remarks

The **glTranslate** function produces the translation specified by (*x*, *y*, *z*). The translation vector is used to compute a 4x4 translation matrix:

$$\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The current matrix (see **glMatrixMode**) is multiplied by this translation matrix, with the product replacing the current matrix. That is, if M is the current matrix and T is the translation matrix, then M is replaced with M•T. MCQ's

If the matrix mode is either GL_MODELVIEW or GL_PROJECTION, all objects drawn after **glTranslate** is called are translated. Use **glPushMatrix** and **glPopMatrix** to save and restore the untranslated coordinate system.

The following functions retrieve information related to **glTranslated** and **glTranslatef**

Error Codes

modeling transformations are *discussed* first, even if viewing transformations are actu *issued* first. This order for discussion also matches the way many programmers th when planning their code: Often, they write all the code necessary to compose the sc which involves transformations to position and orient objects correctly relative to other. Next, they decide where they want the viewpoint to be relative to the scene th composed, and then they write the viewing transformations accordingly.

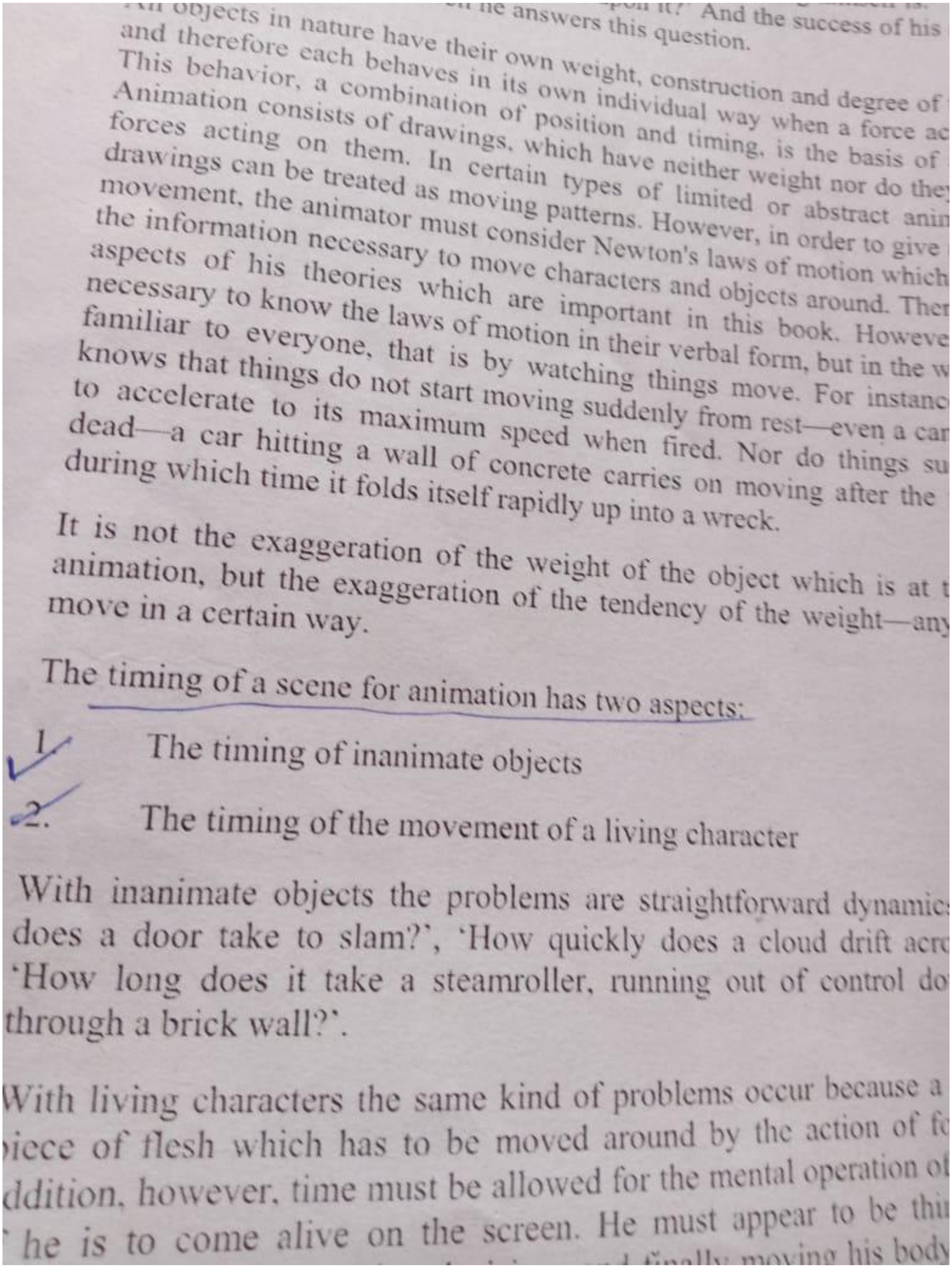
Modeling Transformations

The three OpenGL routines for modeling transformations are **glTransl** **glRotate*()**, and **glScale*()**. As we might suspect, these routines transform an ob coordinate system, if we're thinking of it that way) by moving, rotating, str shrinking, or reflecting it. All three commands are equivalent to producing an app translation, rotation, or scaling matrix, and then calling **glMultMatrix*()** with tha as the argument. However, these three routines might be faster tha **glMultMatrix*()**. OpenGL automatically computes the matrices for we. In the c summaries that follow, each matrix multiplication is described in terms of what the vertices of a geometric object using the fixed coordinate system approac terms of what it does to the local coordinate system that's attached to an object.

*void **glTranslate**{fd}(TYPEx, TYPE y, TYPEz);*

Multiplies the current matrix by a matrix that moves (translates) an object by
, and z values (or moves the local coordinate system by the
ame amounts).

igure 5 shows the effect of **glTranslate*()**.



upon it?' And the success of his he answers this question.

All objects in nature have their own weight, construction and degree of and therefore each behaves in its own individual way when a force ac This behavior, a combination of position and timing, is the basis of Animation consists of drawings, which have neither weight nor do they forces acting on them. In certain types of limited or abstract anin drawings can be treated as moving patterns. However, in order to give movement, the animator must consider Newton's laws of motion which the information necessary to move characters and objects around. Ther aspects of his theories which are important in this book. Howeve necessary to know the laws of motion in their verbal form, but in the w familiar to everyone, that is by watching things move. For instanc knows that things do not start moving suddenly from rest—even a car to accelerate to its maximum speed when fired. Nor do things su dead—a car hitting a wall of concrete carries on moving after the during which time it folds itself rapidly up into a wreck.

It is not the exaggeration of the weight of the object which is at t animation, but the exaggeration of the tendency of the weight—any move in a certain way.

The timing of a scene for animation has two aspects:

1. The timing of inanimate objects
2. The timing of the movement of a living character

With inanimate objects the problems are straightforward dynamic does a door take to slam?', 'How quickly does a cloud drift acro 'How long does it take a steamroller, running out of control do through a brick wall?'.

With living characters the same kind of problems occur because a piece of flesh which has to be moved around by the action of fo ddition, however, time must be allowed for the mental operation of he is to come alive on the screen. He must appear to be thi

You can also [illegible] execute display lists the preceding functions. If any other OpenGL function is called bet glEnd, the error flag is set and the function is ignored.

Regardless of the value chosen for *mode* in **glBegin**, there number of vertices you can define between **glBegin** and **glEnd** quadrilaterals, and polygons that are incompletely specified incomplete specification results when either too few vertices are even a single primitive or when an incorrect multiple of vertice incomplete primitive is ignored; the complete primitives are drawn

The minimum specification of vertices for each primitive is:

mcqs

Minimum number of vertices	Type of primitive
1	Point
2	Line
3	Triangle
4	Quadrilateral
3	Polygon

can
be made to fit in ... ues will

MCQS $$i_{total} = i_a + \sum(i_d + i_s)$$

Our final scene with ambient, diffuse, and (Blinn's) specular light contributions (w white light above and to the left of the viewer) looks like Figure 1.

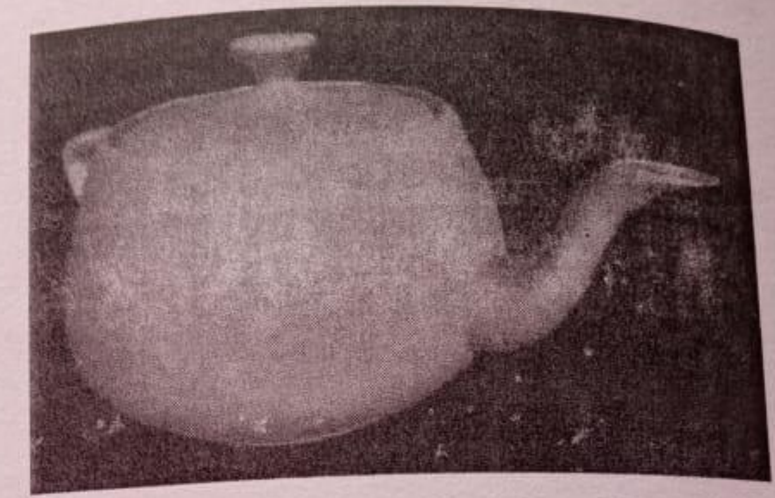


Figure 1: A combination of ambient, diffuse, and specular illumination.

It may be surprising to discover that there's more than one way to calculate the shading of an object, but that's because the model is empirical, and there's no correct way, just different ways that all have tradeoffs. Until now though, the only lighting equation we've been able to use has been the one we just formulated. Most of the interesting work in computer graphics is tweaking that equation, or in some cases, throwing it out altogether and coming up with something new.

The next sections will discuss some refinements and alternative ways of calculating the various coefficients of the lighting equation.

Light Attenuation

Light in the real world loses its intensity as the inverse square of the distance from t light source to the surface being illuminated. However, when put into practice, t seemed to drop off the light intensity in too abrupt a manner and then not to vary much after the light was far away. An empirical model was developed that seems to satisfactory results. This is the attenuation model that's used in OpenGL and DirectX. *f*atten factor is the attenuation factor. The distance *d* between the light and the vert always positive. The attenuation factor is calculated by the following equation:

MCQS $$f_{atten} = 1/(k_c + k_l d + k_q d^2)$$

$y' = x \sin\theta + y \cos\theta$

$z' = z$

by Cyclic permutation

Rotation about x-axis
(i.e. in yz plane):

$x' = x$

$y' = y \cos\theta - z \sin\theta$

$z' = y \sin\theta + z \cos\theta$

and
Rotation about y-axis
(i.e. in xz plane):

$x' = z \sin\theta + x \cos\theta$

$y' = y$

$z' = z \cos\theta - x \sin\theta$

b) SCALING:-

- Coordinate transformations for scaling relative to the origin are
- X` = X . Sx
- Y` = Y . Sy
- Z` = Z . Sz

Uniform Scaling

mcqs

- We preserve the original shape of an object with a uniform scaling
- (Sx = Sy = Sz)

Differential Scaling

mcqs

- We do not preserve the original shape of an object with a differential scaling
- (Sx <> Sy <> Sz)

Scaling w.r.t. Origin

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

27.33 PROJECTION

Projection can be defined as a mapping of point P(x,y,z) onto its P`(x`,y`,z`) in the projection plane or view plane, which constitutes the display surfa

Figure 7: A soybean field showing differing reflection properties.

how the backscattering image shows a near uniform diffuse illumination, whereas ard scattering image shows a uniform dull diffuse illumination. Also note that we specular highlights and more color variation because of the shadows due to the rface whereas the backscattered image washes out the detail. In an effort to better ough surfaces, Oren and Nayar [OREN 1992] came up with a generalized version nbertian diffuse shading model that tries to account for the roughness of the They applied the Torrance—Sparrow model for rough surfaces with isotropic s and provided parameters to account for the various surface structures found in ance—Sparrow model. By comparing their model with actual data, they d their model to the terms that had the most significant impact. The Oren– ffuse shading model looks like this.

$$i_d = \frac{\rho}{\pi}E_0\cos(\theta_i)(A + B\max[0, \cos(\phi_r - \phi_i)]\sin(\alpha)\tan(\beta))$$

$$A = 1 - 0.5\frac{\sigma^2}{\sigma^2 + 0.33}$$

$$B = 0.45\frac{\sigma^2}{\sigma^2 + 0.09}$$

may look daunting, but it can be simplified to something we can appreciate if e the original notation with the notation we've already been using. ρ/π is a flectivity property, which we can replace with our surface diffuse color. E0 is a t energy term, which we can replace with our light diffuse color. And the θi st our familiar angle between the vertex normal and the light direction. Making anges gives us

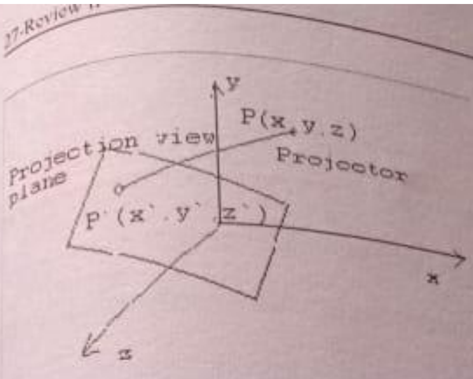
$$i_d = (m_d \otimes s_d)(\hat{n} \bullet \hat{l})(A + B\max[0, \cos(\phi_r - \phi_i)]\sin(\alpha)\tan(\beta))$$

(Oren-Nayer)

ks a lot more like the equations we've used, there are still some parameters to

α is
β is

N
m
0

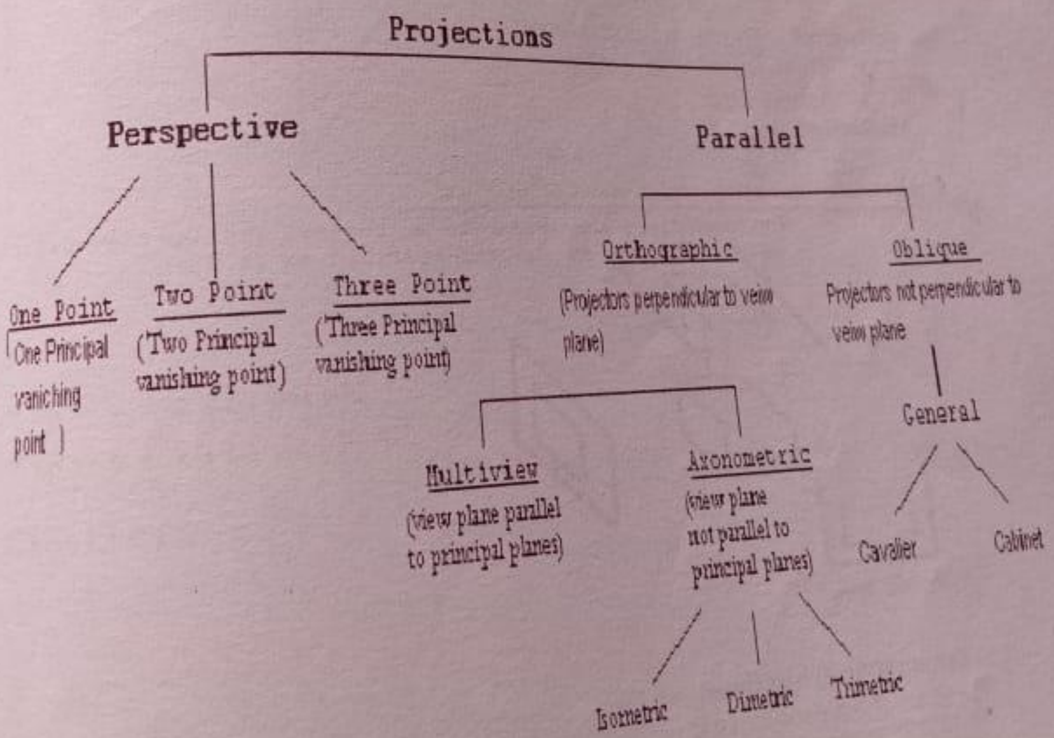


The Problem of Projection

Methods of Projection

mcqs

- Parallel Projection
- Perspective Projection


Projections
Perspective
Parallel
One Point
(One Principal vaniching point)
Two Point
(Two Principal vanishing point)
Three Point
(Three Principal vanishing point)
Orthographic
(Projectors perpendicular to veiw plane)
Oblique
Projectors not perpendicular to veiw plane
General
Multiview
(view plane parallel to principal planes)
Axonometric
(view plane not parallel to principal planes)
Cavalier
Cabinet
Isometric
Dimetric
Trimetric

Toxonomy of Projection

Mcqs

Parallel Projection is divided into Orthographic and Oblique transformations.

- Orthographic

1. Isometric

The projection plane intersects each coordinate axis in the model coordinate system at an equal distance or (the direction of projection makes equal angles with all of the three principal axes)

2. Dimetric

The direction of projection makes equal angles with exactly two of the principal axes

3. Trimetric

The direction of projection makes unequal angles with the three principal axes

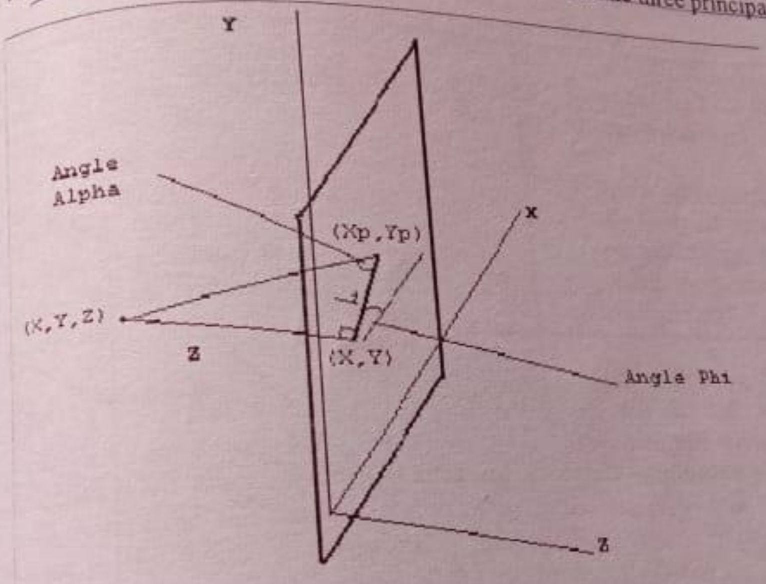


Figure: Oblique Projection of coordinate position (x,y,z) to position (Xp,Yp) on the view plane

- $Xp = x + z \, (L1 \cos(\Phi))$
- $Yp = y + z \, (L1 \sin(\Phi))$

Where $L1 = L/z$

where the kc, kl, and kq parameters are the constant, linear, and quadratic attenuation constants respectively, to get the "real" attenuation factor, we can set kq to one and the others to zero. The attenuation factor is multiplied by the light diffuse and specular values. Typically, each light will have a set of these parameters for itself. The lighting equation with the attenuation factor looks like this.

MCQS $$i_{total} = i_a + \sum f_{atten}(i_d + i_s)$$

Figure 2 shows a sample of what attenuation looks like. This image is the same as the one shown in Figure 1, but with light attenuation added.

Figure 2: A scene with light attenuation. The white sphere is the light position.

Schlick's simplification for the Specular Exponential Term

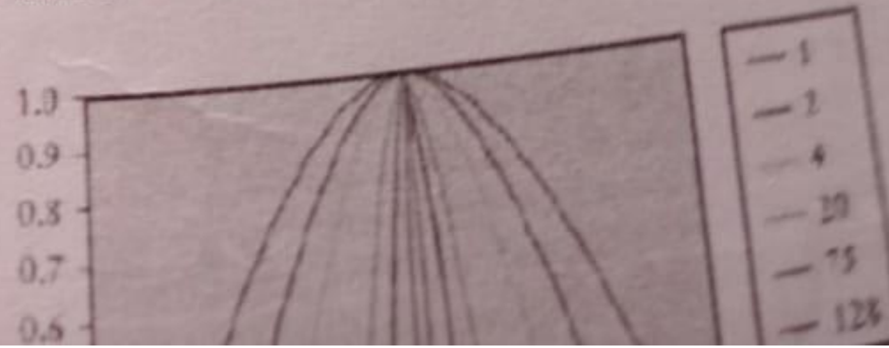
Real-time graphics programmers are always looking for simplifications. We've probably gathered that there's no such thing as the "correct" lighting equation, just a series of hacks to make things look right with as little computational effort as possible. Schlick [SCHLICK 1994] suggested a replacement for the exponential term since that's a fairly expensive operation. If we define part of our specular light term as follows,

$$(S)^{m_s}.$$

where S is either the Phong or Blinn-Phong flavor of the specular lighting equation, then Schlick's simplification is to replace the preceding part of the specular equation with

MCQS ✓ $$\frac{S}{m_s - m_s S + S}$$

Which eliminates the need for an exponential term, At first glance, a plot of Schlick's function looks very similar to the exponential equation (Figure 3).



Lecture No.29 Mathematics of Lighting and Shading Part III

Traditional 3D Hardware-Accelerated Lighting Models

We will now take a look at the traditional method of calculating lighting in hardware—a method that we'll find is sufficient for most of our needs. The traditional approach in real- meas time computer graphics has been to calculate lighting at a vertex as a sum of the ambient, diffuse, and specular light. In the simplest form (used by OpenGL and Direct3D), the function is simply the sum of these lighting components (clamped to a maximum color value). Thus we have an ambient term and then a sum of all the light from the light sources.

meas

$$i_{total} = k_a i_a + \sum (k_d i_d + k_s i_s))$$

Where i_{total}, is the intensity of light (as an **rgb** value) from the sum of the intensity of the global ambient value and the diffuse and specular components of the light from the light sources. This is called a *local lighting model* since the only light on a vertex is from a light source, not from other objects. That is, lights are lights, not objects. Objects that are brightly lit don't illuminate or shadow any other objects. We've included the *reflection coefficients* for each term, **k** for completeness since we'll frequently see the lighting equation. The reflection coefficients are in the [0, 1] range and are specified as part of the material property. However, they are strictly empirical and since they simply adjust the overall intensity of the material color, the material color values are usually adjusted so the color intensity varies rather than using a reflection coefficient, so we'll ignore them in our actual color calculations. This is a very simple lighting equation and gives fairly good results. However, it does fail to take into account any gross roughness or anything other than perfect isotropic reflection. That is, the surface is treated as being perfectly smooth and equally reflective in all directions. Thus this equation is really only good at modeling the illumination of objects that don't have any "interesting" surface properties. By this we mean anything other than a smooth surface (like fur or sand) or a surface that doesn't really reflect light uniformly in all directions (like brushed metal, hair, or skin). However, with liberal use of texture maps to add detail, this model has served pretty well and can still be used for a majority of the lighting processing to create a realistic environment in real time. Let's take a look at the individual parts of the traditional lighting pipeline.

Ambient Light

that comes from all directions—thus all surfaces are illuminated

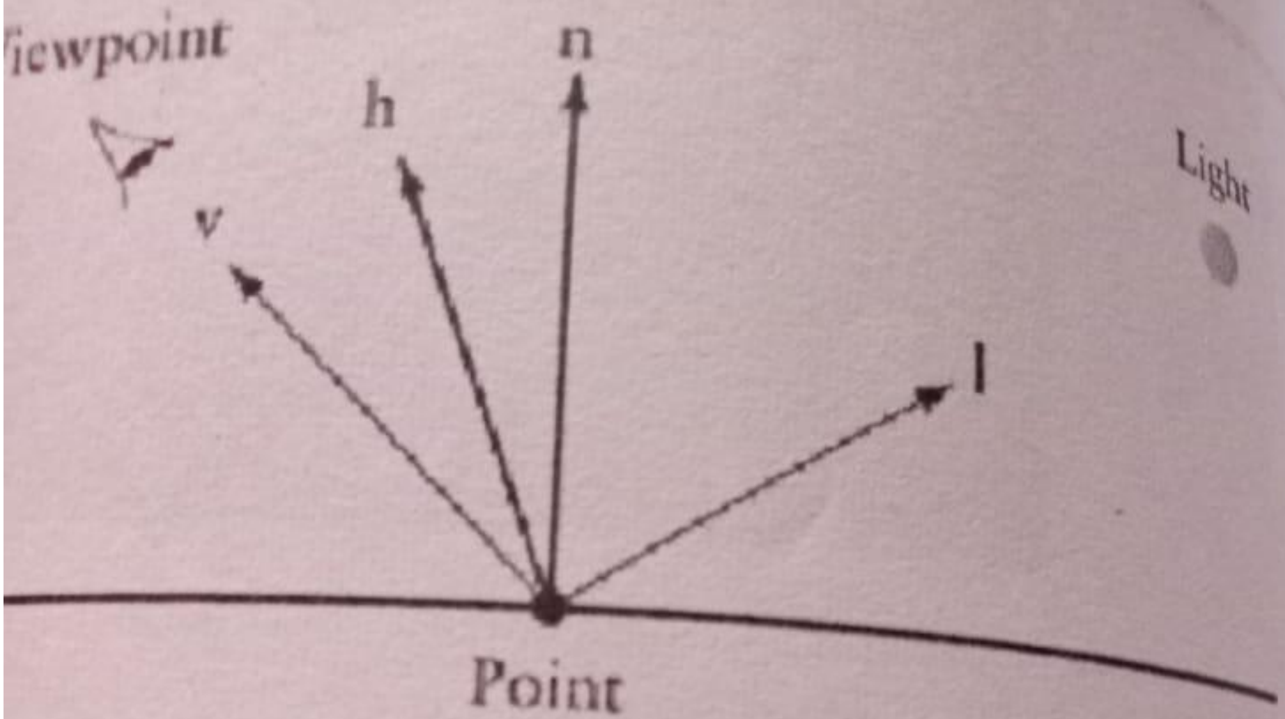
However, this is a big hack in traditional lighting

comes from the light reflected from th

would require ray tracing

of glob

ng)



: The half-angle vector is an averaging of the light and view vectors. where the half vector is defined as

MCQs

$$h = \frac{l+v}{|l+v|}$$

s that no reflection vector is needed; instead, we can use values that
, namely, the view and light vectors. Note that both OpenGL and
Blinn's equation for specular light. Besides a speed advantage, there
ts to note between Phong's specular equation and Blinn's. If we mul
by 4, we approximate the results of Phong's equation. Thus if there
the value of the exponent, Phong's equation can produce sh
• v angles greater than 45° (i.e., when the light is behind an objec
an edge), the highlights are longer along the edge direction for Ph
equation

28.9 Gouraud Shading

MCQs

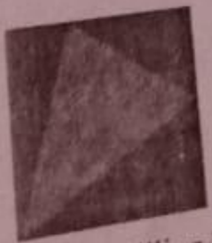
The idea of gouraud and flat triangle is nearly the same. Gouraud takes only three parameters more (the color value of each of the vertices), and the routine just interpolates among them drawing a beautiful, shaded triangle.
You can use 256-colors mode, in which vertices' colors are simply indices to palette or hi-color mode (recommended).

Flat triangle interpolated only one value (x in connection with y), 256 colors gouraud needs three (x related to y, color related to y, and color related to x), hi-color gouraud needs seven (x related to y, red, green and blue components of color related to y, and color related to x (also three components))

MCQs

Drawing a gouraud triangle, we add only two parts to the flat triangle routine. The horizline routine gets a bit more complicated due to the interpolation of the color value related to x but the main routine itself remains nearly the same.

28.10 Textured Triangles



We can also apply any bitmap on triangle for filling it.

I'll show you the idea of linear (or 'classical') texture mapping (without perspective correction). Linear mapping works pretty well (read: fast) in some scenes, but perspective correction is in some way needed in most 3D systems.

interpolation: now we'll code a texture triangle filler. And two more values to interpolate, that is five v related to y, and u and v tion is mayb

- With curved surfaces, the accuracy of the approximation is directly proportional to the number of polygons used in the representation.
- More polygons (when well used) yield a better approximation.
- But more polygons also exact greater computational overhead, thereby degrading interactive performance, increasing render times, etc.

27.27 *Rendering*

- The process of computing a two dimensional image using a combination of a three-dimensional database, scene characteristics, and viewing transformations. Various algorithms can be employed for rendering, depending on the needs of the application.

27.28 *Tessellation*

- The subdivision of an entity or surface into one or more non-overlapping primitives. Typically, renderers decompose surfaces into triangles as part of the renderin process.

27.29 *Sampling*

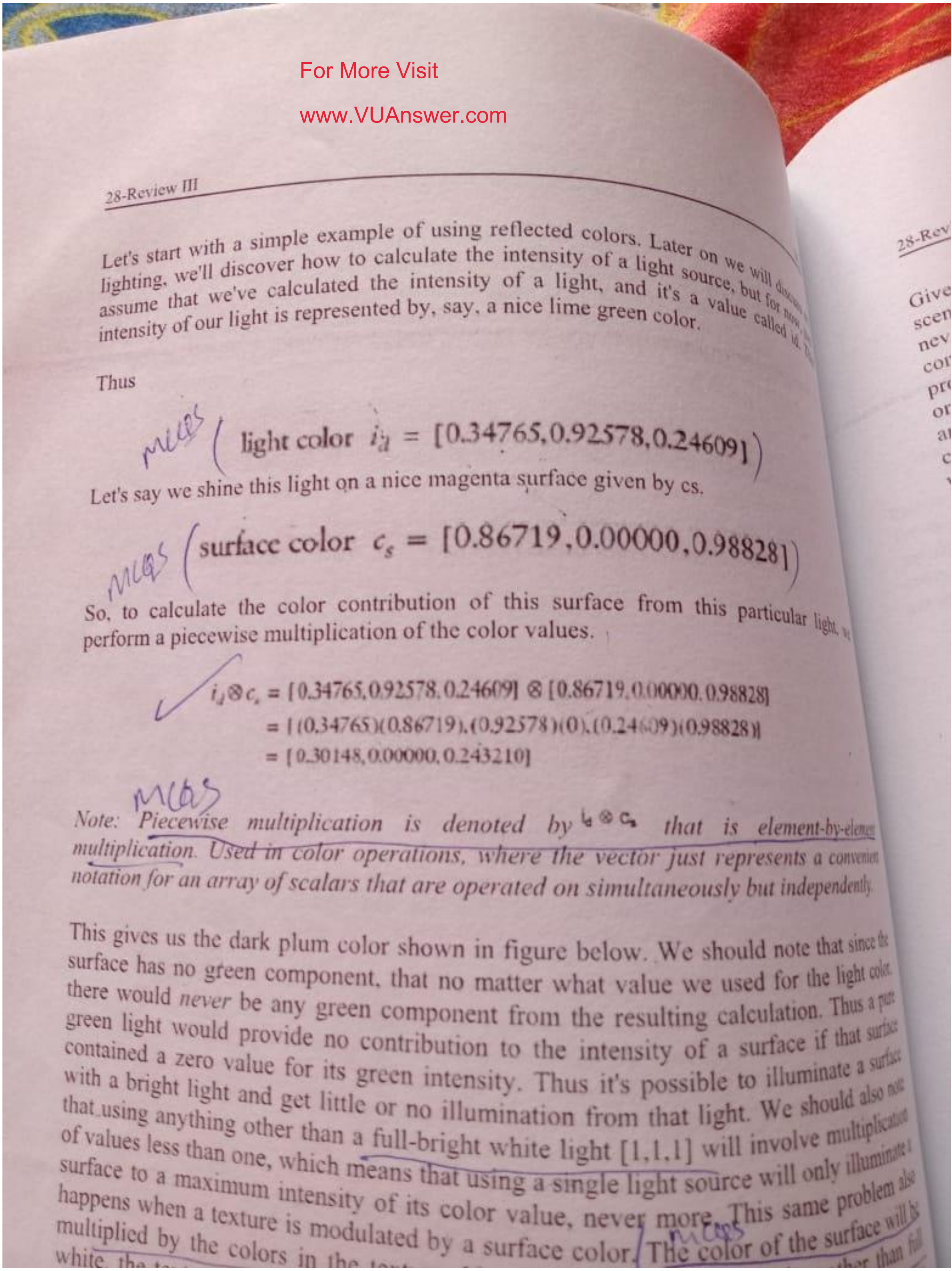
- The process of selecting a representative but finite number of values along continuous function sufficient to render a reasonable approximation of the funct for the task at hand.

27.30 *Level of Detail (LOD)*

- To improve rendering efficiency when dynamically viewing a scene, more or detailed versions of a model may be swapped in and out of the scene dat depending on the importance (usually determined by image size) of the object current view.

27.31 Transformations

- The process of moving points in space is called transformation.



Let's start with a simple example of using reflected colors. Later on we will discuss lighting, we'll discover how to calculate the intensity of a light source, but for now assume that we've calculated the intensity of a light, and it's a value called id. The intensity of our light is represented by, say, a nice lime green color.

Thus

MCQs

$$\text{light color } i_d = [0.34765, 0.92578, 0.24609]$$

Let's say we shine this light on a nice magenta surface given by cs.

MCQs

$$\text{surface color } c_s = [0.86719, 0.00000, 0.98828]$$

So, to calculate the color contribution of this surface from this particular light, we perform a piecewise multiplication of the color values.

$$\begin{aligned} i_d \otimes c_s &= [0.34765, 0.92578, 0.24609] \otimes [0.86719, 0.00000, 0.98828] \\ &= [(0.34765)(0.86719), (0.92578)(0), (0.24609)(0.98828)] \\ &= [0.30148, 0.00000, 0.243210] \end{aligned}$$

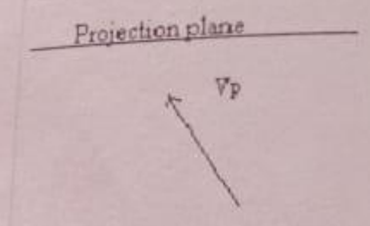
MCQs

Note: Piecewise multiplication is denoted by $i_d \otimes c_s$ *that is element-by-element multiplication. Used in color operations, where the vector just represents a convenient notation for an array of scalars that are operated on simultaneously but independently.*

This gives us the dark plum color shown in figure below. We should note that since the surface has no green component, that no matter what value we used for the light color, there would *never* be any green component from the resulting calculation. Thus a pure green light would provide no contribution to the intensity of a surface if that surface contained a zero value for its green intensity. Thus it's possible to illuminate a surface with a bright light and get little or no illumination from that light. We should also note that using anything other than a full-bright white light [1,1,1] will involve multiplication of values less than one, which means that using a single light source will only illuminate a surface to a maximum intensity of its color value, never more. This same problem also happens when a texture is modulated by a surface color. MCQs The color of the surface will be multiplied by the colors in the te ther than full white, tha

Orthographic projection

- **Oblique**



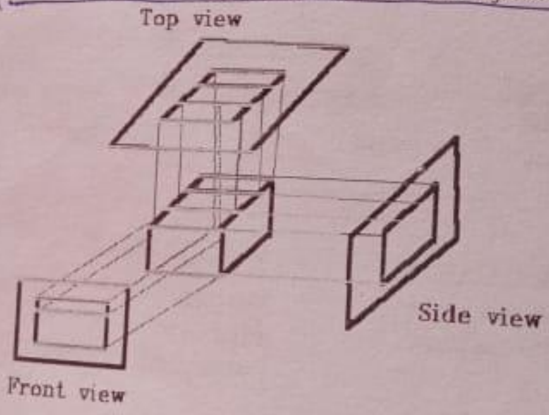
Oblique Projection

Orthographic projection has two types

MCQs
- Multiview
- Axonometric

Multiview:

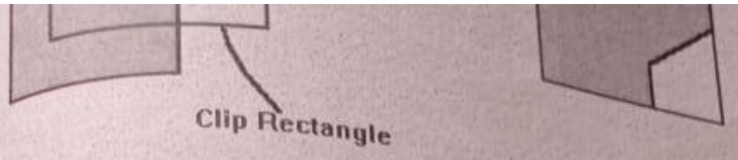
MCQ There are three orthographic views of an object.



Axonometric projections:

There are three axonometric projections:

MCQs
- Isometric
- Dimetric
- Trimetric



27.7 *Sutherland and Hodgman's polygon-clipping algorithm:-*

Sutherland and Hodgman's polygon-clipping algorithm uses a divide-and-conquer strategy: It solves a series of simple and identical problems that, when combined, solve the overall problem. The simple problem is to clip a polygon against a single infinite clip edge. Four clip edges, each defining one boundary of the clip rectangle, successively clip a polygon against a clip rectangle.

Note the difference between this strategy for a polygon and the Cohen-Sutherland algorithm for clipping a line: The polygon clipper clips against four edges in succession, whereas the line clipper tests the outcode to see which edge is crossed, and clips only when necessary.

27.8 *Steps of Sutherland-Hodgman's polygon-clipping algorithm*

MCQS

- Polygons can be clipped against each edge of the window one at a time. Windows/edge intersections, if any, are easy to find since the X or Y coordinates are already known.
- Vertices which are kept after clipping against one window edge are saved for clipping against the remaining edges.
- Note that the number of vertices usually changes and will often increase.

We are using the Divide and Conquer approach.

27.9 *Shortcoming of Sutherlands -Hodgeman Algorithm*

Convex polygons are correctly clipped by the Sutherland-Hodegeman algorithm, but concave polygons may be displayed with extraneous lines. This occurs when the clipped polygon should have two or more separate sections. But since there is only one output vertex list, the last vertex in the list is always joined to the first vertex. There are several things we could do to correct display concave polygons. For one, we could split the concave polygon into two or more convex polygons and process each convex polygon separately.

27.32 Types of Transformation

There are various types of transformations as we have seen in case of 2D transformations. These include:

MCQs
- a. Translation
- b. Rotation
- c. Scaling
- d. Reflection
- e. Shearing

(a) Translation

MCQs

Translation is used to move a point, or a set of points, linearly in space. Since now we are talking about 3D, therefore each point has 3 coordinates i.e. x, y and z. similarly, the translation distances can also be specified in any of the 3 dimensions. These Translation Distances are given by tx, ty and tz.

For any point P(x,y,z) after translation we have P′(x′,y′,z′) where

$$x' = x + tx\,,$$
$$y' = y + ty\,,$$
$$z' = z + tz$$

and (tx, ty, tz) is Translation vector

Now this can be expressed as a single matrix equation:

$$\boxed{P' = P + T}$$

Where:

$$P = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \qquad P' = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} \qquad T = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

3D Translation Example

We may want to move a point "3 meters east, -2 meters up, and 4 meters north." What would be done in such event?

Steps for Translation

Given a point in 3D and a translation vector, it can be translated as follows:

```
        Point3D point = (0, 0, 0)
        Vector3D vector = (10, -3, 2.5)
Adding vector to point
        point.x = point.x + vector.x;
        point.y = point.y + vector.y;
        point.z = point.z + vector.z;
And finally we have translated point.
```

Homogeneous Coordinates

Analogous to their 2D

the ... diffuse light ... the same. Thus, unlike ... and is a function of the angle of ... type of shading is called *Lambertian shading* after Lambert... which states that the intensity of the light reflected from an ideal diffuse surface is proportional to the cosine of the direction of the light to the vertex normal. Since we're dealing with vertices here and not surfaces, each vertex has a normal associated with it. We might hear talk of per-vertex normals vs. per-polygon normals. The difference being that per polygon has one normal shared for all vertices in a polygon, whereas per vertex has a normal for each vertex. MCQs OpenGL has the ability to specify per-polygon normals, and Direct3D does not. Since vertex shaders can't share information between vertices (unless we explicitly copy the data our self). We'll focus on per-vertex lighting. Figure 2 shows the intensity of reflected light as a function of the angle between the vertex normal and the light direction.

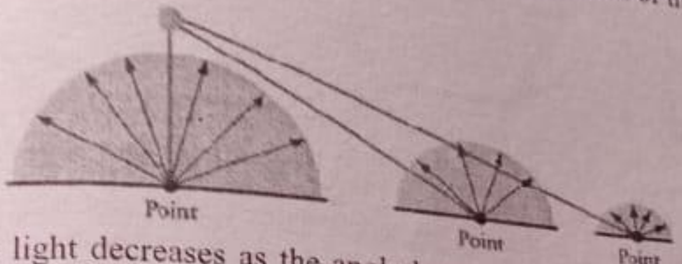


Figure 2: Diffuse light decreases as the angle between the light vector and the surface normal increases.

The equation for calculating diffuse lighting is

MCQs
$$i_d = (\hat{n} \bullet \hat{l})(m_d \otimes s_d)$$

Which is similar to the ambient light equation, except that the diffuse light term is now multiplied by the dot product of the unit normal of the vertex and the unit direction vector to the light from the vertex (not the direction *from* the light). Note that the **md** value is a color vector, so there are **rgb** or **rgba** values that will get modulated.

Since $(\hat{n} \bullet \hat{l}) = |\hat{n}||\hat{l}| \cos(\theta)$, where theta is the angle between vectors, when the angle between them is zero, cos(theta) is 1 and the diffuse light is at its maximum. When the angle is 90°, cos (theta) is zero and the diffuse light is zero. One calculation advantage is that when the cos(theta) value is negative, this means that the light isn't illuminating the vertex at all. However, since we (probably!) don't want the light illuminating sides that it physically can't shine on, we want to clamp the contribution of the diffuse light to contribute only when cos(theta) is positive. Thus the equation in practice looks more like MCQs

MCQs
$$i_d = \text{MAX}(0, (\hat{n} \bullet \hat{l})(m_d \otimes s_d))$$

Where we've clamped the diffuse value to only positive values, Figure 3 was rendered with just diffuse lighting. Notice how we can tell a lot more detail about the objects and pick up distance cues from the shading.

*Where **ia** is the ambient light intensity, **ma** is the ambient material color, and* *light source ambient color*. Typically, the ambient light is some amount of w equal **rgb** values) light, but we can achieve some nice effects using colored am Though it's very useful in a scene, ambient light doesn't help differentiate obj scene since objects rendered with the same value of ambient tend to blend resulting color is the same. Figure 1 shows a scene with just ambient illumination can see that it's difficult to make out details or depth information with just ambient l

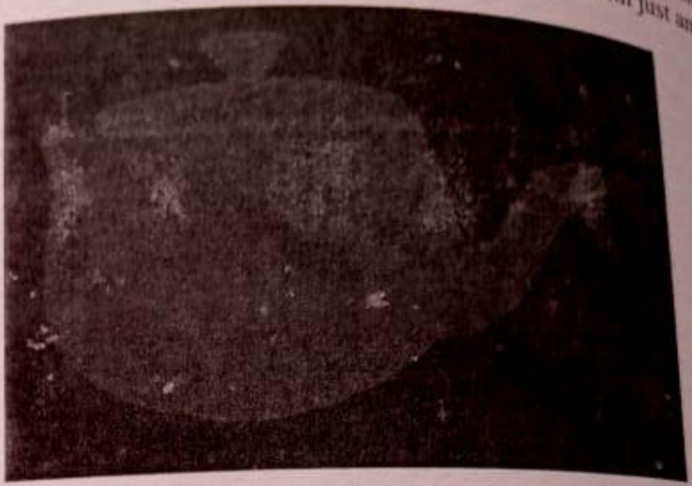
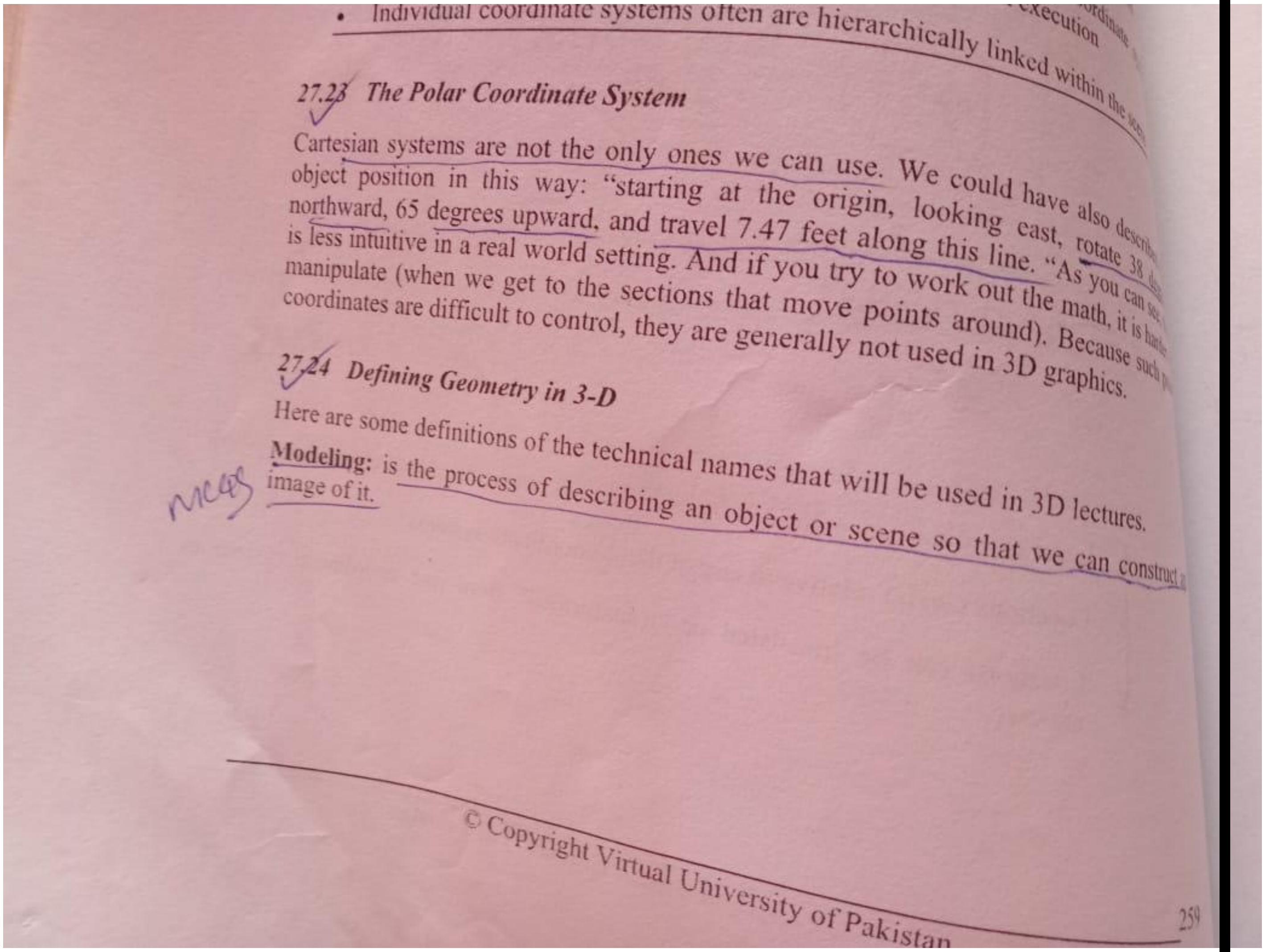


Figure 1: Ambient light provides illumination, but no surface details.

mcqs Ambient lighting is our friend. With it we make our scene seem more realistic than it A world without ambient light is one filled with sharp edges, of bright objects surrounde by sharp, dark, harsh shadows. A world with too much ambient light looks washed ou and dull. Since the number of actual light sources supported by hardware FFP is limited (typically to eight simultaneous), we'll be better off to apply the lights to add detail to the area that our user is focused on and let ambient light fill in the rest. Before we point ou that talking about the hardware limitation of the number of lights has no meaning in shaders, where *we* do the lighting calculations, we'll point out that eight lights were typically the maximum that the hardware engineers created for *their* hardware. It was a performance consideration. There's nothing stopping us (except buffer size) from writing a shader that calculates the effects from a hundred simultaneous lights. But we think tha we'll find that it runs much too slowly to be used to render our entire scene. But the ni thing about shaders is we *can*.

Diffuse Light

mcqs Diffuse light is the light that is absorbed by a surface and is reflected in all directio the traditional model, this is *ideal* diffuse reflection—good for rough surfaces whe reflected intensity is constant across the surface and is independent of viewpo



ordinate
execution

- Individual coordinate systems often are hierarchically linked within the scen

27.23 The Polar Coordinate System

Cartesian systems are not the only ones we can use. We could have also describ object position in this way: "starting at the origin, looking east, rotate 38 de northward, 65 degrees upward, and travel 7.47 feet along this line. "As you can se is less intuitive in a real world setting. And if you try to work out the math, it is harde manipulate (when we get to the sections that move points around). Because such p coordinates are difficult to control, they are generally not used in 3D graphics.

27.24 Defining Geometry in 3-D

Here are some definitions of the technical names that will be used in 3D lectures.

MCQs

Modeling: is the process of describing an object or scene so that we can construct a image of it.

object an illuminated matte surface, specular light is what gives the highlights to an object. These highlights are greatest when the viewer is looking directly along the reflection angle from the surface. This is illustrated in Figure 5.

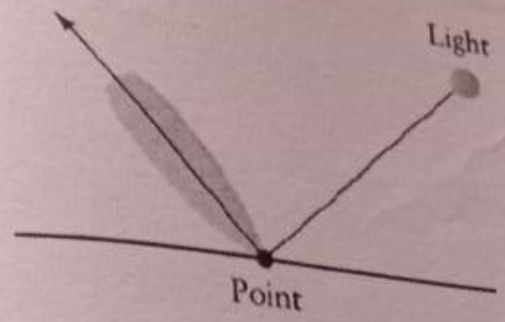


Figure 5: Specular light's intensity follows the reflection vector.

Most discussions of lighting (including this one) start with Phong's lighting equation (which is not the same as Phong's shading equation). In order to start discussing specular lighting, let's look at a diagram of the various vectors that are used in a lighting equation. We have a light source, some point the light is shining on, and a viewpoint. The light direction (from the point *to* the light) is vector l, the reflection vector of the light vector (as if the surface were a mirror) is **r**, the direction *to* the viewpoint from the point is vector **v**. The point's normal is **n**.

Phong's Specular Light Equation

MCQ

Warnock [WARNOCK 1969] and Romney [ROMNEY 1969] were the first to try to simulate highlights using a cos n (θ) term. But it wasn't until Phong Bui-Tong [BUI 1998] reformulated this into a more general model that formalized the power value as a measure of surface roughness that we approach the terms used today for specular highlights. Phong's equation for specular lighting is

MCQS

$$i_s = (m_s \otimes s_s)(\hat{r} \bullet \hat{v})^{m_s} \qquad \text{(Phong)}$$

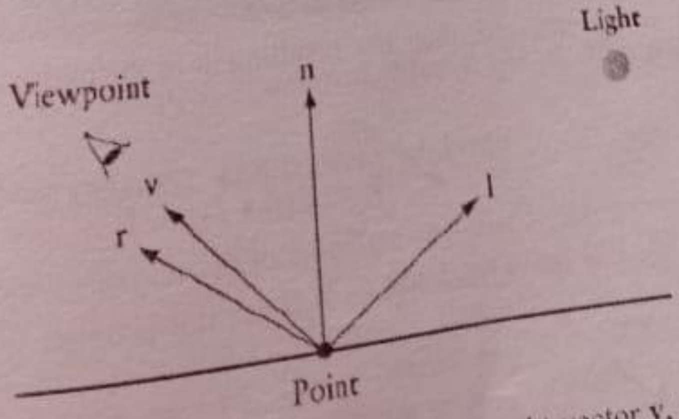


Figure 6: The relationship between the normal **n**, the light vector **v**, the view direction,

It's ... normal <a,b,c> does not have to be unit-length for it to ... on. But since things end up nicer if the normal is unit-length.

Constructing a plane given three points that lie in the plane is a simple task. You just perform a cross product between the two vectors made up by the three points

<point₂ − point₀>

<point₁ − point₀>

and find a normal for the plane. After generating the normal and making it unit length, finding the d value for the plane is just a matter of storing the negative dot product of the normal with any of the points. This holds because it essentially solves the plane equation above for d. Of course plugging a point in the plane equation will make it equal 0, and this constructor has three of them.

28.5 Back-face Culling

Now that you know how to define a point with respect to a plane, you can perform back-face culling, one of the most fundamental optimization techniques of 3D graphics.

Let's suppose you have a triangle whose elements are ordered in such a fashion that when viewing the triangle from the front, the elements appear in clockwise order. Back-face culling allows you to take triangles defined with this method and use the plane equation to discard triangles that are facing away. Conceptually, any closed mesh, a cube for example, will have some triangles facing you and some facing away. You know for a fact that you'll never be able to see a polygon that faces away from you; they are always hidden by triangles facing towards you. This, of course, doesn't hold if you're allowed to view the cube from its inside, but this shouldn't be allowed to happen if you want to really optimize your engine.

Rather than perform the work necessary to draw all of the triangles on the screen, you can use the plane equation to find out if a triangle is facing towards the camera, and discard it if it is not. How is this achieved? Given the three points of the triangle, you can define a plane that the triangle sits in. Since you know the elements of the triangle are listed in clockwise order, you also know that if you pass the elements in order to the plane constructor, the normal to the plane will be on the front side of the triangle. If you then think of the location of the camera as a point, all you need to do is perform a point-plane test. If the point of the camera is in front of the plane, then the triangle is visible and should be drawn. MCQs

There's an optimization to be had. Since you know three points that lie in the plane (the three points of the triangle) you only need to hold onto the normal of the plane, not the entire plane equation. To perform the back-face cull, just subtract one of the triangle's points from the camera location and perform a dot product with the resultant vector and the normal. If the result of the dot product is greater than zero, then the view point was in front of the triangle. Figure below can help explain the point.

MCQs

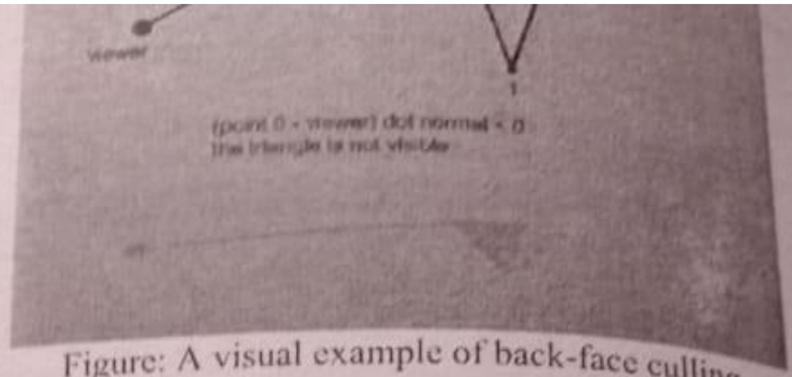

Viewer
1
(point 0 - viewer) dot normal < 0
the triangle is not visible

Figure: A visual example of back-face culling

In practice, 3D accelerators can actually perform back-face culling by themsel the triangle rates of cards increase, the amount of manual back-face culli performed has steadily decreased. However, the information is useful for engines that don't plan on using the facilities of direct hardware acceleration.

28.6 *Intersection between a Line and a Plane*

This occurs at the point which satisfies both the line and the plane equations.

MCQs

Line equation: p = org + u * dir **(1)**

Plane equation: p * normal - k = 0. **(2)**

Substituting (1) into (2) and rearranging we get:

(org + u * dir) * normal - k = 0

ie u * dir * normal = k - org * normal

ie u = (k - org * normal) / (dir * normal)

MCQ If (d * normal) = 0 then the line runs parrallel to the plane and no intersecti n o exact point at which intersection does occur can be found by plugging u ba k i equation in (1).

28.7 *Triangle Rasterization*

High performance triangle rasterization is a very important topic in Computer today's world.

Triangles are the foundation of modern real time graphics, and are by far the rendering primitive. Most computer games released in the last few year completely dependent on triangle rasterization performance. Recentl graphics performance optimization is beginning to shift to bandwidth req well as transformation and lighting. Nevertheless, rasterization performa factor, and this lecture will provide most of the basics of high perform rasterization. Also, it will go into detail about two often neglected rend improvements, sub-pixel and sub-texel accuracy. Also, smooth shadin mapping techniques will be described.

Abbreviated as:

$$P' = T\,(tx, ty, tz)\ .\ P$$

On solving the RHS of the matrix equation, we get:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} x+t_x \\ y+t_y \\ z+t_z \\ 1 \end{bmatrix}$$

Which shows that each of the 3 coordinates gets translated by the corresponding translation distance.

Rotation

Rotation is the process of moving a point in space in a non-linear manner

- We need to know three different angles:
- How far to rotate around the X axis(YZ rotation, or "pitch")
- How far to rotate around the Y axis (XZ rotation, or "yaw")
- How far to rotate around the Z axis (XY rotation, or "roll")

Column vector representation:

$$\boxed{P' = R\ .\ P}$$

Where

$$P' = \begin{bmatrix} x' \\ y' \end{bmatrix} \qquad R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \qquad P = \begin{bmatrix} x \\ y \end{bmatrix}$$

Rotation: Homogeneous Coordinates

The rotation can now be expressed using homogeneous coordinates as:

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

Abbreviated as:

$$P' = R\,(\theta)\ .\ P$$

... Now in 3D

- Rotation can be about any of the three axes:
- About z-axis (i.e. in xy plane)
- About x-axis (i.e. in yz plane)
- About y-axis (i.e. in xz plane)

Roll : around z-axis
Pitch: around x-axis
Yaw: around y-axis

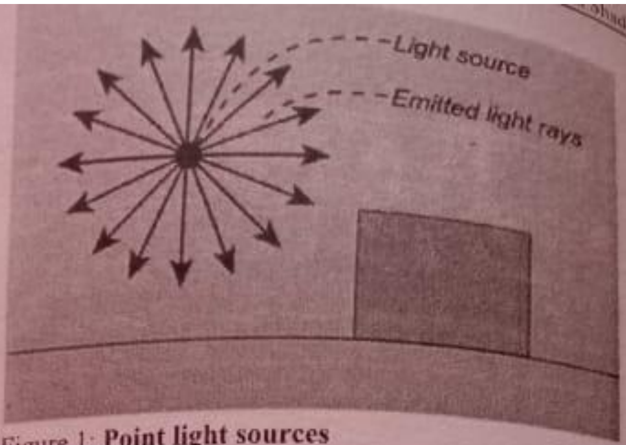


Figure 1: **Point light sources**

III. Spotlights

Spotlights are the most expensive type of light we discuss in this course and should be avoided if possible because it is not for real time environment. We model a spotlight unlike the type we would see in a theatrical production. They are point lights, but light only leaves the point in a particular direction, spreading out based on the aperture of the light.

MCQs Spotlights have two angles associated with them. One is the internal cone whose angle is generally referred to as theta (θ). Points within the internal cone receive all of the light of the spotlight; the attenuation is the same as it would be if point lights were used. There is also an angle that defines the outer cone; the angle is referred to as phi. Points outside the outer cone receive no light. Points outside the inner cone but inside the outer cone receive light, usually a linear falloff based on how close it is to the inner cone.

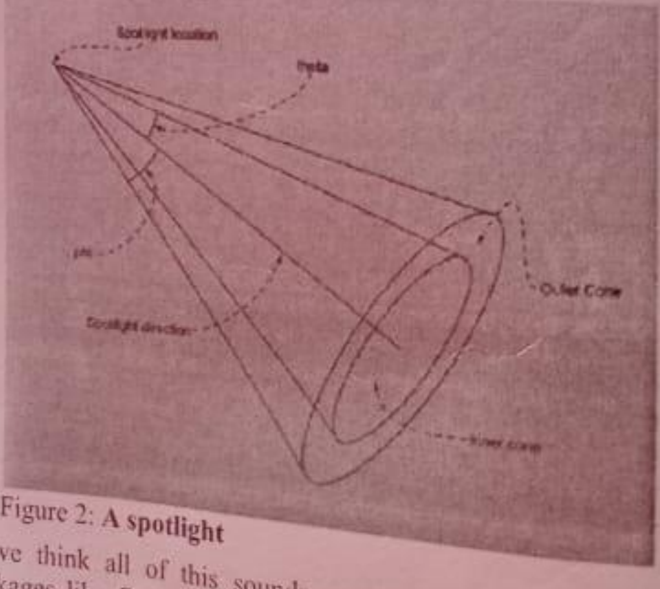


Figure 2: **A spotlight**

If we think all of this sounds mathematically expensive, we're right. Some library packages like OpenGL and Direct3D implements lighting for us, so we won't need to worry about the implementation of the math behind spotlights, but rest assured that they're extremely expensive and can slow down our graphics application a great deal.

Then again, they do provide a incre
we will have to figure out a li e betw

Shading Models

Once we've found basic li hting
with the supplied informa ion. T
become a hardware feat re wi
studied flat and gouraud hadin

I. Lambert

Triangles that use La berti
gradient. Typically e ch tr
looks very angular a d sha
weren't fast enough o do
the lighting equati ns u
triangle.

Figure 3: lat s

II. G ura

Gouraud (pr
accelera ed 3
each ve tex
the tria gle.
lightin for

Of c urse
inste d is
the erte
(rat er t
Thi cre

Lecture No.26 Mathematics of Lighting and Shading Part II Light Types and Shading Models

Light Types

Now that we have a way to find the light hitting a surface, we're going to need some lights! There are three types of lights we are going to discuss.

I. Parallel Lights (or Directional Lights) MCQs

MCQ Parallel lights cheat a little bit. They represent light that comes from an infinitely far away light source. Because of this, all of the light rays that reach the object are parallel (hence the name). The standard use of parallel lights is to simulate the sun. While it's not infinitely far away, 93 million miles is good enough!

The great thing about parallel lights is that a lot of the math goes away. The attenuation factor is always 1 (for point/spotlights, it generally involves divisions if not square roots). The incoming light vector for calculation of the diffuse reflection factor is the same for all considered points, whereas point lights and spotlights involve vector subtractions and a normalization per vertex.

MCQs

Typically, lighting is the kind of effect that is sacrificed for processing speed. Parallel light sources are the easiest and therefore fastest to process. If we can't afford to do the nicer point lights or spotlights, falling back to parallel lights can keep our frame rates at reasonable levels.

II. Point Lights

MCQ Point lights are one step better than directional lights. They represent infinitesimally small points that emit light. Light scatters out equally in all directions. Depending on how much effort we're willing to expend on the light, we can have the intensity falloff based on the inverse squared distance from the light, which is how real lights work.

$$\text{attenuation\_factor} = \frac{k}{|\text{surface\_location} - \text{light\_location}|^2}$$

The light direction is different for each surface location (otherwise the point light would look just like a directional light). The equation for it is:

$$\text{light\_direction} = \frac{\text{surface\_location} - \text{light\_location}}{|\text{surface\_location} - \text{light\_location}|}$$

Then again, they do provide an incredible amount of atmosphere when used correctly, so we will have to figure out a line between performance and aesthetics.

Shading Models

Once we've found basic lighting information, we need to know how to draw the triangles with the supplied information. There are currently three ways to do this; the third has just become a hardware feature with DirectX 9.0 In our previous lectures we have already studied flat and gouraud shading triangle algorithms.

I. Lambert

Triangles that use Lambertian shading are painted with one solid color instead of using a gradient. Typically each triangle is lit using that triangle's normal. The resulting object looks very angular and sharp. Lambertian shading was used mostly back when computers weren't fast enough to do Gouraud shading in real time. To light a triangle, you compute the lighting equations using the triangle's normal and any of the three vertices of the triangle.

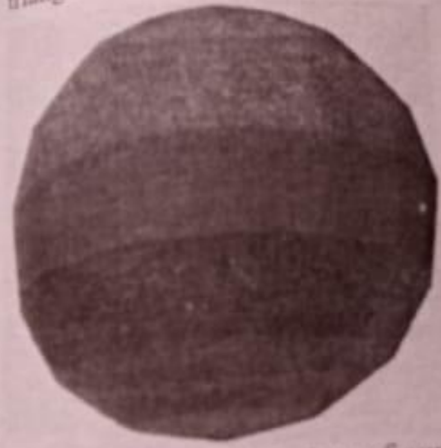


Figure 3: Flat shaded view of our polygon mesh

II. Gouraud

Gouraud (pronounced *garrow*) shading is the current de facto shading standard in accelerated 3D hardware. Instead of specifying one color to use for the entire triangle, each vertex has its own separate color. The color values are linearly interpolated across the triangle, creating a smooth transition between the vertex color values. To calculate the lighting for a vertex, we use the position of the vertex and a vertex normal.

Of course, it's a little hard to correctly define a normal for a vertex. What people do instead is average the normals of all the polygons that share a certain vertex, using that as the vertex normal. When the object is drawn, the lighting color is found for each vertex (rather than each polygon), and then the colors are linearly interpolated across the object. This creates a slick and smooth look, like the one in Figure 4.

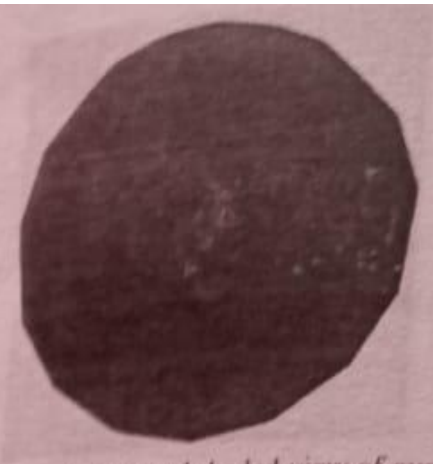


Figure 4: Gouraud shaded view of our polygon mesh

Quiz MCQ One problem with Gouraud shading is that the triangles' intensities can never be than the intensities at the edges. So if there is a spotlight shining directly into the a large triangle, Gouraud shading will interpolate the intensities at the three dark resulting in an incorrectly dark triangle. The internal highlighting problem that bad. If there are enough triangles in the model, the interpolation done by G shading is usually good enough. If we really want internal highlights but only Gouraud shading, we can subdivide the triangle into smaller pieces.

III. Phong

MCQ Phong shading is the most realistic shading model. We are going to talk about, and the most computationally expensive. It tries to solve several problems that arise when use Gouraud shading. If we're looking for something more realistic, some authors also discussed nicer shading models like Tarrence-Sparrow, but they aren't real time least not right now). First of all, Gouraud shading uses a linear gradient. Many objects real life have sharp highlights, such as the shiny spot on an apple. This is difficult handle with pure Gouraud shading. The way Phong does this is by interpolating the normal across the triangle face, not the color value, and the lighting equation is solved individually for each pixel.

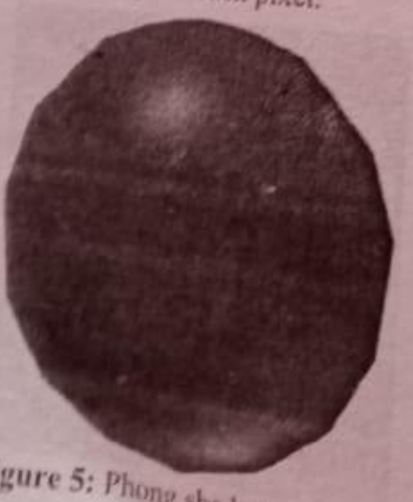


Figure 5: Phong shaded view of a polygon mesh

Phong shading isn't technically supported in hardware. But we can now program our own Phong rendering engine, and many other technology.

27.1 CLIPPING C

It is desirable to restr protect other portion clipping rectangle.

The default clippi cannot see any gr

A simple exampl

This is a simple default clipping

The red box i of the four ed