Graphics Pipeline and Coordinate Systems

Computer Graphics

- We learn/explore the environment around us largely by seeing or viewing:

with eyes,

with cameras.

- But biological vision/cameras have limitations

We cannot see items that do not possess a physical form, e.g., something that no longer exists or a process that is invisible (e.g., air flow).

Cameras cannot capture our imagination as do the artists.

- Many applications require images to be created under afore-mentioned circumstances.
- Computer technologies make this possible by imitating the biological viewing/optical imaging process.

Instead of involving physical entities, "objects" being "viewed" are created and represented in form of models.

Instead of using optical lens, geometric projections are used to simulate the behaviour of light rays when they pass through lenses/eyes – the viewing and rendering.

- The history of CG research has been mainly about finding out accurate & efficient methods for creating (synthesising) realistic, artistic or other forms of images.

To Create Images

- We need to:

    1. Represent (describe) shapes in some forms, i.e., object models.
    2. Describe a view angle/point in terms of the position and orientation of the camera towards the objects – a view platform.
    3. Simulate the optics of cameras by a process called projection

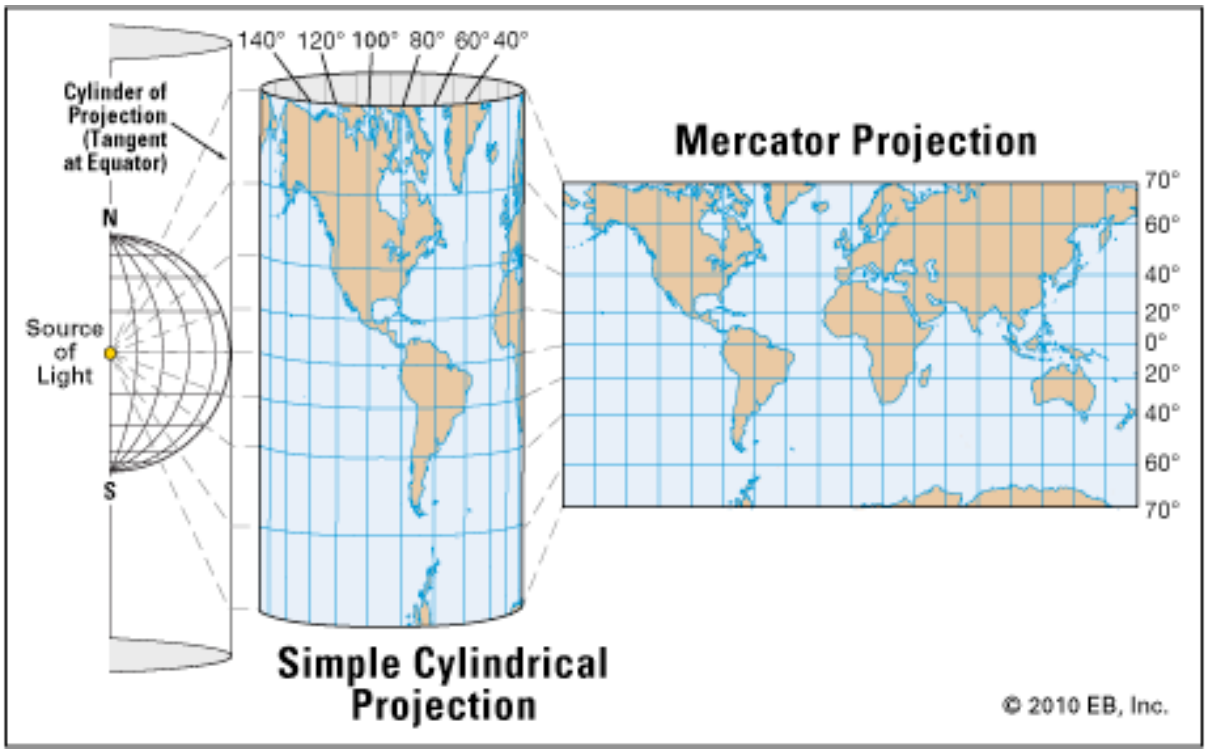
emitting light rays from a particular point (centre of projection)

tracing the paths of the light rays and determining where on the model the rays hit and where on the display screen pixels appear.

Projections

- Different forms of projections were developed in the study of geometry, painting and map-making.
- Perspective projection (eyes, cameras)
- Orthographic (or parallel) projection (engineering drawing)
- Others, e.g., cylindrical and conic projections (used in map making), oblique projection, and etc.

Map Projections



Orthographic Projection

- Orthographic projection create an image by using light rays that are

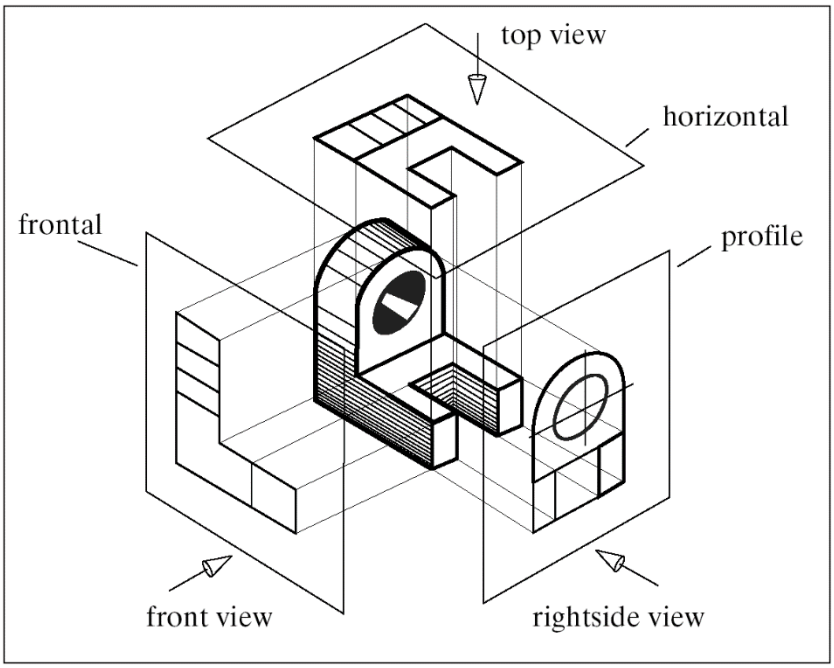
parallel to each other

perpendicular to the viewing/image plane (oblique rays are used sometimes, but such a projection is not useful/convenient for creating images in CG)

- The projection preserve object's sizes, e.g., the height of the image of the candle is the height of the real candle.
- A complete projection usually consists of three views:

Top view, side view, front view

See interface of 3D Studio Max

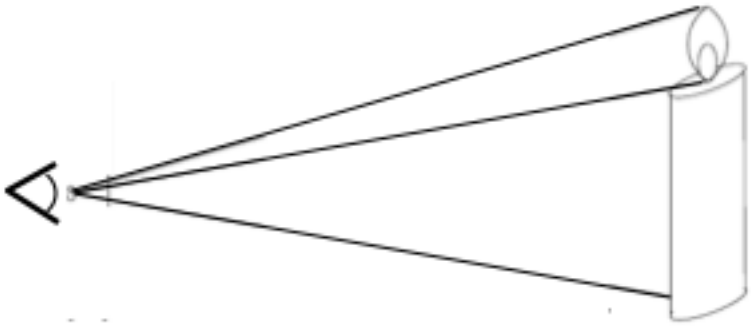


Perspective Projection

- Perspective projection

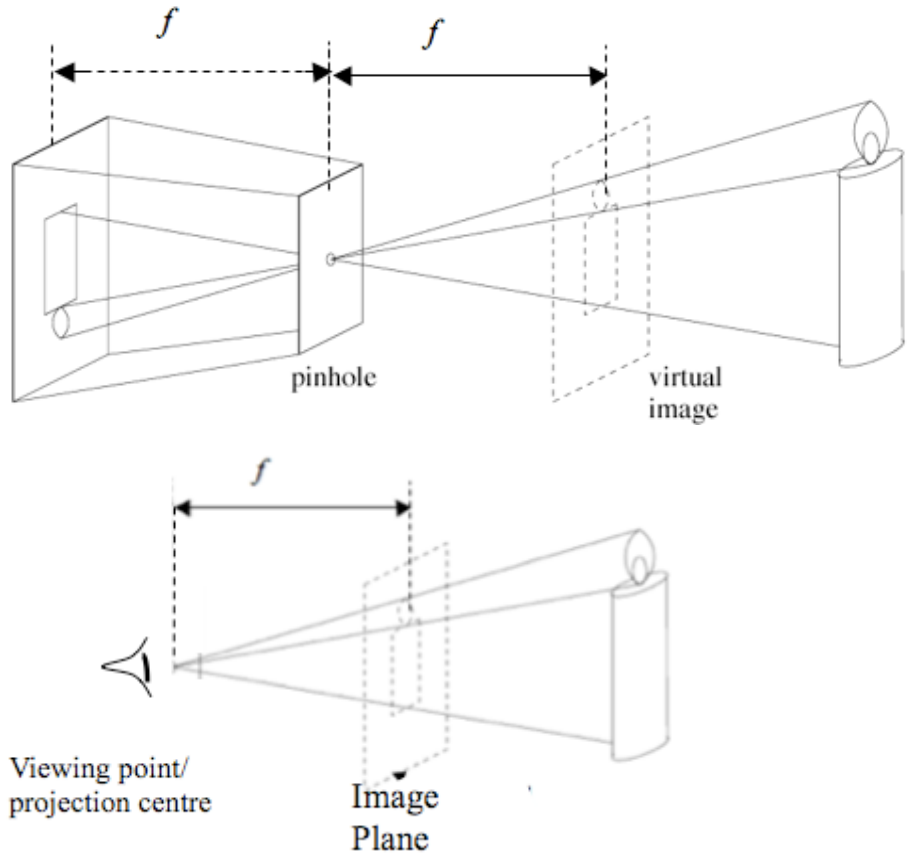
All light rays pass through a particular point – the *centre of projection*.

Perspective projection approximates the imaging processes of the lens in cameras and human eyes.



Virtual Camera Model

- In CG, for simplicity the image plane (film) on the other side of the projection centre and drop the concept of the box.
- This leads to the *virtual camera model* (figure on the right), which defines the *projection matrix* (the relationships among the geometric quantities of the model).



Perspective Distortions

- Perspective projection produces distortions

An image produced by a real camera. Where are the distortions?

- Sizes are not preserved (distant objects appear smaller).

- Parallelism are not normally preserved (they eventually intersect).

Projections in CG

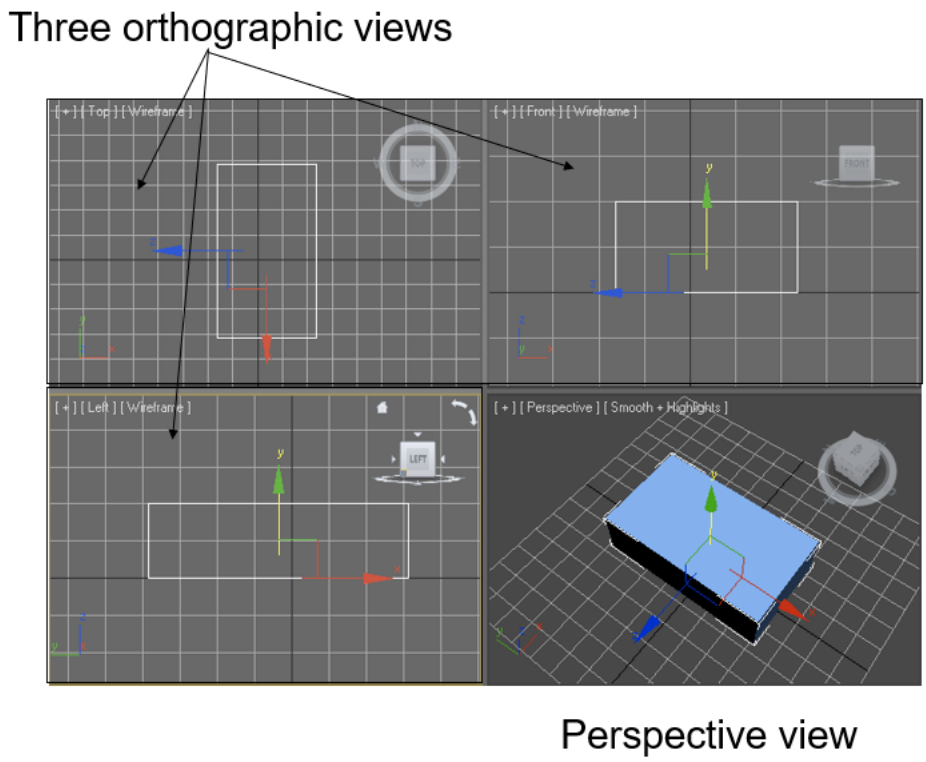
- In CG, perspective and orthographic projections are used:

Perspective: used to create images. It is the native way to create visually correct views

Orthographic:

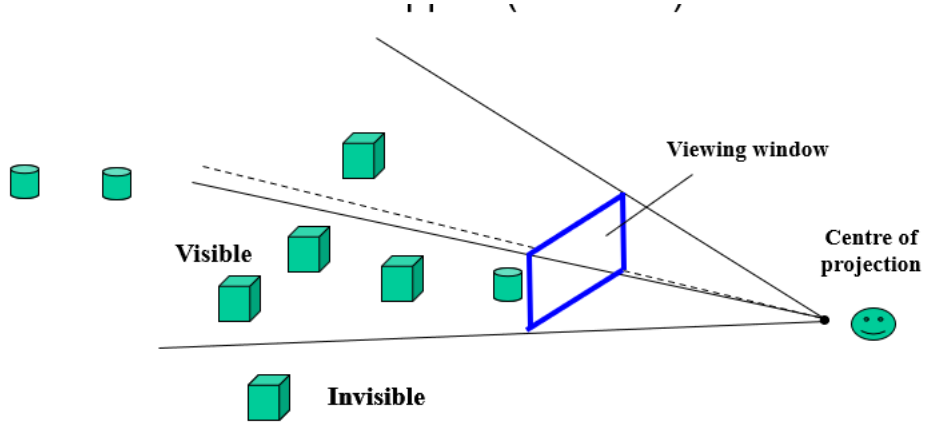
As a design aid to help to perceive scene/objects correctly

As an approximation of perspective projection to simplify computation



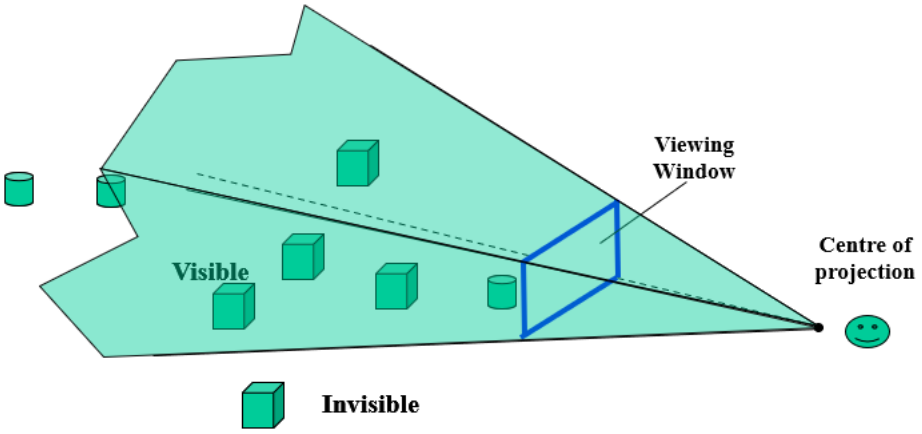
Viewing Window

- Like a real camera, the image plane of the virtual camera has a size. It has a different name, called viewing window or clipping window.
- The viewing window is a window against which a scene is clipped so that the objects or features outside the window are clipped (removed).



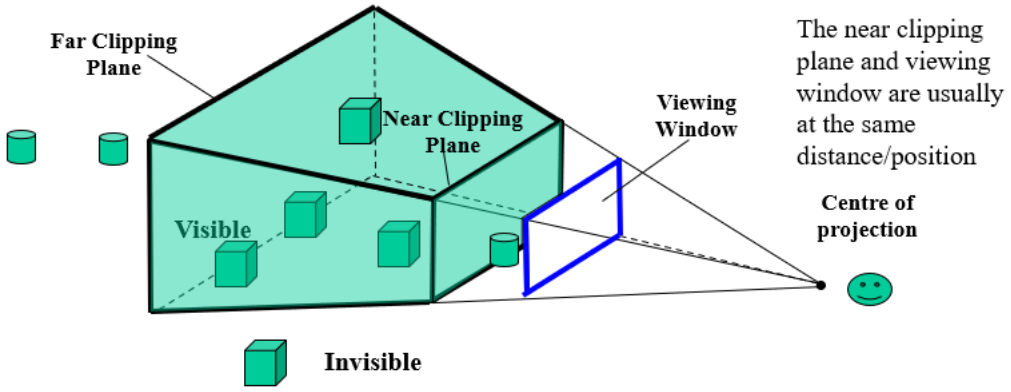
Viewing Volume

- The pyramid formed by the four bounding planes are called *view volume*, or *pyramid of vision*.



Viewing Frustum

- The scope of viewing can be further reduced by introducing *near* and *far clipping planes* – to remove the objects that are too far from or too close to the view plane
- View frustum: view volume formed by adding near and far clipping planes to the pyramid of vision



Shapes, Sizes, & Positions

- To create a view (image) of scene, we have to decide whether or not an object is inside the view frustum

To answer this question, we need to know the relative positions of the object and the camera (view frustum).

- If yes, then we project it onto the virtual film. For this, we need to know:

What are its shape and size?

Where is the exact location the object with respect to the camera?

- To facilitate all these calculations, we need to define a few coordinate systems.

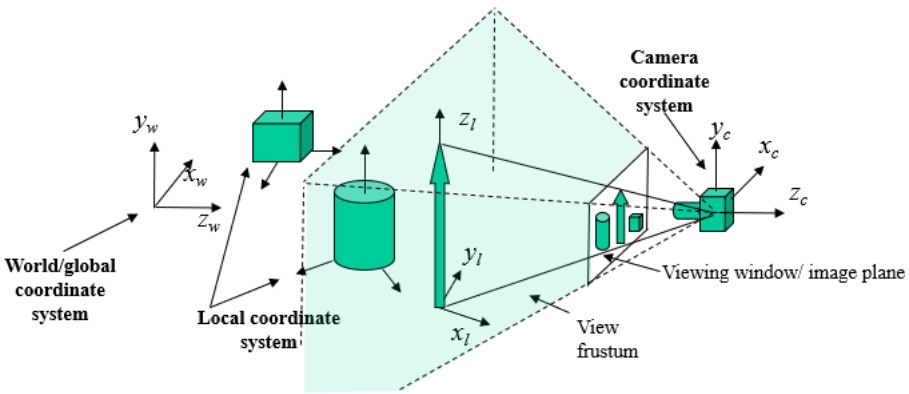
Coordinate Systems

- Spaces and coordinate systems

One “world coordinate system” in which to define positions/orientations of all items: camera, lights, objects,…

One or more “local coordinate system” in which coordinates of objects are defined

One “viewing” or “camera” coordinate system in which viewing frustum and image/view plane are defined



World Coordinates

- All the objects in the world have particular locations. These locations are specified in terms of a unique coordinate system - the *world* or *global coordinate system*

Local Coordinates

- A *local coordinate system* is the coordinate system that is attached to an object. Each object (shapes, lights, camera, etc) will have its own local coordinate system.
- It is easiest to define an individual object in a local coordinate system.

For instance, a cube is easiest to define with faces parallel to the coordinate axis.

- Defining objects in their local coordinate systems makes object reuse easier.

The copy/paste operation would become very complicated if the object have been defined in global coordinates.

Camera Coordinates

- The coordinate system that is fixed to the camera. The origin of the coordinate system is normally at the centre of projection (may also on the image plane).
- By convention, the axes of coordinate are assigned as:

One orthogonal to the image plane, z_c

One pointing up, y_c, and

One pointing to the right, x_c

- Camera parameters are define in this space

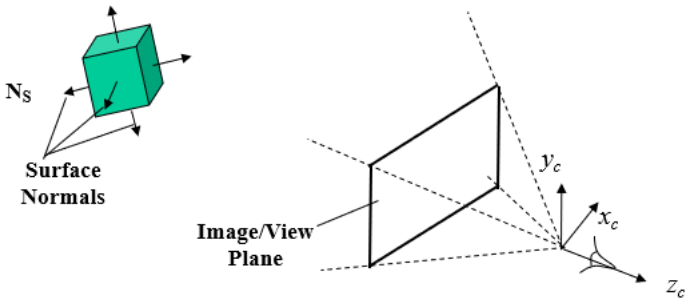
View point (position of projection centre)

View volume/frustum

View window (image plane)

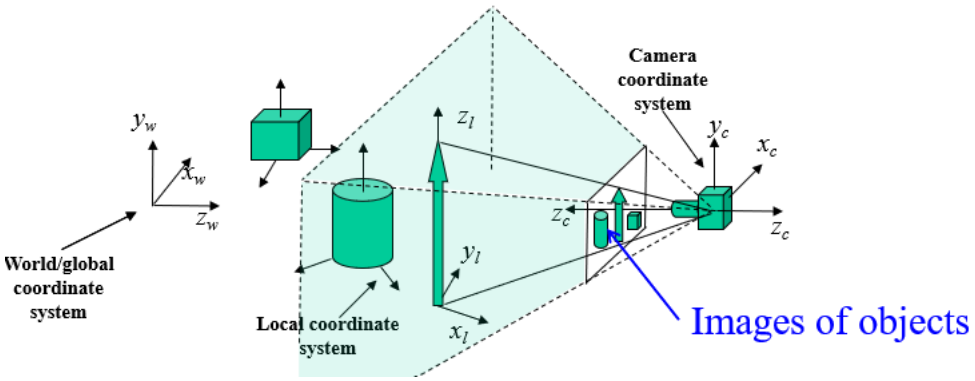
- Some operations carried out in this space:

Culling or *back surface removal* (a surface is invisible if $N_s \bullet Z_c < 0$).



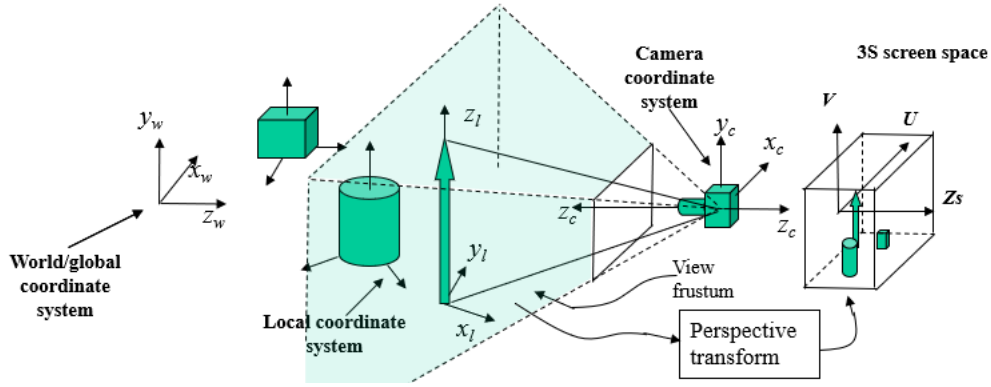
Create An Image

- In principle, knowing the relative positions of an object (vertices/points representing its surface) we can apply the principle of perspective projection to create an image, as illustrated below.



3D Screen Space

- However, to make the depth calculation simpler (z-buffer algorithm), in a real graphics system, the view frustum is mapped to a **cuboid** called *3D screen space*.
- This is done through *perspective transformation.*



- This is a abstract space with its dimensions being specified by mapping the view frustum into:

2D (square) virtual screen (U:[-1,1], V: [-1,1])

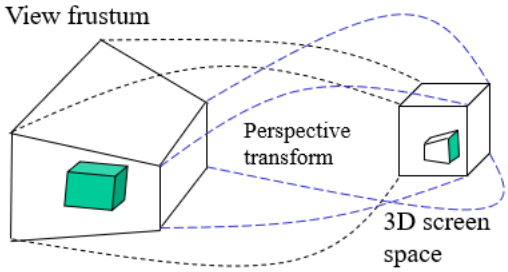
1D of depth value (Z_S:[0,1])

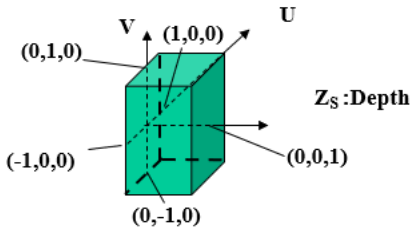
These called *normalised coordinates*

- Task to do:

hidden surface (obscured by other obj.) *removal* – surfaces that are obscured by others are removed e.g., using *z-buffer algorithm*.

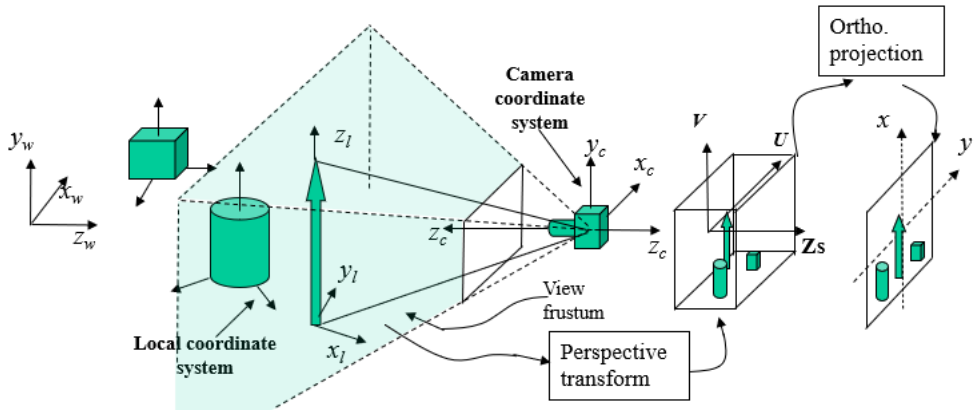
Rasterization – project visible points onto the virtual screen via orthographic projection





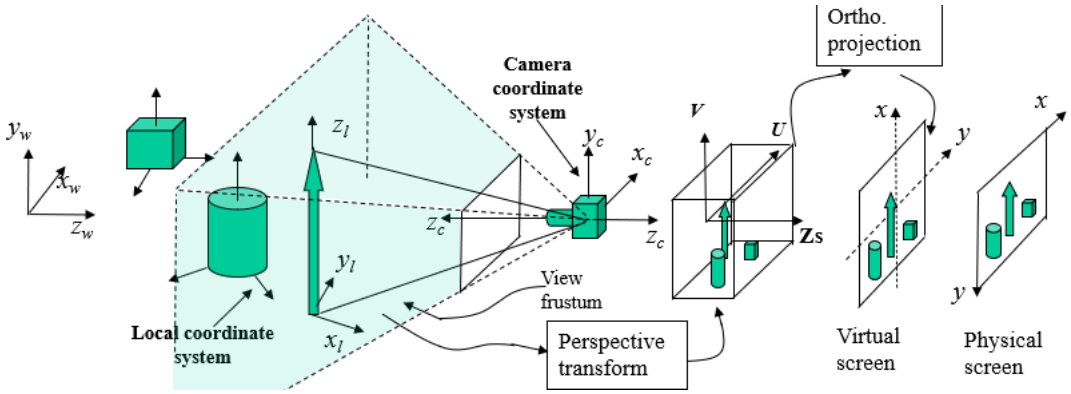
Virtual Screen

- This 3D screen space is then projected to a *square virtual screen* via orthogonal projection.
- This virtual screen has aspect ratio (width to height ratio) of 1:1 (because 3D screen space has a size of 2x2x1)



Physical Display

- Then the image on the square virtual screen is transformed to an image on *physical screen*, which has a different aspect ratio, e.g., 4:3 or 16:9
- In many cases, we don't use the entire screen. Instead, we may a window to draw the graphics. In this case, we map the square virtual window to the actual window.



Transformations and Pipeline

- The spatial relationships betweens the coordinate systems, i.e. the positions and poses, are defined by transformations (rotations and translations).

Defined as matrices.

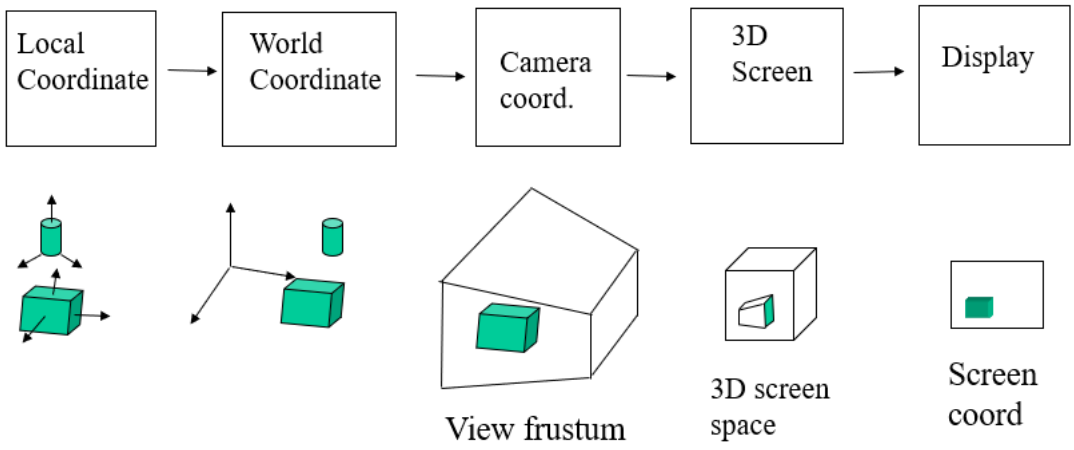
Transformations mean matrix operations.

- In the process of rendering, the vertices (their coordinate values) will go through the following transformations and necessary processing is done in the relevant coordinate systems:

Local coordinates→world coordinates→camera coordinates→ 3d screen → virtual screen → physical display

- These processing steps are fixed and very efficient algorithms for them are implemented in software and hardware as standard. They form the graphics pipeline – the processing procedure and and algorithms for rendering images.

Graphics Pipeline

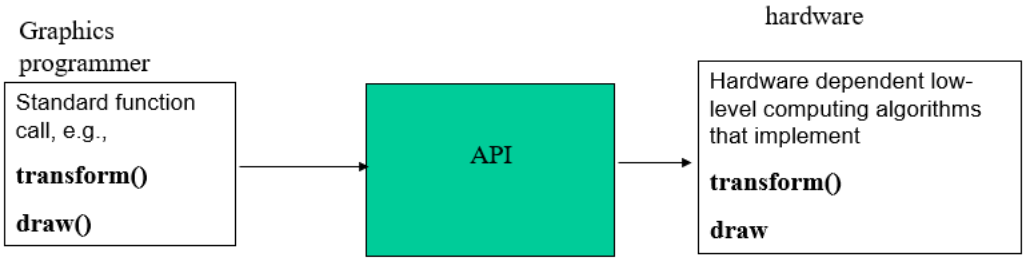


Standards and APIs

- Aim: to improve the software portability – make graphics implementations and applications hardware independent

Software developers need standardised ways to implement typical graphics operations/functions, e.g., transformation

Hardware manufacturers need a standard to develop hardware that support standard graphics functions



APIs – OpenGL

- GL – Graphics Library

Initially implemented as the graphics library (GL) on Silicon Graphics workstations (SGI) in 1980s

GL supported fast real-time rendering

GL was soon extended to other hardware systems (OpenGL, 1991) and became the de facto standard

OpenGL is the hardware-independent interface between application programs and hardware structure

Efficient processing of 3-dimensional applications

Implemented for different high-level languages, e.g., C, C++, VB, Fortran

APIs – WebGL

- WebGL is an API specifically developed for authoring 3D graphics for web applications in JavaScript.
- It is a variation of OpenGL:

  OpenGL → OpenGL ES → WebGL
- Most browsers support it, e,g., Firefox, Chrome, Safari, IE 11, Opera

API - Direct3D

- It is a 3D graphics API for Windows platform and Game consoles (XBox)
- It is one of the graphics APIs in DirectX (the other one is DirectDraw, for 2D graphics)
- Direct3D Supports various hardware acceleration
- A main competitor of OpenGL

Info: Coordinate Systems in 3DS Max

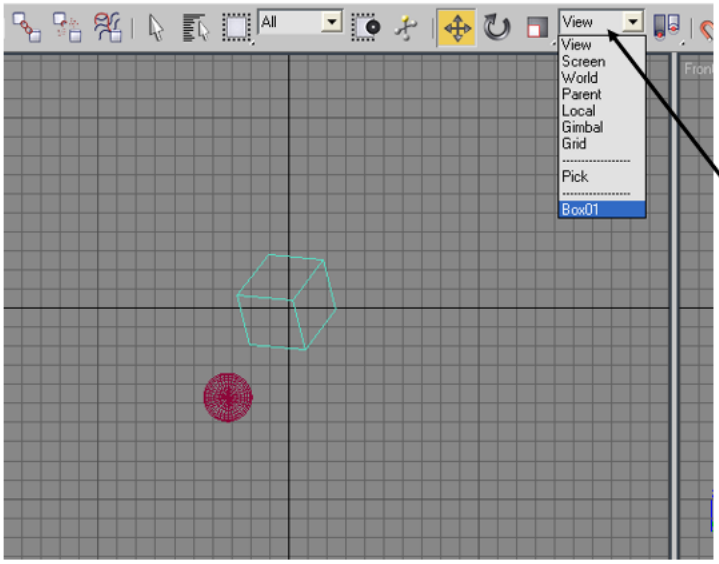
- 3DS Max provides different reference coordinate systems in which you can apply transformations to objects.

Coordinate systems:

Screen, view, World, Local, Gimbal (Euler angle), Grid, and Pick

Transformations: translation, rotation, and scaling

- These coordinate systems are used as a tool to assist you in your design, but not all are used in the graphics pipeline.



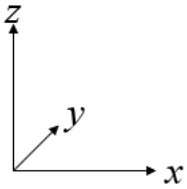
"World" Coordinate System

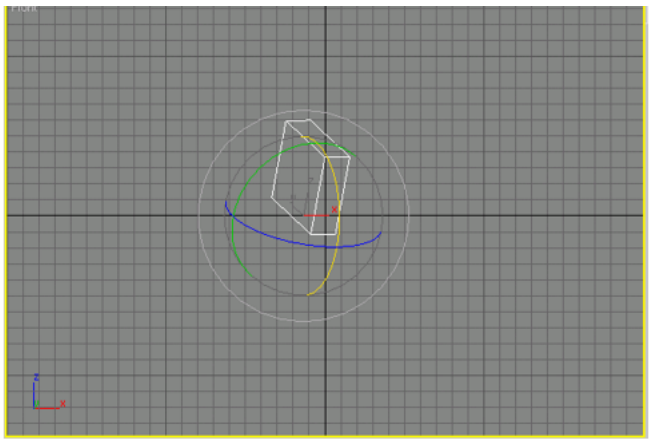
- World reference system and direction of the axes (as see from the front viewport)

X points to right.

Y: points into the screen away from you.

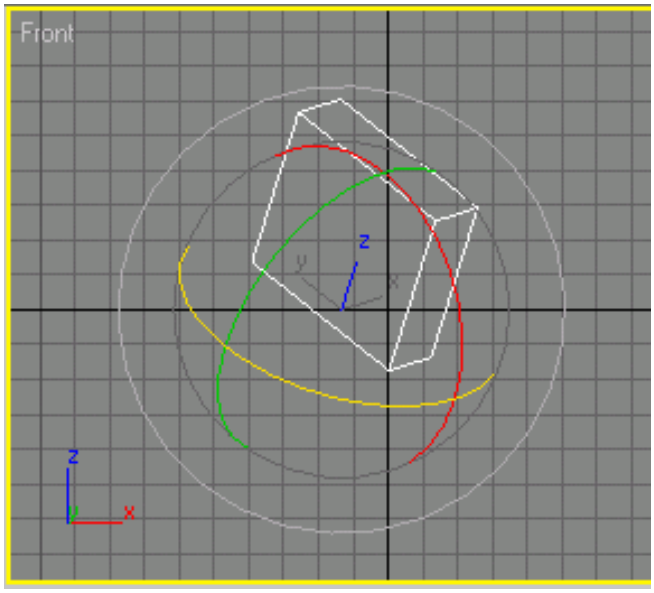
Z: upward.





"Local" Coordinate System

- Local coordinate system is attached to an object and moves with the object when you move or rotate the object



Modelling Using Geometric Primitives and Volume-Based Modelling

Geometric Primitives

- Geometric primitives: a range of geometric objects that are easy to define (described by a few parameters) and to manufacture, e.g.

Cylinder. Parameters: radius & length.

Sphere. Parameters: radius.

Cone. Parameters: radius & height.

Cube/cuboid. Parameters: lengths of its sides

Etc

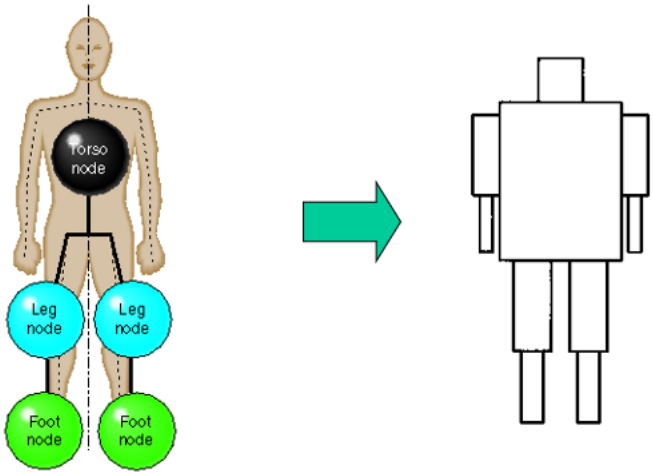
- Primitives can be used in two ways:

Hierarchical modelling (usually surface-based), or

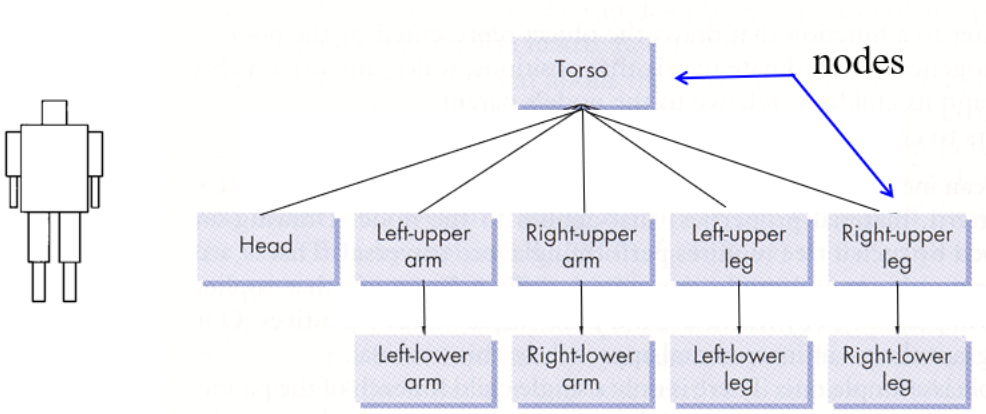
CSG (usually volume-based).

Hierarchical Modelling

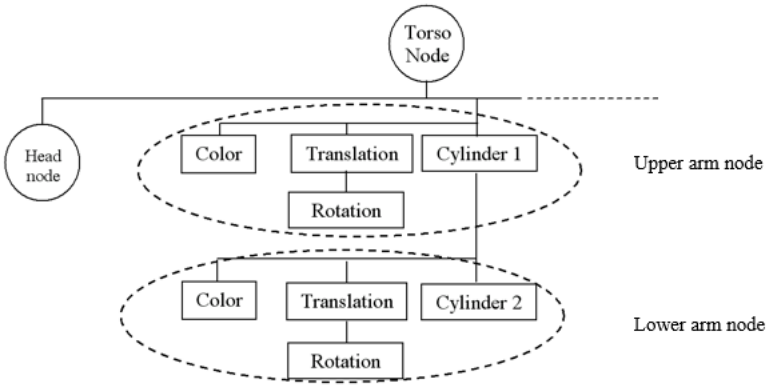
- A hierarchical model combines several primitives (through transformations) into one linked complex object.



- The overall structure of the object can be represented as a hierarchical graph that reflects the relationships between the geometric primitives or the sub-parts of the object.



- The graph can be further detailed to contain other attributes of the primitives such as materials, textures, and transformations.

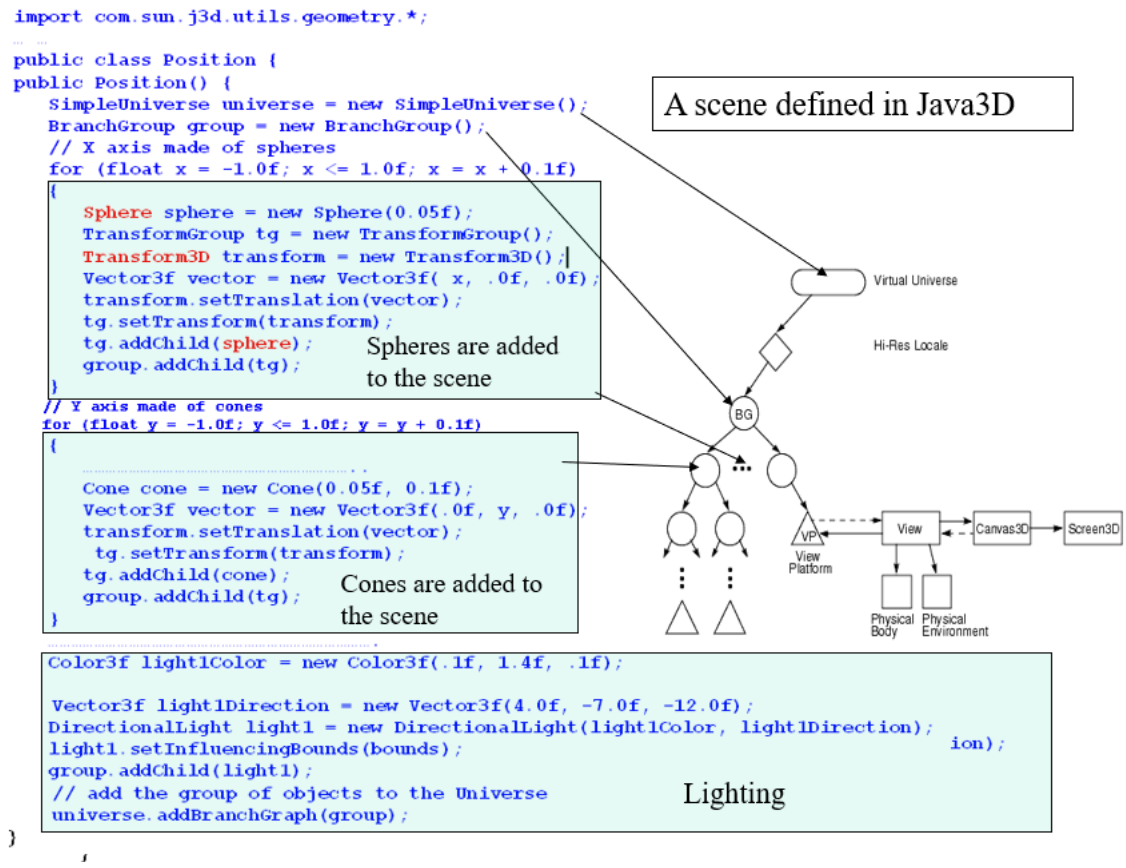


Applications of Hierarchical Modelling

- Hierarchical modelling usually find use in graphics applications written in high-level programming languages such as C and Java

A useful technique for organise and construct articulated character for animation.

Some APIs and scripting language that support hierarchical modelling: Java3D for Java, OpenGL for C (++), WebGL for JavScript, VRML.



Limitations of Hierarchical Modelling

- Hierarchical modelling is useful in applications where the organisation (connection relationships) of objects are important.
- The modelling method can potentially cause some problems:

Transparency problem (if some parts are transparent, the internal surfaces of the intersecting parts become visible)

Efficiency problem (it takes time to process the internal surfaces, overlapping parts although they contribute nothing to the rendered graphics)

Duplicated computation problem in mass/volume calculations that are necessary in physics-based simulation.

Constructive Solid Geometry

- CSG representation is a method that creates new objects by applying set (Boolean) operators and transformations on (solid) geometric primitives

Union (OR,), Intersection (AND,), Difference/subtraction (AND NOT, -)

Transformations: translation, rotation and scale

- CSG representation is motivated by CAD and CAM (Computer Aided Design and Manufacturing) because machine tools, e.g., lathes and milling machines, are designed for manufacturing primitives.

Examples

- Intersection:

The lens object is the intersection of the spheres (***L*** = ***S₁*** ***S₂***, or ***S₁*** AND ***S₂***)

- Difference (subtraction):

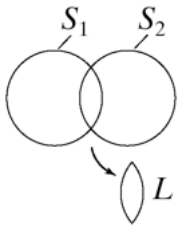
a bowl shape object can be constructed in two steps

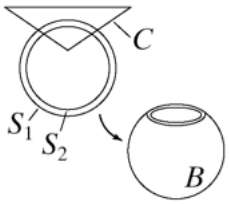
Find the difference of ***S₁*** and ***S₂*** (***A*** =***S₁*** – ***S₂***, or ***S₁*** AND (NOT) ***S₂***)

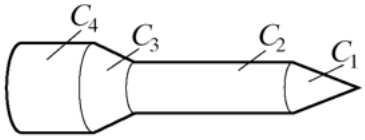
Find the difference of ***A*** and ***C***: ***B***=(***S₁*** – ***S₂***) – ***C***

- Union:

R = ***C₁*** ***C₂*** ***C₃*** ***C₄*** (read ***C₁*** OR ***C₂*** OR….)







CSG: Shell v.s. Volume

- Truly CSG objects are solids (instead of “shells” or “skins” as in the case of a polygon-mesh model)

This causes rendering difficulties for renderers such as scanline renderers as they are designed to work with polygons.

For such renderers, CSG objects must be converted into polygon-mesh models before being rendered.

In principle, a raytracing renderer can render CSG objects directly.

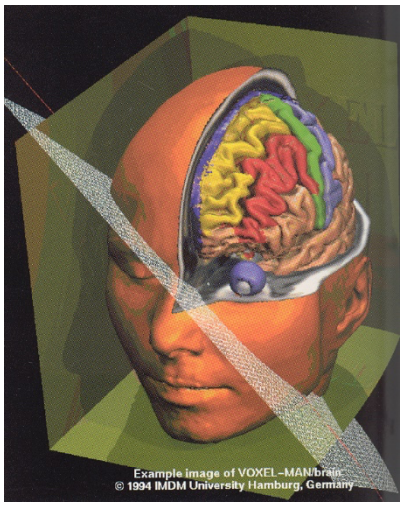
- In 3DS Max, objects obtained from Boolean operations are not solid.

Volume Based Modelling

- Some graphics applications require modelling of the skin/shell **AND** the internal structures of objects.

- Such models cannot be achieved by surface-based methods: no matter how many layers of surfaces you have, the interior is hollow.

Methods for representing volumes are needed.



Octree

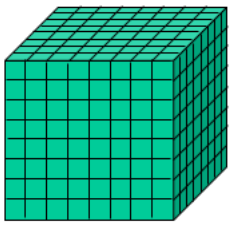
- We call a method capable of representing solids being volumetric (v.s. surface presentation).
- One of the volume-based modelling methods is the so called *octree space-division* method.
- The name comes from the fact that the underlying structure of the resulted data can be organised as an octree – a tree structure that each of its nodes has eight branches.
- The idea of the method is simple and intuitive: label the space occupied by an object.
- The method involves the use of two schemes:
    - A tessellation scheme: the way to divide the world space of the objects into small pieces that we call voxels.
    - A labelling scheme: the way to enumerate the resulted voxels to create a numeric representation of a solid.

World Space

- The world space is the minimum envelope (bounding volume) of an object.
- Rectangular boxes are often used as the world space for their simplicity for tessellation. For example, a rectangular box/cuboid can be used as the bounding volume of a human head.(See image above)

Tessellation

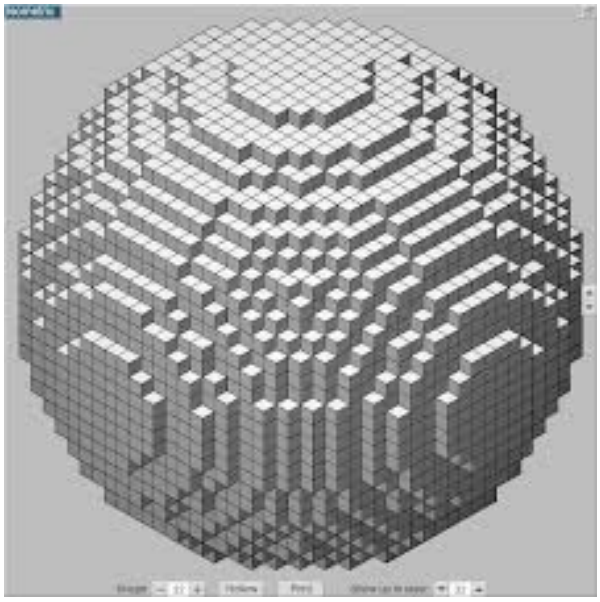
- Tessellation is crucial for the method: it affects the modelling and storage efficiency.
- Different tessellation schemes can be used depending upon the nature of the objects being modelled.
- A simplest tessellation scheme is to divide the world space into uniform rectangular (e.g., cubic) blocks/voxels.



Cubic voxel

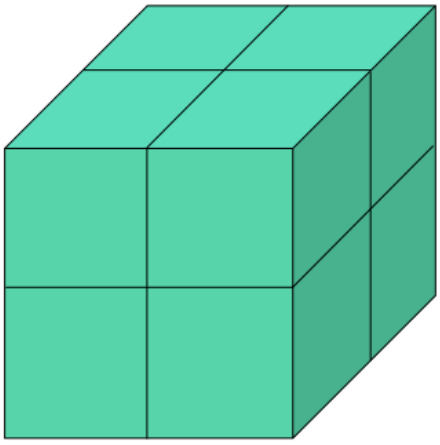
Uniform Tessellation

- With this tessellation, an object, e.g., a sphere/cylinder, can be represented by the voxels it occupies/contains

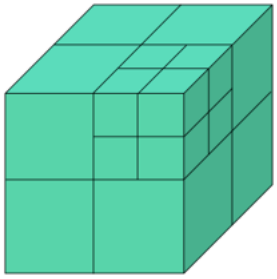


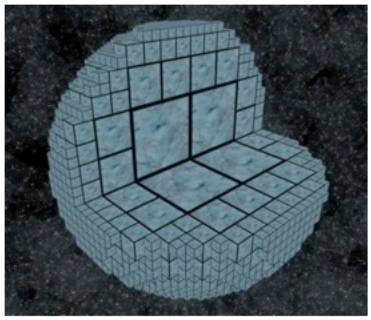
Non-Uniform Tessellation

- For uniform tessellation, we have to balance between the accuracy of the representation and the amount of data created.
- A more useful division scheme is to divide the space according to the actual structure of an object – finer division for the complex parts or regions of the object.
- Non-uniform tessellation works by first dividing world space into **eight octants** that form 8 "big" voxels and inspecting how the voxels represents the object and then deciding if any further division is made.



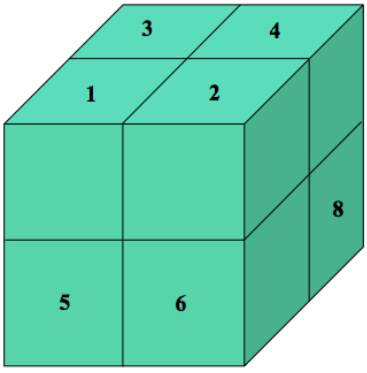
- If a “big” voxel is ***fully occupied or not occupied*** at all by the object, no further division is necessary for that voxel.
- If a voxel is partially occupied, further division is needed to decide the space/volume occupied by the object (e.g., the top-right voxel in the front side of the left figure, below).
- The process is repeated recursively until the desired precision is achieved.



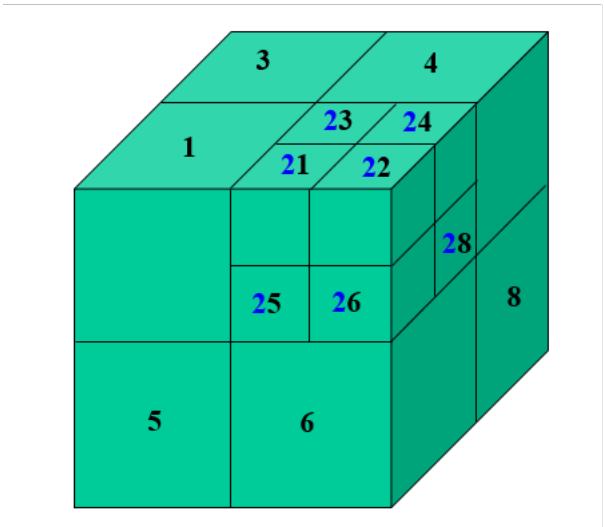


Labelling

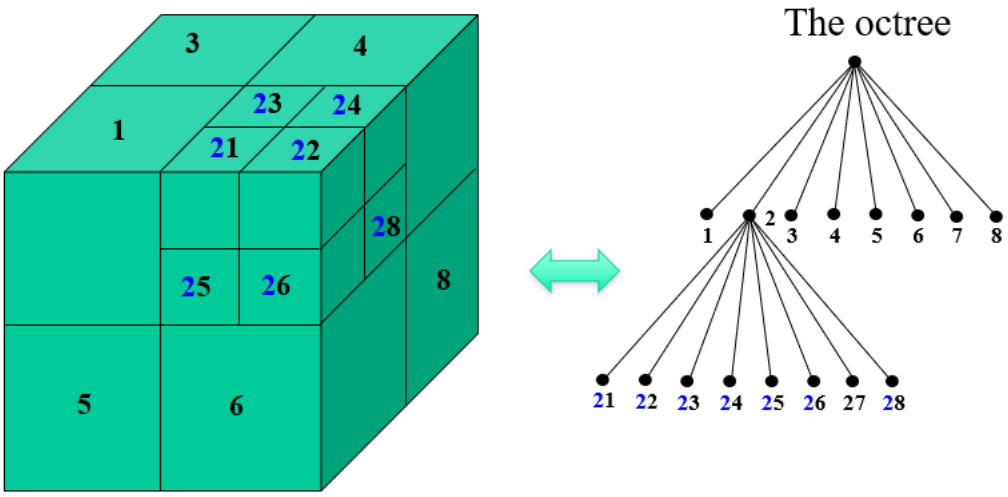
- To describe the voxels that form an object and their positions in the world space, these voxels need to indexed.
- A typical indexing scheme is to assign the indices of voxels in such an order: at a level of tessellation start from the top-left voxel, move to the right, then to the back, and to the bottom:



The same labelling scheme is recursively applied to the voxels in the next level of tessellation/division, but the labels for next level of tessellation carry the index of the upper level in the form of a leading digit to identify the position of the parent voxel at the upper level,



Octree-Labeling



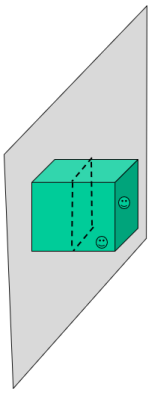
BSP

- Binary Space Partition: developed from the same idea of quadtree
- Applicable 2D, 3D or higher dimensional spaces
- Useful for fast determining space occupancy for some applications

Ray tracing rendering

Collision detection

Path planning



Modelling with Polygon Mesh

Evaluate 3D Modelling Methods

- There are many ways to represent the shape and volume of an object

So far, we have introduced hierarchical modelling, CSG, Octree.

- There are many reasons to prefer one method to the others:

How well/accurate it represents the objects of interest,

How easy it is to render (or convert to polygons),

How compact it is to store and transmit,

How easy it is to create object models.

by hand, procedurally, or automatically,

how easy it is to interact with,

modifying it, animating it.

How easy it is to perform geometric computations on it

Distance, intersection, normal vectors, curvature, …

Modelling With Polygons

- Among the various modelling methods, using polygonal facets to approximate curved surfaces is perhaps the most popular approach. In this approach, the connected polygonal facets forms a polygon mesh.
- A polygon mesh is defined as a connected collection of polygonal facets *along with the directional information* (normal vector) that indicates the direction that each facet is facing.
- The mesh forms the “skin” of an object, so the representation is surface-based (v.s. volume-based, such as octree).

Polygon Properties

- Two requirements on polygon mesh:

It must be “watertight”. This is to ensure that it forms a continuous surface, even when it is deformed.

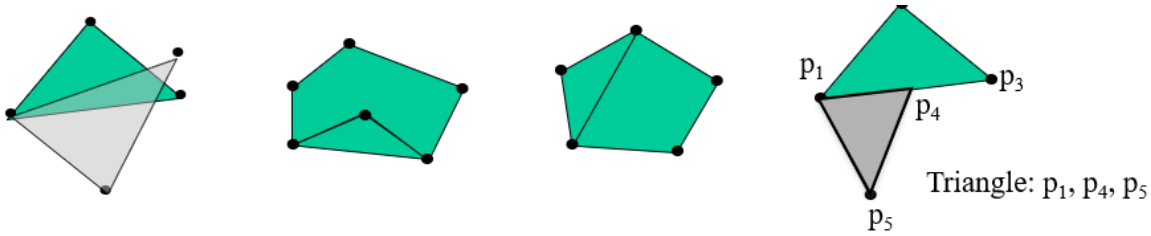
Every single polygon is planar - to make it easy to perform shading and rendering operations.

- These requirements are equivalent to requiring a polygon (a facet) to have:

straight edges,

facets intersect each other only at edges and vertices, and

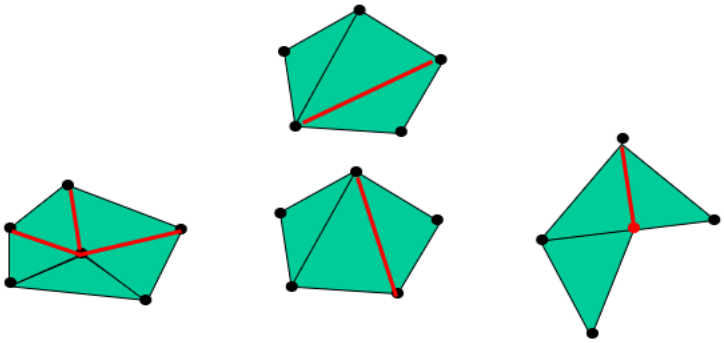
edges only intersect at vertices



Triangular Polygons

If we use only triangles in building up polygon meshes, these requirements can be automatically met.

In 3DS Max, you can draw shapes/polygons with more than three vertices, but the software automatically converts them into meshes of triangles.



- In fact, 3DS Max supports two types of polygon meshes:

Editable Mesh: inherited from early versions of 3DS Max and requires polygons to be triangular.

Editable Poly that accepts non-triangular polygons, which improves user's convenience and efficiency.

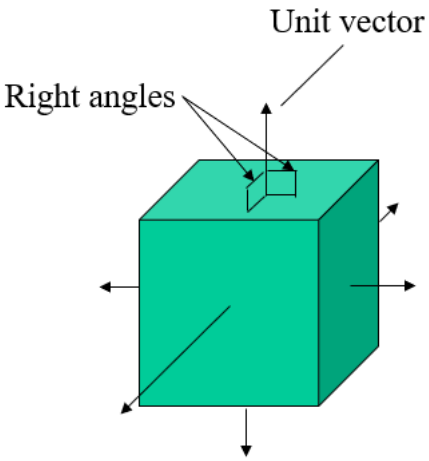
Editable Mesh emphases vertex-based operations whereas Editable Poly focus more on polygon-based operations and is believed to be more powerful.

- To use a geometric primitive (shape) as the starting object for creating more generic shapes, the primitive, e.g., a sphere, must be converted to Editable Mesh or Editable Poly (choose from the menu by right clicking it).

Surface Normals

- In CG, a surface has two sides: inside and outside. A surface is only visible when we see it from outside. Seeing from inside, a surface is transparent.

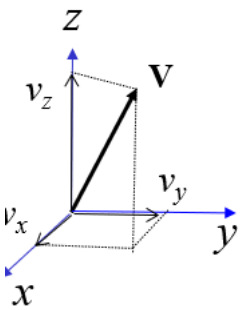
- The outside of a surface is indicated by its normal – a unit vector that is orthogonal to the surface where it adheres to and points away from the surface.



Vectors

- A vector is a math (geometry) entity that has a direction and length – you can think it as an arrow.
- In math form, a vector is written as a column matrix:

$$\mathbf{V} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \quad \text{or} \quad \mathbf{V} = \begin{bmatrix} v_x, v_y, v_z \end{bmatrix}^T$$



where v_x, v_y and v_z are the components (called projections) of the vector along the coordinate axes.

- Its length is calculated as

$$l = \sqrt{v_x^2 + v_y^2 + v_z^2}$$

Unit Vector & Normalisation

- A unit vector is a vector that has a unit length, i.e.,

$$l = \sqrt{v_x^2 + v_y^2 + v_z^2} = 1$$

- We normally denote a unit vector with **bold case** letter **n** and its components by n_x, n_y and n_z, i.e.,

$$\mathbf{n} = \left[\mathrm{n_x}, \mathrm{n_y}, \mathrm{n_z}\right]^T$$

- A vector that is not unit vector can be normalised to make it a unit one: For example, a vector **V** can be normalised:

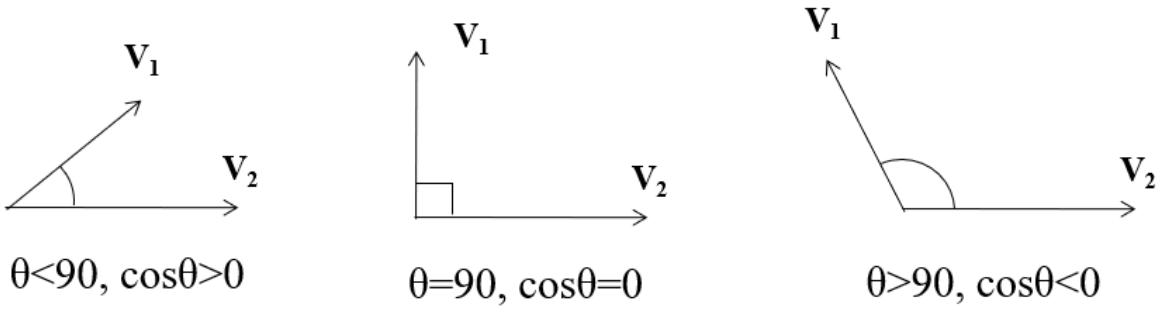
$$\mathbf{n} = \frac{\mathbf{V}}{l} = \frac{\mathbf{V}}{\sqrt{v_x^2 + v_y^2 + v_z^2}} = \begin{bmatrix} v_x / l \\ v_y / l \\ v_z / l \end{bmatrix}$$

Dot Product of Vectors

- The dot product of two vectors,v_1 and v_2 are defined as

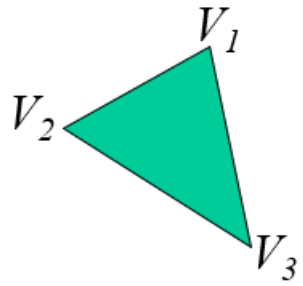
$$\mathbf{V_1} \bullet \mathbf{V_2} = l_1 l_2 \cos\theta = \left(\sqrt{v_{1x}^2 + v_{1y}^2 + v_{1z}^2}\sqrt{v_{2x}^2 + v_{2y}^2 + v_{2z}^2}\right)\cos\theta$$

- Therefore we can estimate, from the value of the dot product, the range of the angle between two vectors:
    - if the dot product is 0, the angle between them is 90 degrees,
    - if the dot product is greater than 0, the angle is less than 90 degrees,
    - if the dot product is less than 0, the angle is greater than 90 degrees.

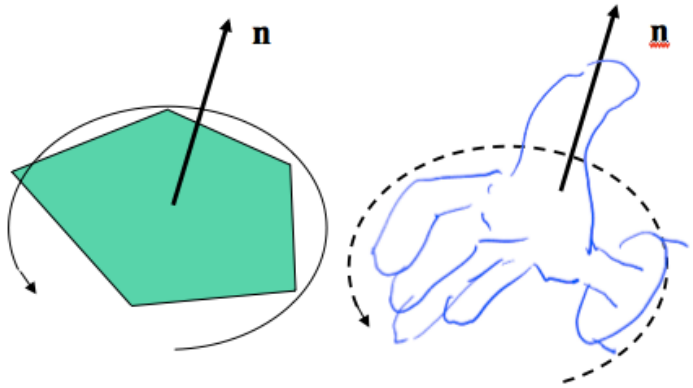


Polygon Normal Vectors

- In CG, a polygon is specified by its vertices, e.g., V1, V2, V3



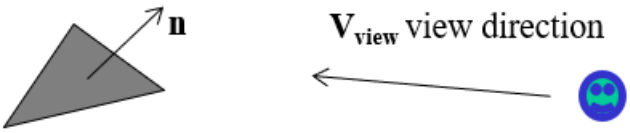
- The convention for specifying normal direction is that if we visit the vertices in counter-clockwise order, the direction of the normal always points to your side.



- Therefore, normal direction depends on the order that the vertices are given (by the order of clicking them as in 3DS max or by the sequence of listing their coordinates in a program)
- By the convention, (V1, V2, V3), (V2, V3, V1) or (V3, V1, V2) define the same triangle. But (V1, V3, V2), (V3, V2, V1), or (V2, V1, V3) define a different triangle (i.e., they face at different directions).

Use of Normals

- Normals help in deciding which surface is visible and which is not. This is important, because we don't want to waste computing resources on invisible surfaces, i.e., back-facing surfaces (surface with $\mathbf{n}.\mathbf{V}_{view}>0$) – They are removed by operation called *back surface culling*. Which relies on surface normal and viewing direction.

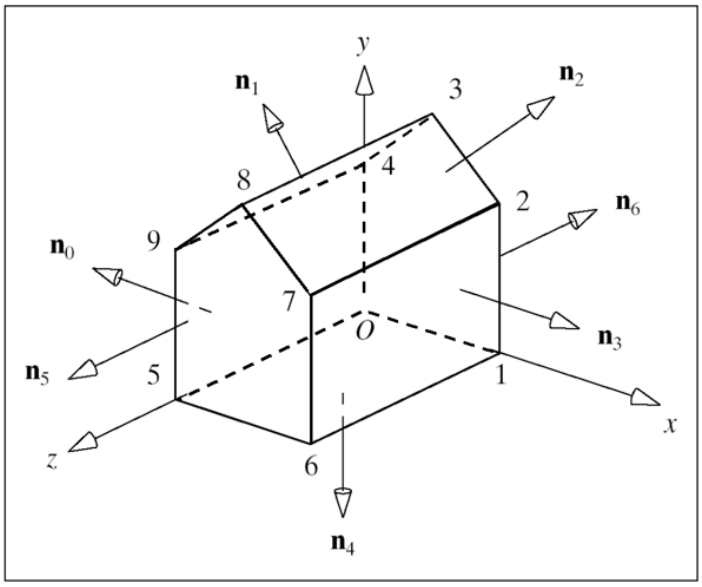


- They are also used in calculating the amount of light reflected from a surface – this is called shading (more on this topic in later lectures)

Define A Polygon Mesh

- There are typically two types of information included in a polygon-mesh model:
    - Geometric information:
        - The location of the vertices – coordinates.

        - The connectivity information - which vertices make up which polygon (vertex enumeration).
    - Associated data:
        - Values of normal vectors. These can be calculated from the coordinates of the enumerated vertices, so usually you don't have to supply them explicitly.
        - Texture coordinates - coordinates for applying textures.
- The *collection* of such information forms a *polygon-mesh model* and is stored in computers as data of some format.
    - Geometric information
        - *Ordered* list of coordinates of vertices.
        - *Ordered* list of the indices of the vertices that make up each individual polygons.
    - Associated data:
        - *Ordered* list of normal coordinates (in software packages like 3DS Max, they can be derived from the polygon lists if you don't provide them, but you may need to provide them if you do CG by programming)
        - *Ordered* list of texture coordinates (not all polygon have textures)
- This is what is called a CG *model*. What you see on a computer screen is the image/representation of a model, not the model itself!



- The simple shape is defined by three lists:
  - Vertex list (vertex coordinate values)
  - Face (polygon) list (ordered list of the vertices associated with a face)
  - Normal list (You may need them if you do CG by programming)

Vertex list

vertex	x	y	z
0	0	0	0
1	1	0	0
2	1	1	0
3	0.5	1.5	0
4	0	1	0
5	0	0	1
6	1	0	1
7	1	1	1
8	0.5	1.5	1
9	0	1	1

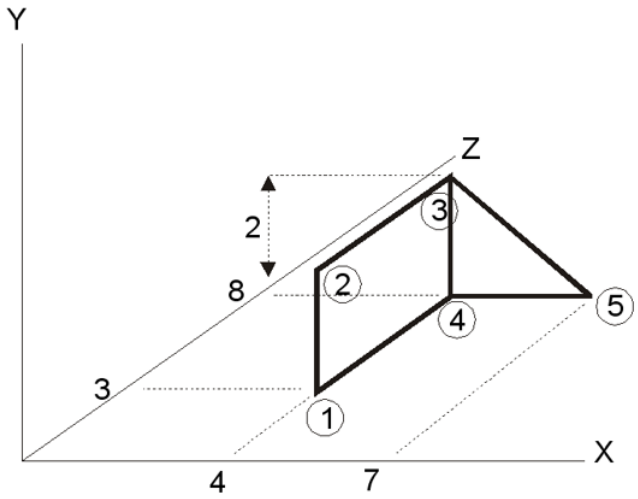
Polygon list

Face	Vertices	Associated Nor.
0	0,5,9,4	0,0,0,0
1	3,4,9,8	1,1,1,1
2	2,3,8,7	2,2,2,2
3	1,2,7,6	3,3,3,3
4	0,1,6,5	4,4,4,4
5	5,6,7,8,9	5,5,5,5,5
6	0,4,3,2,1	6,6,6,6,6

Normal list

Normal	n_x	n_y	n_z
0	-1	0	0
1	-0.707	0.707	0
2	0.707	0.707	0
3	1	0	0
4	0	-1	0
5	0	0	1
6	0	0	-1

Another Example



Vertex list
1 (4,0,3)
2 (4,2,3)
3 (4,2,8)
4 (4,0,8)
5 (7,0,8)

Polygon list
1, 2, 3, 4
3, 4, 5

Can you see any problems with the model?

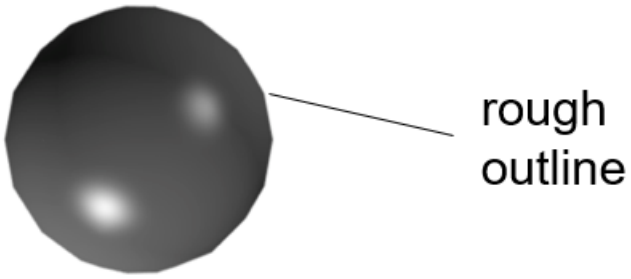
Advantages of Polygon Meshes

- Polygons are the dominant force in 3D modelling:

- Creating polygonal objects is straightforward (by specifying the vertices), and sometimes it offers the simplest solution. Eg, a cube or a cuboid.
- Modelling can be quite accurate (although at the expenses of increased data/storage and degraded performance).
- Almost everything can be turned into polygons.
- We know how to render polygons quickly (graphics pipeline for polygon rendering).
    - Easy to represent (a sequence of vertices).
    - Simple for transformation (matrix operations).
- Models in other representations are often converted into polygon mesh prior to rendering.

Problems with Polygons

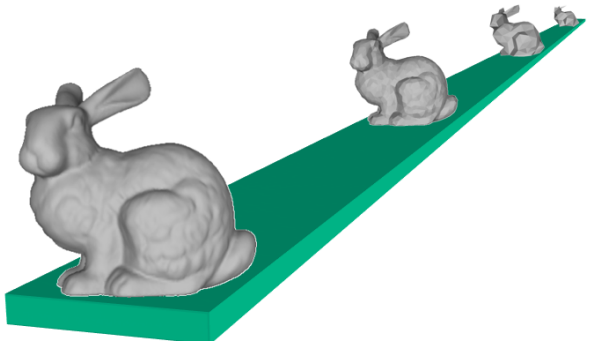
- A polygon mesh is inherently an approximation.
- Things like silhouettes/outlines can never be prefect without using very large numbers of polygons.



- Interactive design could be a problem.
    - Dragging points around is time-consuming and hard to visualise.
- Maintaining things like smoothness is difficult.
- Hard to increase, or decrease, the resolution (some schemes can be used to split or fuse polygons).

Issue of Resolutions

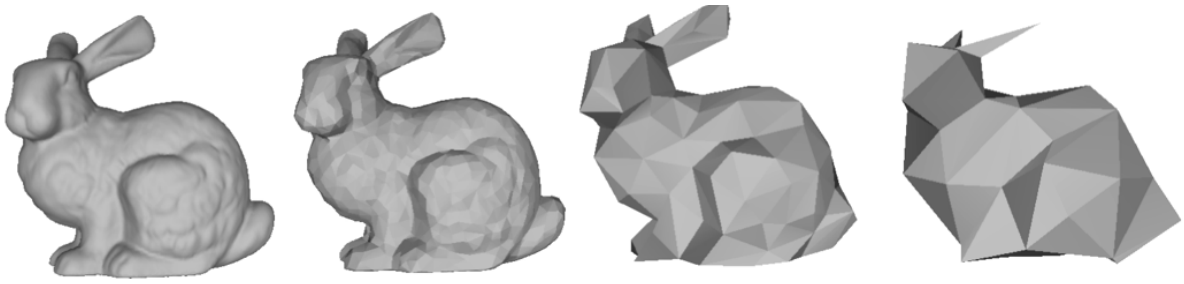
- To achieve desirable realism, models of generic (freeform) objects often contain many polygons, e.g., the model of a rabbit contains a few thousand polygons.



- Having too many polygons slows down rendering speed.
- How can we speedup rendering without sacrificing visual realism/quality ?

Varying Resolution

- A possible solution is to use models of varying resolution: use high resolution models for close-up view and low resolution ones for distant view.
- This technique is called varying level of detail (LOD)



69,451 polygons | 2502 polygons | 251 polygons | 76 polygons

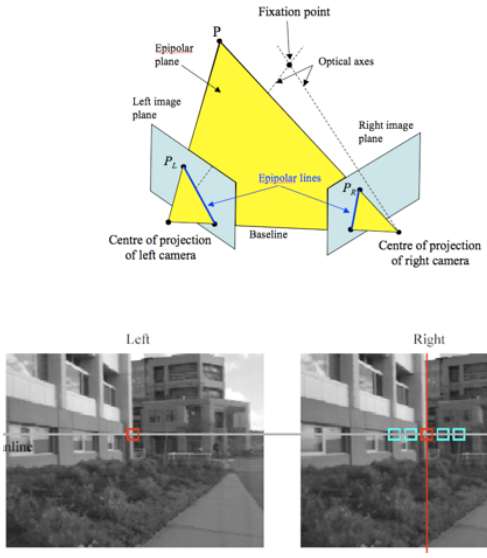
LOD

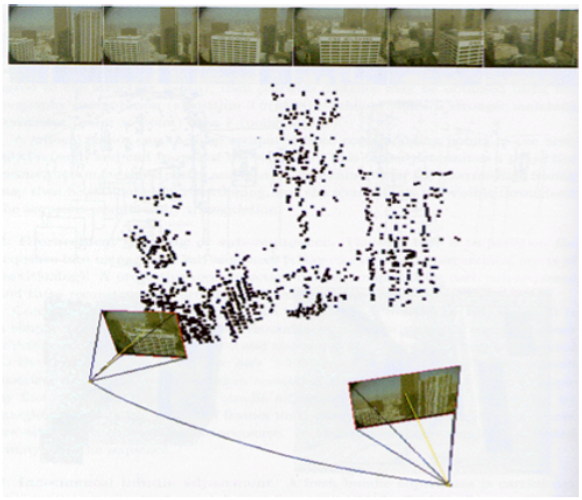
- Varying LoD can be achieved by either:
  - maintaining multiple models of different resolutions for a single object, or
  - having a high resolution model that can adapt its resolution according to viewing conditions.
- If well implemented, LoDs
  - speed up rendering,
  - is useful for progressive transmission of models across network.
- Unfortunately, varying LoD is hard to achieve – it still is an research issue.

Acquiring Polygon Mesh Model

- Manual
  - In low level graphics modelling, e.g., using OpenGL, numeric data must be supplied via programming.
  - If CG software is used, e.g., 3DS Max, Maya, AutoCAD, tools are available for assisting model creation.
- Automatic
  - Image-based modelling.
- Various scanners.

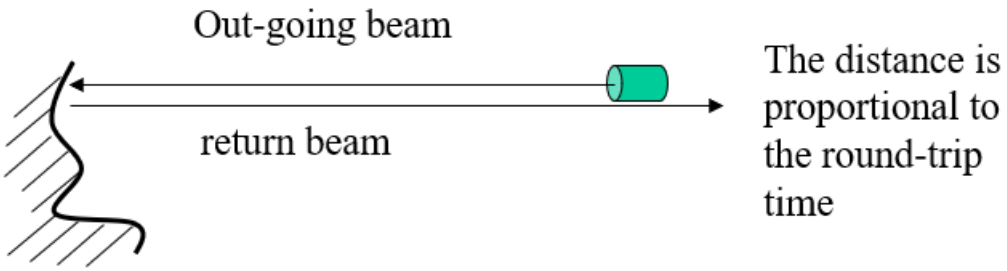
Image Based Modelling

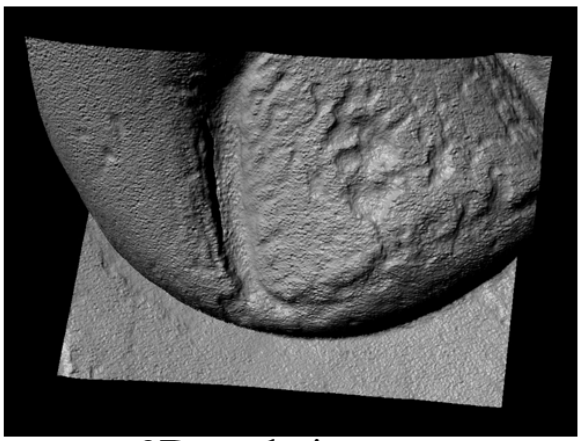




Scanner/Range Finder

- Scanners and range finder can find distances to points on surfaces.
- The distances are recorded as range data (or called depth map) that are used to reconstruct 3D shapes.





3D rendering

Photograph

- Higher resolution can be achieved by 3D depth scanning (More precise detail is obtained)

Representation of Curved Surface and Bezier Curves

Introduction

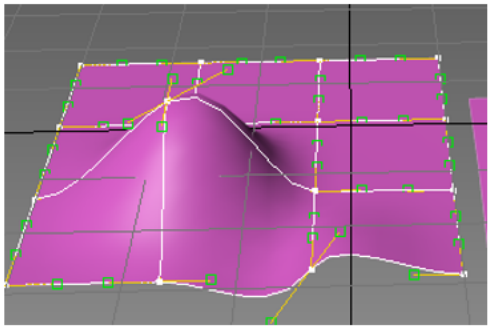
- Polygon meshes have provided good ***approximation*** to the surfaces of objects.
- But the representation is not exact:
  - the edges of the polygons are straight lines, therefore the polygons are flat,
  - the representation is accurate only at the vertices of the polygons.

Polygon Meshes Become Inadequate when:

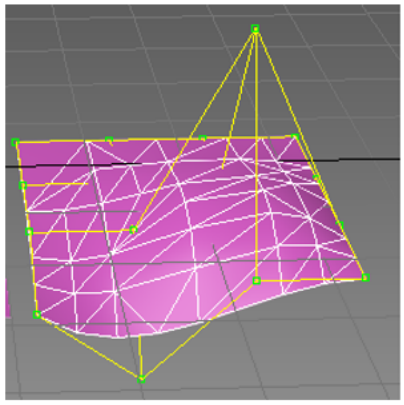
  - We need smooth, accurate or exact representations to reduce the defects in shadows, highlights and outlines.
  - We want more efficient ways for shape editing
    - Pulling vertices around is difficult and results in peaks/spikes and unwanted deformations.
  - We use graphical model to control high precision manufacturing, e.g., for CAD/CAM and 3D printing where models are transformed into real objects.

- A solution to these problems is to define "polygons" that have curved "edges" and non-flat surfaces. Such polygons are called curvilinear polygons and their surfaces are parametric surfaces.
- 3DS Max and other 3D software normally provide two types of parametric surface primitives:
  - *Patch Grid*: Bezier surface – curvilinear surface defined with Bezier curves, and
  - *NURBS Surfaces* – curvilinear surface defined with B-splines.

Patch Grid

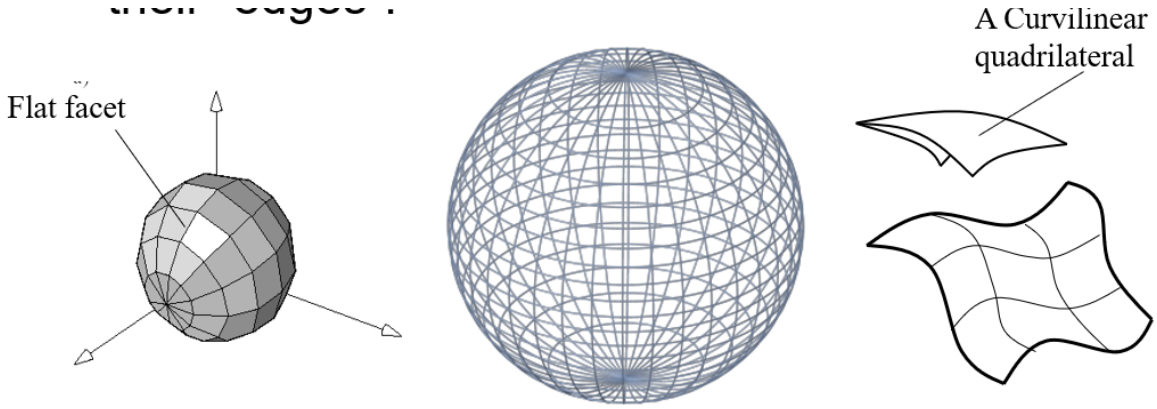


NURBS Surfaces



Curvilinear Quadrilaterals

- The parametric surfaces for 3D modelling take the form of curvilinear quadrilaterals – quadrilaterals that have parametric curves as their "edges"



- A curvilinear quadrilateral is defined by four curved edges.
- Defining a curve is more difficult than define a line. Two end points define a line fully, but this is not the case of curves – a good way to represent "good " curves is needed.
- Ways to define curves:
  - giving the discrete points along the curve – not helpful;
  - using curves that have a formula, e.g., circles, parabola, ellipses – the shapes are limited and they may not be the ones we want;
  - approximating curves using some formula – an approach used in many areas, e.g., fitting a curve to data;
  - constructing curves from simpler ones by ***blending*** – the approach adopted in CG.

Defining Curves – Polynomials

- Polynomial: sum of the products of numbers and the alphabets that represent numbers. It takes the form

$$C(u) = a_0 + a_1u^1 + a_2u^2 + a_3u^3 + ... + a_nu^n$$

  - Where a_k are coefficients (real number), n is nonnegative integer.
- The highest power in a polynomial is called the *degree* of the polynomial.
- The *order* of a polynomial is *one* more than its degree.
- Many curves have polynomials as their formulas, e.g.,
  - when n=1: $c(u) = a_0+a_1u$ is a straight line;
  - when n=2: $c(u) = a_0+a_1u + a_2u^2$ is a quadratic curve, e.g., parabolas;
  - when n=3: $c(u) = a_0+a_1u + a_2u^2 + a_3u^3$ is a cubic curve, …

Cubic Polynomials/Curves

- Polynomials also provide good approximation to many curves if we can afford to have higher degree and more terms.
- However, higher degree means many coefficients (the *a*'s in the formula) have to be used to define the shape, which is undesirable in CG.
- In CG, polynomials that have a degree higher than 3, are seldom used. Instead, the ***cubic polynomial*** (because its highest power is 3) is used. The polynomials take the form:

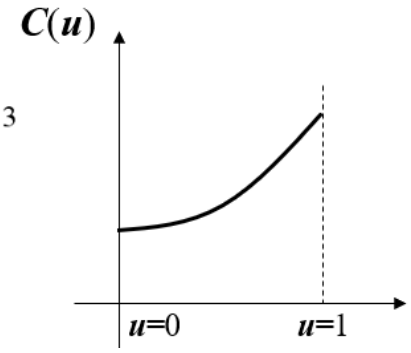
$$C(u) = a_0 + a_1 u^1 + a_2 u^2 + a_3 u^3$$

- This cubic polynomial has *4 parameters* (i.e., the a's), which control the shapes (cubic curves) it defines.

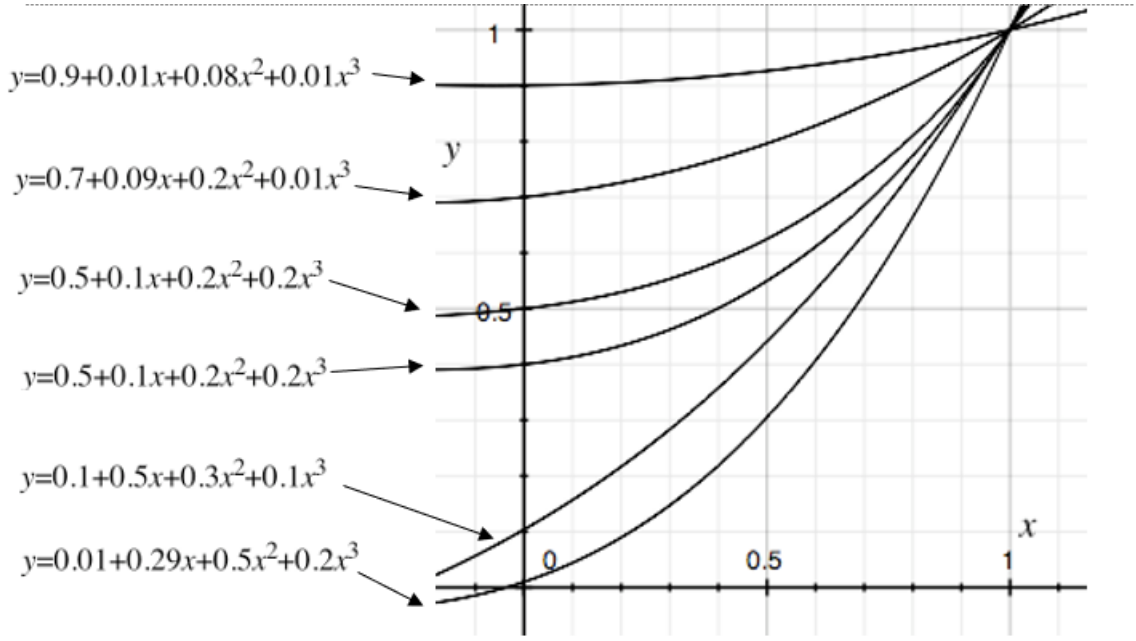
Simple Cubic Curves

- For *a given set of coefficients*, we can plot the curve out by evaluating the polynomial for the given ***u*** values (usually between 0 and 1).

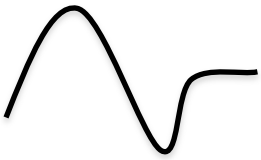
$$C(u) = \sum_{k=0}^{3} a_k u^k = a_0 + a_1 u + a_2 u^2 + a_3 u^3$$



- Because a_k's control the shape of the curve, they are called *coordinates* (*parameters* or *control points*) of the curve.
- By changing the coefficients we can obtain infinite number of different cubic curves

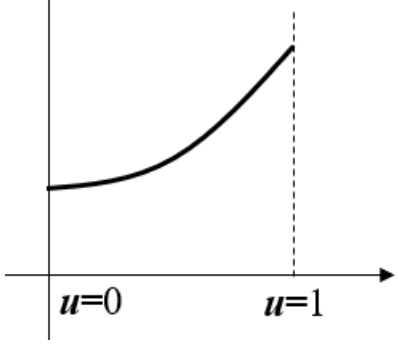


- Obviously, these shapes are rather limited in terms of the complexity (e.g., it has no bend).
- With the formula of cubic curves, no matter how the parameters (i.e., the ***a***'s) are varied, one will never be able to achieve a shape like this:

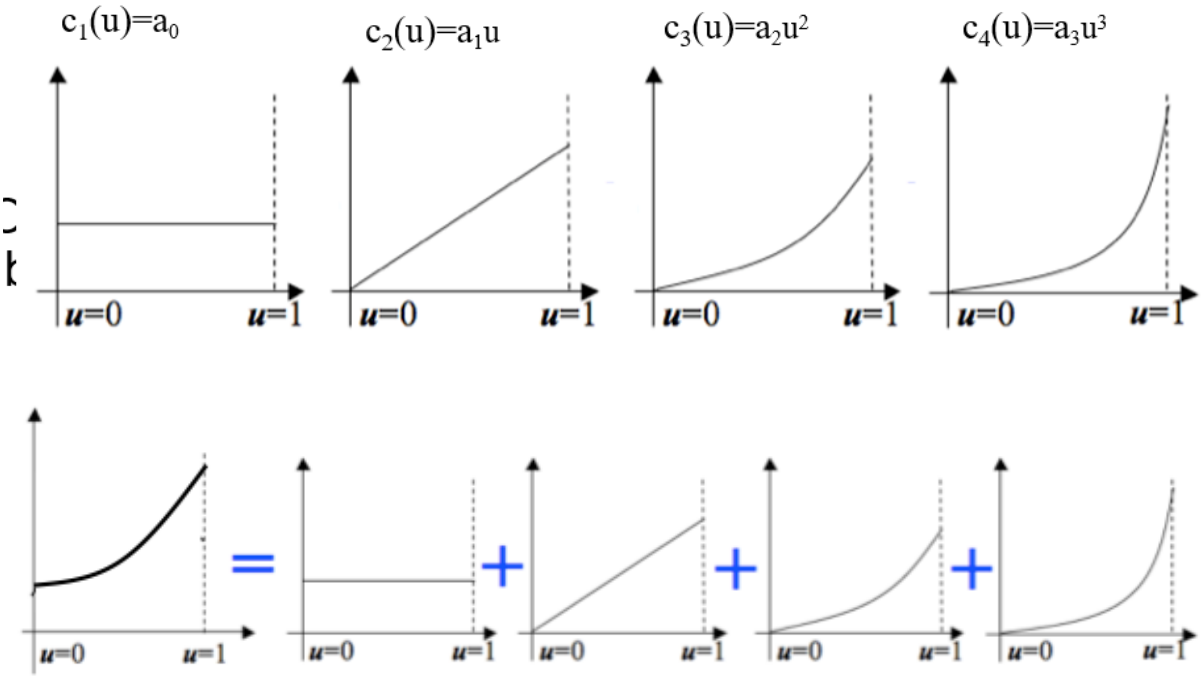


- This means the cubic polynomial is not great for graphics applications.
- However, looking closer at the cubic polynomial reveals something interesting and useful – *curve blending*.
- Now, take a look of the same curve, but from a different perspective: the cubic polynomial is formed by "blending" 4 simpler shapes:

$$C(u) = a_0 + a_1 u^1 + a_2 u^2 + a_3 u^3$$



- The polynomial has 4 terms, i.e., a_0, $a_1 u$, $a_2 u^2$ and, $a_3 u^3$. If we draw them individually:



- For this reason, we say that a generic cubic curve is a composite curve: it is obtained by blending 4 basic (simpler) curves.

Basis Functions

- If we let $B_0(u)=1$, $B_1(u)=u$, $B_2(u)=u^2$ and, $B_3(u)=u^3$, and use a different symbol P_i to replace the ***a'***s, rewrite these terms, we have

  $c_0(u)=P_0 B_0(u)$,

  $c_1(u)=P_1 B_1(u)$,

  $c_2(u)=P_2 B_2(u)$ and,

  $c_3(u)=P_3 B_3(u)$

- Because the shape of $B_0(u)=1$, $B_1(u)=u$, $B_2(u)=u^2$ and, $B_3(u)=u^3$, are fixed, we call them the basis functions.

- P_0 ,P_1, P_3, and P_4 control the weights of each basis function, hence the shapes of the blended curves, and are called control points.

Cubic Polynomial Is Not Good Enough

- We have shown that the shapes can be obtained from blending the simple basis functions are quite simple.

- This is because the basis functions are too simple: a horizontal line($B_0(u)=1$), a slant line ($B_1(u)=u$), a parabola ($B_2(u)=u^{2)}$, and a cubic curve ($B_3(u)=u^3$).

- Borrowing the idea of blending, can we use more sophisticated basis functions so that more useful (complex) curves can be created in the same way?

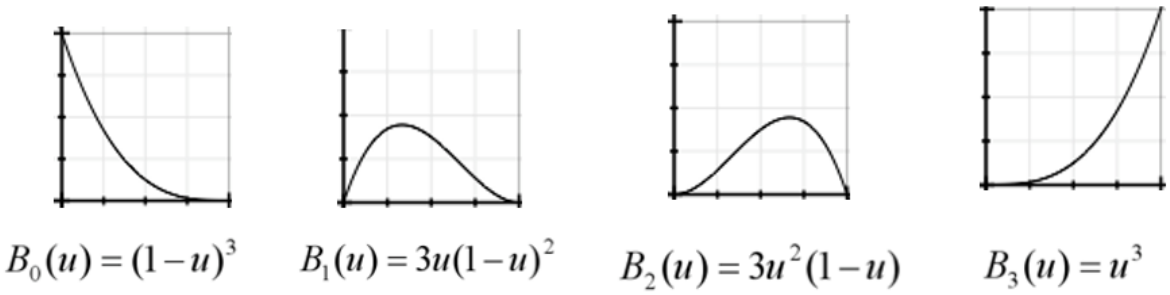
Better Basis Functions

- Indeed, better basis functions exist.
- The following set of basis functions has 4 cubic polynomials/curves:

$$B_0(u) = (1-u)^3$$
$$B_1(u) = 3u(1-u)^2$$
$$B_2(u) = 3u^2(1-u)$$
$$B_3(u) = u^3$$

$$\text{for } (0 =< u =< 1)$$

- This set of basis functions are called Bernstein cubic polynomials (functions). Their shapes look like:

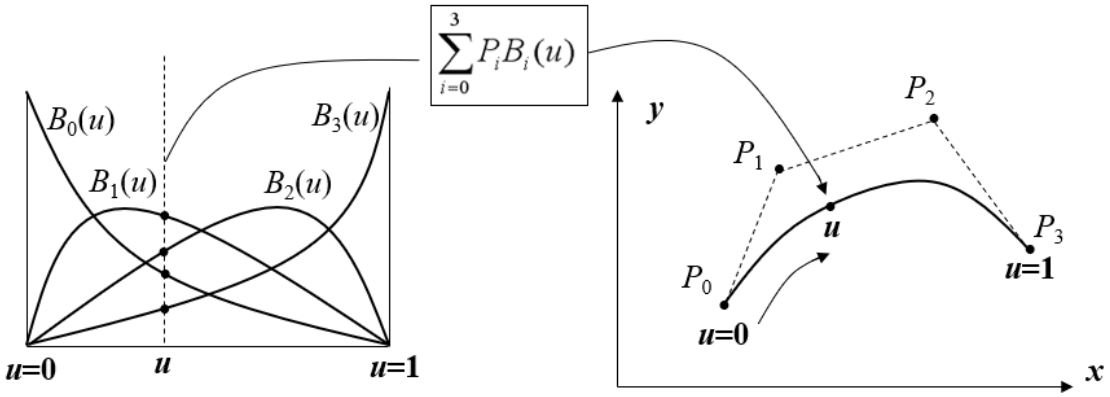


$B_0(u) = (1-u)^3$ $B_1(u) = 3u(1-u)^2$ $B_2(u) = 3u^2(1-u)$ $B_3(u) = u^3$

- The family of the curves obtained by blending this set of basis functions is called Bezier curves - named after a French engineer worked at Renault.

Bézier Curves

$$C(u) = P_0 B_0(u) + P_1 B_1(u) + P_2 B_2(u) + P_3 B_3(u) = \sum_{i=0}^{3} P_i B_i(u)$$



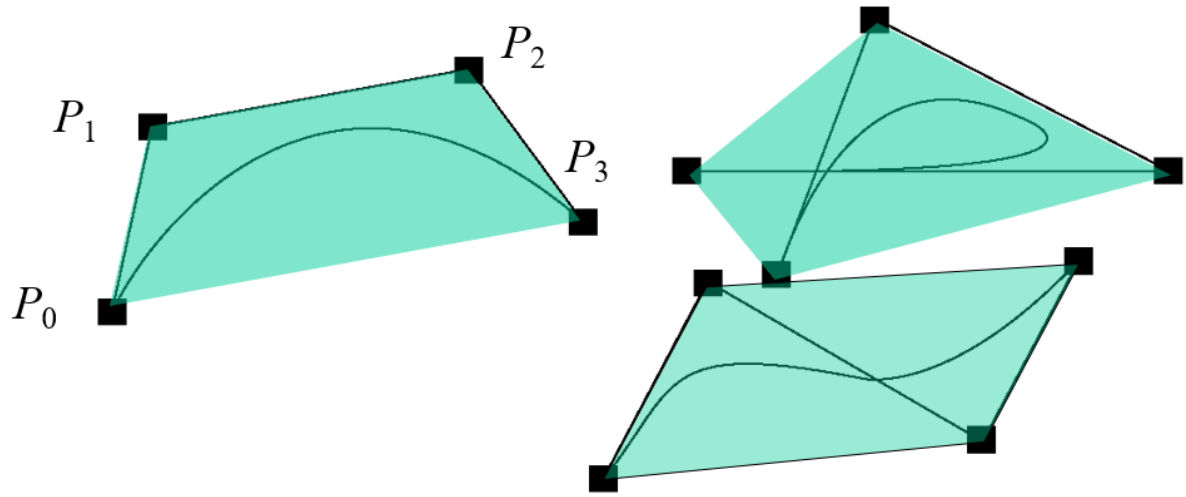
Of course, the shape of Bezier curves are controlled by the coefficients P_0, P_1, P_2, P_3 (the weights of the basis functions) – we still called them ***control points***

Properties of A Bezier Curve

- A Bezier curve has 4 control points and 4 basis functions.
- Moving any control point changes the **entire** shape of the curve.
- The curve passes through the control points at both ends. We say it interpolates the control points at both ends and approximates the middle ones.
- The first and last edges of the control polygon are tangent to the curve at the ending points.
- The curve always lies within the control polygon.

Control Points and Control Polygon

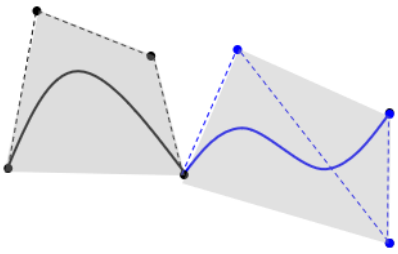
- The convex polygon formed by the control points is called control polygon.
- Bezier curves always lie inside its control polygon



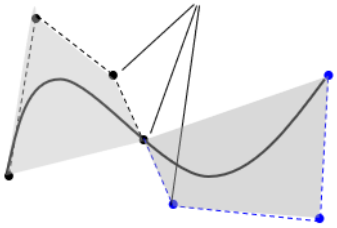
Limitations

- Having only 4 control points limits the shapes of the Bezier curves
- To obtain more complex curves, several Bezier curves need to be jointed together.
- When doing this, two conditions have to be met to ensure the smoothness of the curve at the joining point:
  - geometric continuity (no gap between the segments)

first order continuity (tangential continuity)

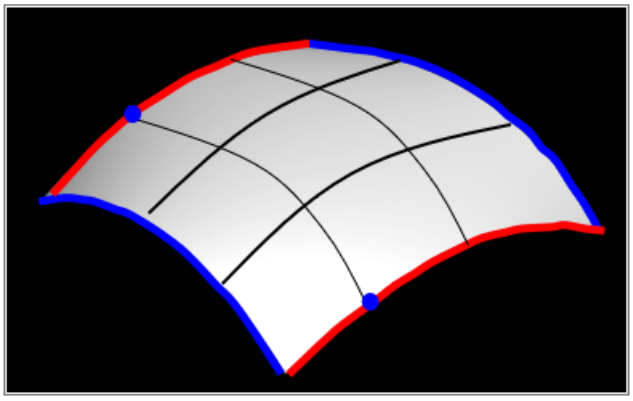


To maintain the 1st order continuity, when editing the shape we must adjust these points to ensure that they are on the same line



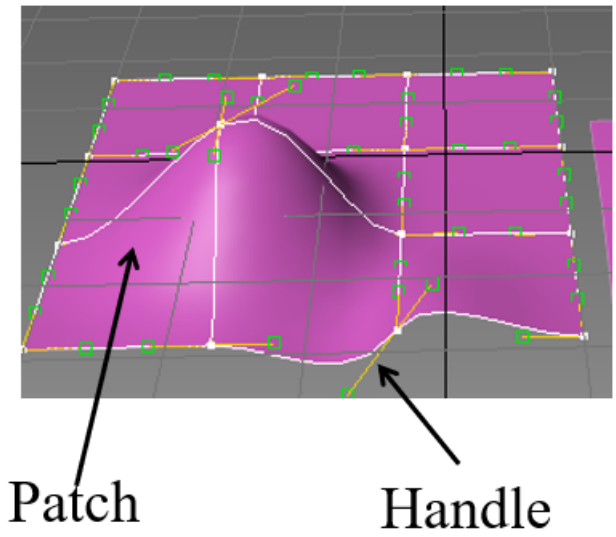
Use of Bezier Curves

- Creating 2D curves and shapes.
- Defining 3D surface patches (as called in 3DS max) – using them as the edge of curvilinear quadrilaterals.
- The interior cross-sections of the patch are also Bezier curves.
- Advantages: to control the shape of the patch, we only need to change a few control points, which make shape editing very easy.



Patch Grid Objects

- Patch Grid objects: *Quad Patch* & *Tri Patch* (They have rectangular and triangular patches respectively when converted into an *editable patch*)
- An editable patch has several sub-patches, and each sub-patch has vertices and edges and handles that control the tangent at a vertex.



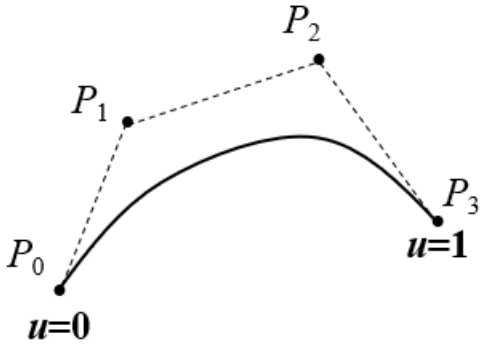
B-Spline and NURBS

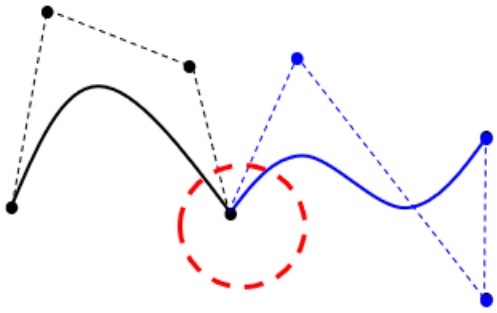
Overview

- B-splines and definitions
- B-spline as basis functions and B-spine curves
- NURBS
- Modelling surface using splines

Limitations of Bezier Curve

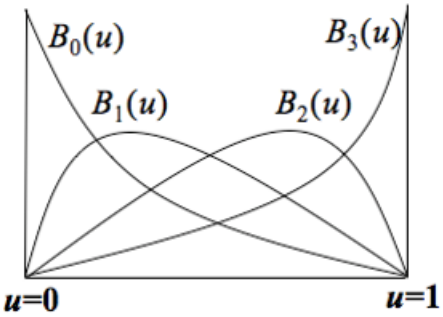
- Only four control points per segment – poor local shape control
- Difficult to control the smoothness across neighbouring segments





Better Basis Functions

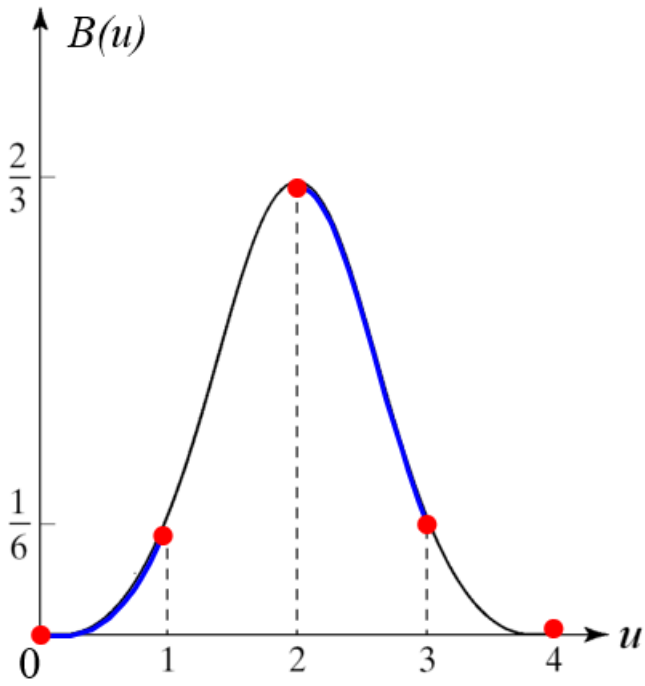
- These problems are related to the basis functions and the way Bezier cures are constructed.



- There exists a particular family of curves, called B-splines, from which curves (and surfaces) possessing better properties can be constructed.
- Used as basis functions, B-splines will
  - allow more control points, which means no need for connecting segments,
  - have better ***local control*** - changing a control point only affects a small portion of the curves.

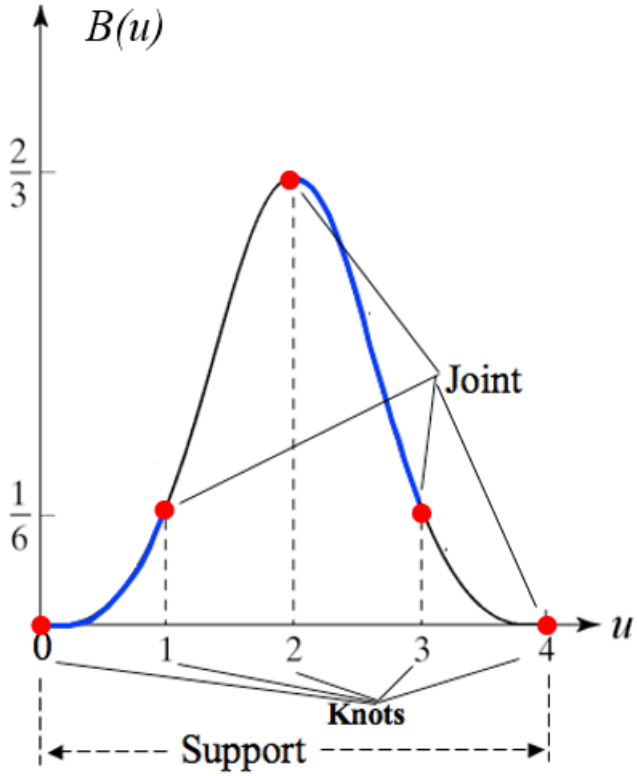
B-Splines

- In CG, spline refers to any composite curve formed with polynomial sections (i.e., each section has a polynomial as its function) satisfying specified continuity/smoothness conditions along the connection boundaries
- B-splines are a family of composite curves (B-splines of degree 0, 1, 3, etc).
- The one we used here is Cubic B-spline
  - It consists of four segments
  - Each segment has its own defining polynomial.

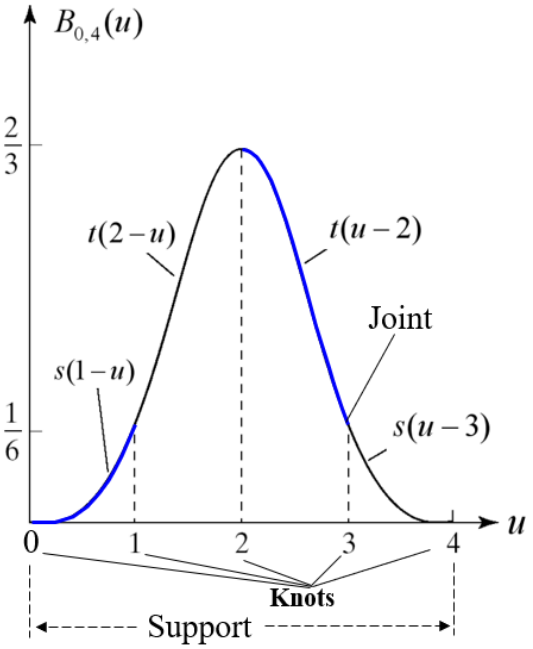


B-Splines: Some Definitions

- The domain of a spline is called the *support* of the spline.
- The conjunctions of the segments are called *joints.*
- *Knots* refer to the parameter values corresponding the conjunctions (or the indices of the values if the intervals are equal).
- *Knot vector* of a spline is the array of values of knots.
    - E.g. [0, 1, 2, 3, 4]



Cubic B-Spline In Math Form



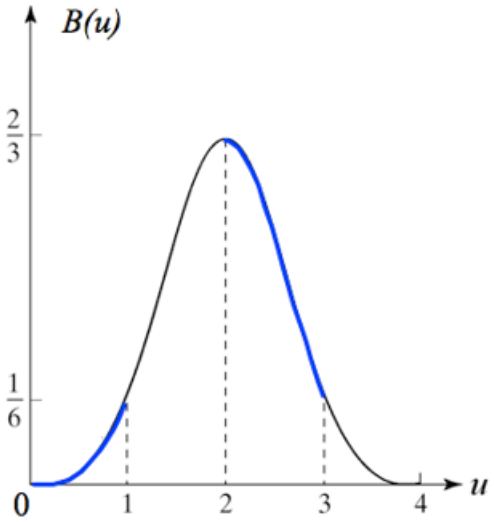
$$B_{0,4}(u) = \begin{cases} s(1-u) & 0 \le u \le 1 \\ t(2-u) & 1 \le u \le 2 \\ t(u-2) & 2 \le u \le 3 \\ s(u-3) & 3 \le u \le 4 \\ 0 & otherwise \end{cases} \quad for$$

With s(x) and t(x) defined by:

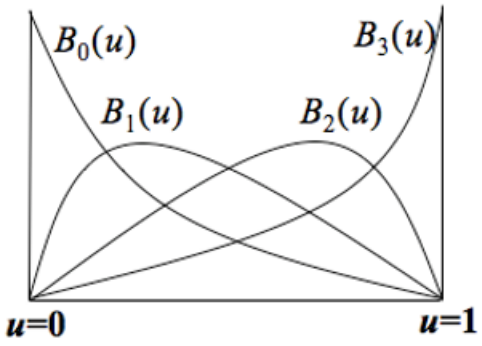
$$s(x) = \frac{1}{6}(1-x)^3$$

$$t(x) = \frac{1}{6}(3x^3 - 6x^2 + 4)$$

One composite curve consisting of 4 sections/segments.



Four different curves



Create Curves Using B-Spline

- Just as obtaining Bezier curves by blending Bernstein basis functions, new curves can be obtained by blending B-splines.
- The resulted new curves are called *B-spline curves* (they are sometimes also called B-splines).

B-Spline Curves

- Like Bezier curve, B-spline curve can be formulated as

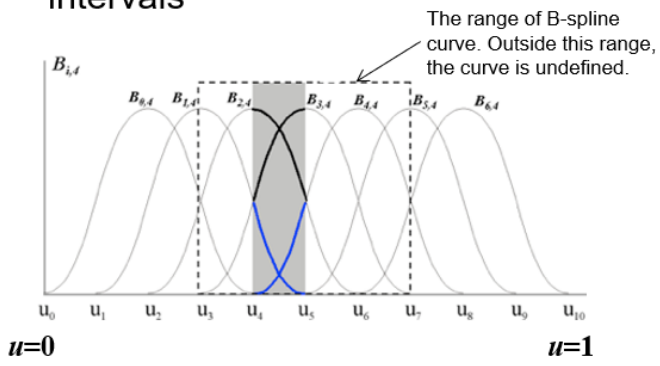
$$P(u) = \sum_{k=0}^{n} P_k B_{k,d}(u)$$

  - ***k*** is the index of knot (remember that we don't a copy of B-spline at the last 4 knots, so *n* should be 4 less than the total number of knots of a curve)
  - $\boldsymbol{P_k}$ is the weight/control point at knot k (***n+1*** control points in total)
  - $\boldsymbol{B_{k,d}}$ is the B-spline at knot k (***n+1*** copies in total)
  - ***d*** is the order of polynomials (for cubic B-spline d=4)

Arrangement of Basis Functions

B-spline curves

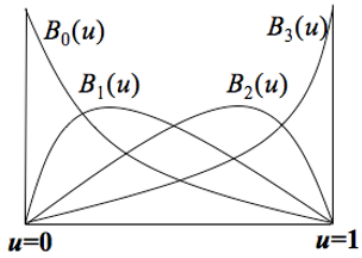
- The basis functions are spread out at ***even*** or ***uneven*** intervals



Every point on a B-spline curve is the result of blending 4 B-spline basis functions

Bezier curves

- The 4 basis functions are fitted into a **single** interval



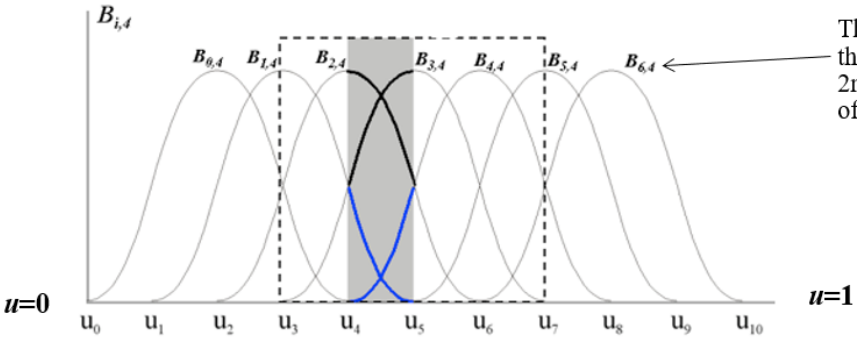
Every point on a Bezier curve is the result of blending 4 Bernstein functions

Uniform and Non-uniform Intervals

- If the intervals of parameter *u* (i.e., knots) are uniform (of equal length), the spline curves formed are **uniform B-spine**
- Otherwise, the curves are **non-uniform**

Construction of Uniform B-Splines

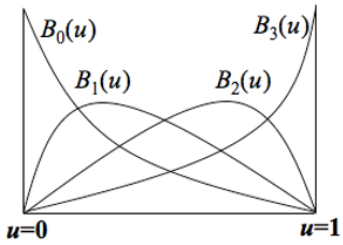
- A B-spline curve can be constructed by:
  - dividing the range of parameter *u* (usually 0~1) into evenly distributed knots (e.g., 10 equal intervals), so it has a uniform knot vector (e.g., 11 knots),
  - putting a copy of the basis function at each knot,
  - assigning a weight (control point) to each basis function to control its contribution in the blended curves sections.

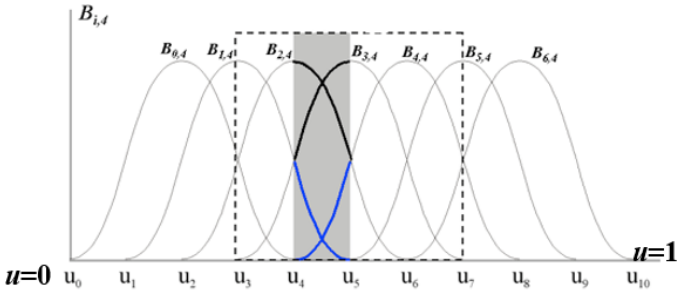


The knot vector of the spline is [0.0, 0.1, 0.2, . . . 0.9, 1.0]

Uniform B-splines v.s. Bezier Curve

- Compare:
  - Basis functions,
  - Ranges of support of basis functions,
  - Number of control points,
  - Local shape control,
  - Continuity.

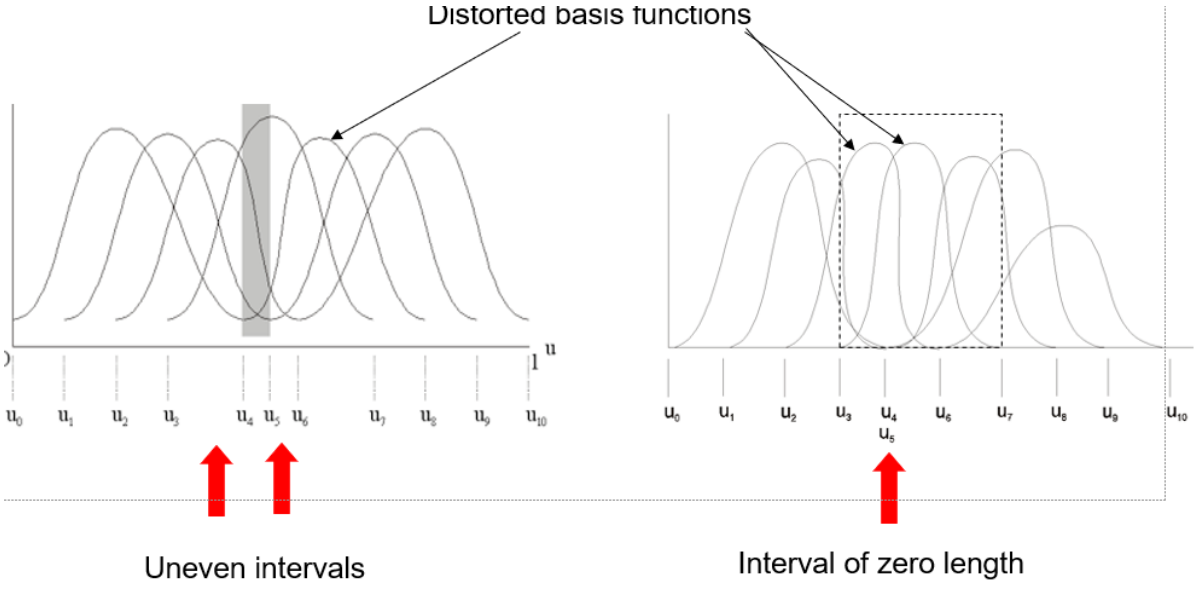




- Compare:
  - Basis functions: many v.s. 4
  - Ranges of support of basis functions: 4 subinterval v.s. entire parameter range
  - Number of control points: many v.s. 4
  - Local control: changing one control point affect only the spline shape over 4 intervals v.s. changing one control point change the entire bezier curve.
  - Continuity: automatic v.s. manual

Non-Uniform B-Splines

- Non-uniform (uneven) knot intervals: some interval are short or even of zero length, which result in **distorted** basis functions



Non-Uniform Knots

- Non-uniform intervals are specified by non-uniform knots - knots that are NOT equally spaced.
  - e.g., [0, 0.1, 0.2, 0.25, 0.3, 0.5, 0.6 …1] specifies unequal interval lengths,
  - [0, 0.1, 0.2, 0.3, 0.3, 0.4, 0.5…] specifies a *zero* interval by using a knot (0.3) *twice*. In this case, we say the knot (0.3) have a multiplicity of 2.

Non-Uniform Rational B-Spline

- NURBS
- Rational spline is the ratio of two spline functions

Characteristics of NURBS

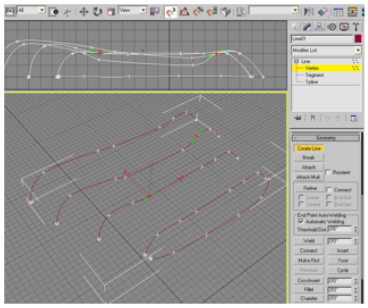
- NURBS have two important advantages over nonrational B-splines
  - They provide an exact representation of conics (straight lines, circles, ellipses, parabola, hyperbola).
  - They are invariant with respect to perspective transformation,
    - Nonrational B-splines do not possess this property

Spline Surfaces v.s. Polygons

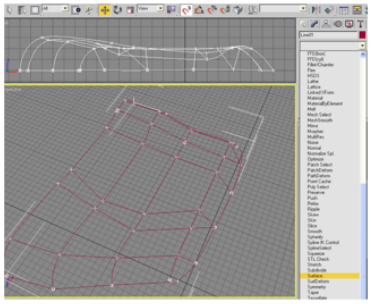
- Bezier curves, splines and NURBS are used to define smooth 2D or 3D shapes
- The surface created from splines are smooth, which are different from the surfaces defined using polygon meshes which are inherently rough and uneven.
- Spline surfaces are easier to edit. Editing a spline surface is done by manipulating a few control points, whereas editing a polygon mesh requires rearrangement of its vertices.
- Spline surfaces, NURBS surfaces in particular, demand for more computing power to create and render than polygon meshes.
- In standard graphics pipelines, spline surfaces are still rendered in the form of polygons. Ray tracing may render spline surfaces directly.

3DS: Modelling with Splines

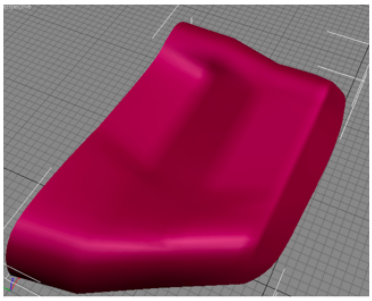
- Use spline to draw a spine "cage"



Draw the cross-sections



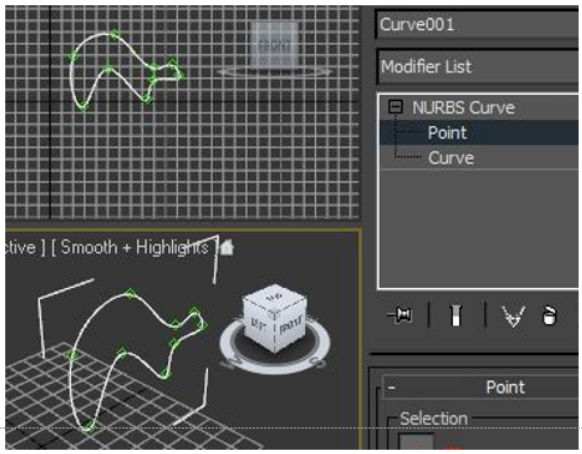
Weave the cross-sections with lateral curves

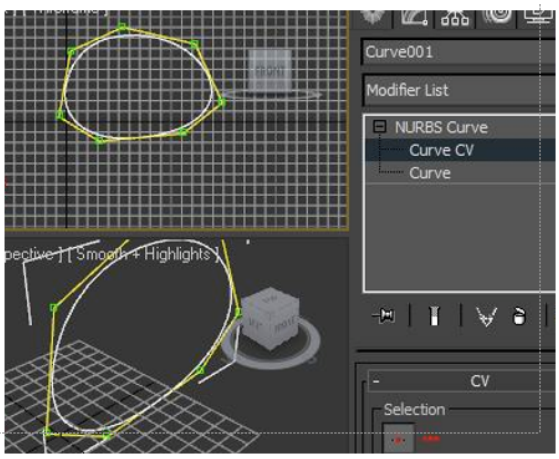


Loft surface

3DS: NURBS Curves

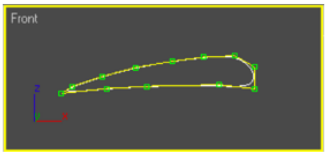
- Surfaces can be created from NURBS curves
- 3DS Max provide two types of NURBS curves:
  - Point curve – the control points on the curve, and
  - CV curve – control points are not on the curve.
- They are functionally the same.



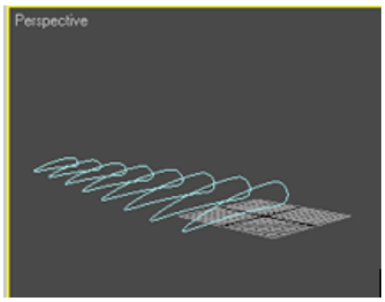


3DS: Modelling with NURBS

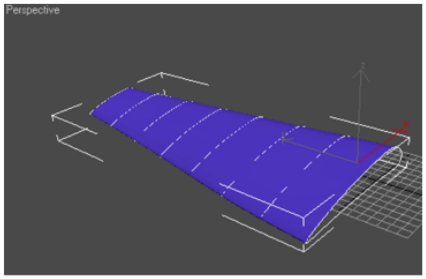
- Draw cross-sections and “loft” the surface



Draw a NURBS cross-section



Make a few copies and scale them down appropriately



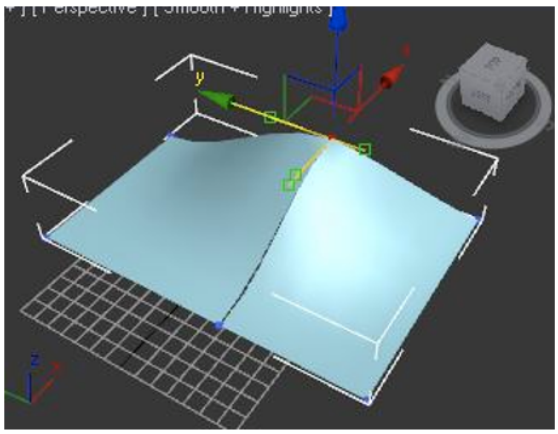
“loft” the NURBS to get a surface

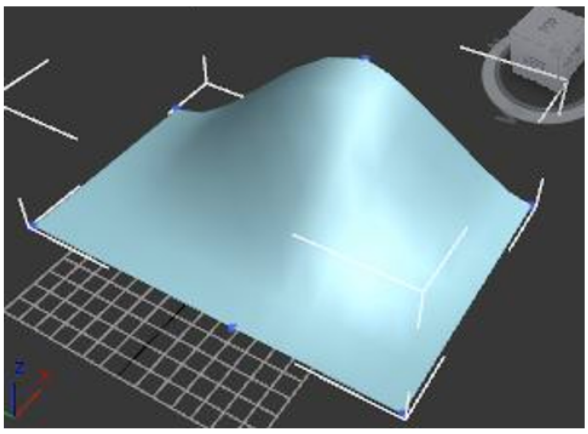
3DS: Surface Primitives

- 3DS provide both Bezier and NURBS surface primitives
  - Patch - the Bezier surface primitive
  - NURBS surface
- Modelling work can start from such surface primitives and new primitives can be “attached” to the current primitive as needed to extend the surface.

3DS: Patches

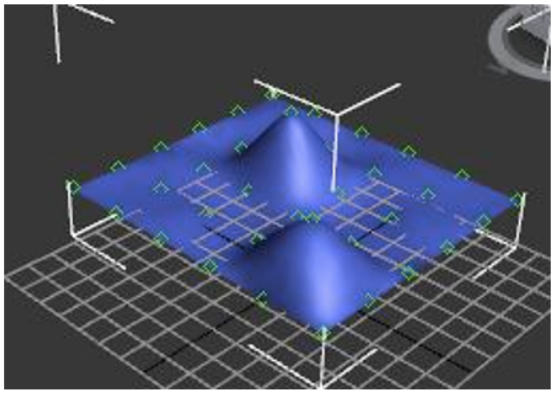
- You can edit a patch by manipulating its control points (you need to convert them into editable patch).
- Two patches can be combined to form a larger surface by “welding” their control points.



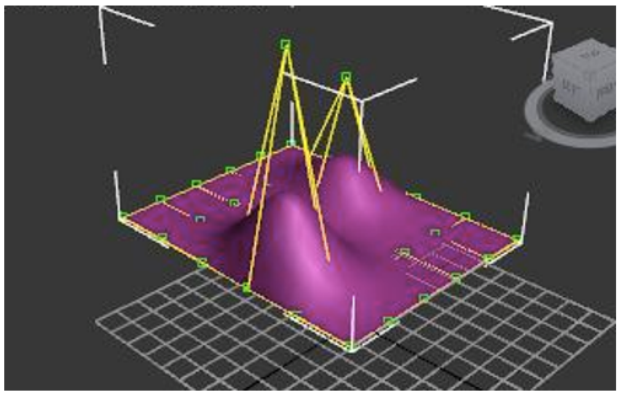


3DS: NURBS Surface

- Like NURBS curves, there are two versions of NURBS surfaces: point surface and CV surface.



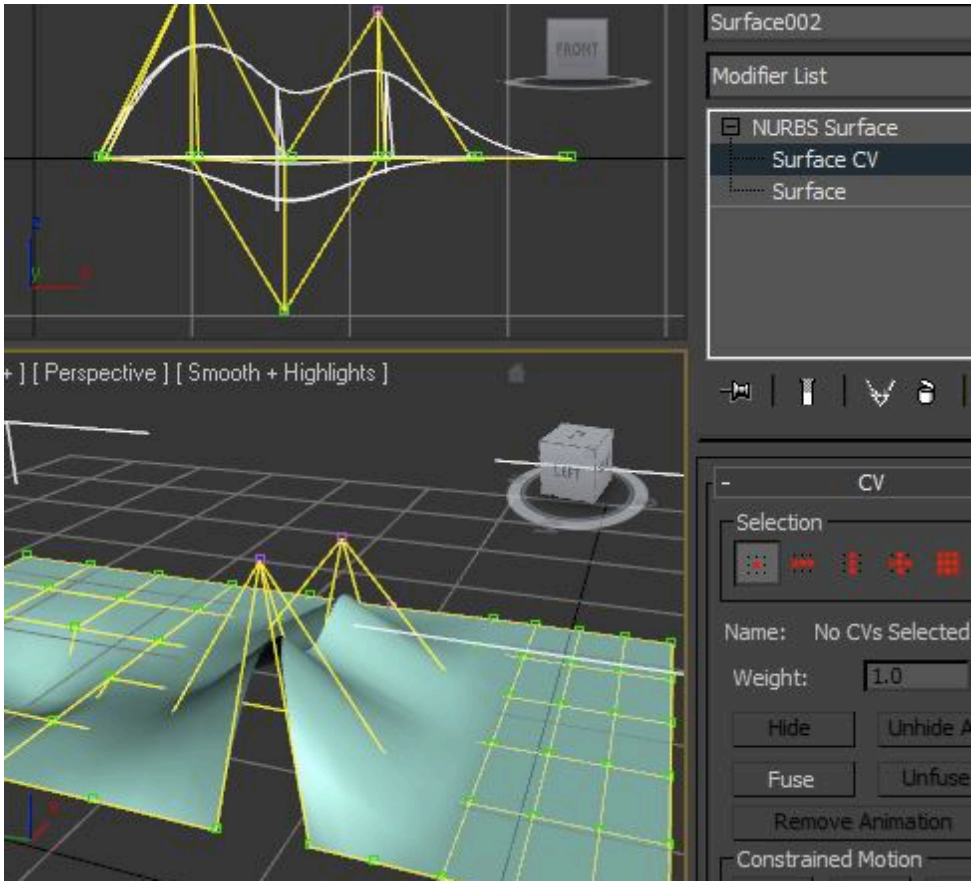
Point surface



CV surface

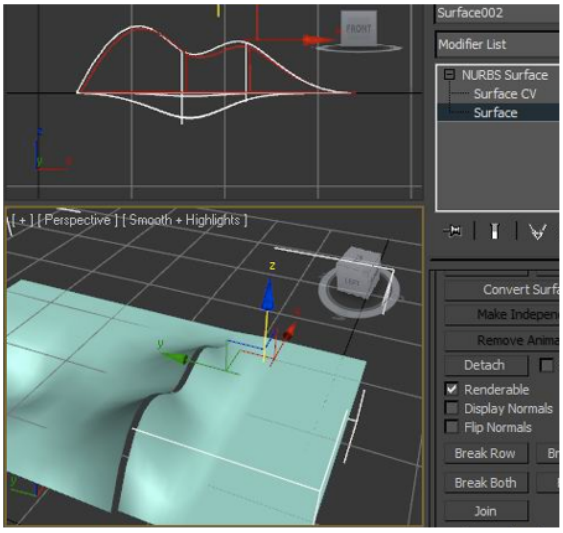
"Fuse" Surfaces

- Two NURBS surface can be combined by "fusing" or "joining" their control points.
- Fusing
  - After fusing, the entire surface consist of two "surface elements"
  - You may notice that a ridge of discontinuity is formed

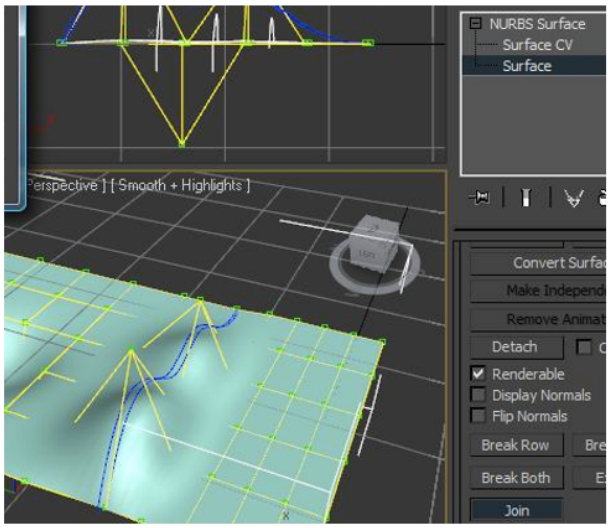


"Join" Surfaces

- To maintain the continuity between the two surfaces, we can "join" two surfaces.



Before joining



After joining

Lighting and Materials

Overview

- The shading problem.
- Light sources and their properties.

- Global lighting v.s. local lighting.
- Phong lighting/reflection model.

The Shading Problem

- Shading is the method/process of determining the shade/colour of a pixel.
- Shading is not a simple problem, at least in practice. Because how accurate/realistic the colours of pixels depends upon how well we simulate the interaction between light rays and surfaces of objects.
- As in many CG problems, we face a dilemma here:
  - High accuracy (or visual realism) demands for accurate models for the microstructure of surfaces and the interactions between lights (photons) and the microstructure.
  - However, accurate models of microstructures are impossible or not practical.
- Any solution for the shading problem must provide visually acceptable and computationally feasible approximation to the following properties or processes:
  - the properties of various light sources, e.g., sun light, light bulbs, etc.
  - the properties of surfaces of different material types,
  - the interactions between lights and the surfaces.

Lighting & Shading Models

- A few decades' research and hardware development have produces some acceptable solutions to the shading problem.
- The solutions normally consist of a lighting model (this lecture) and a shading model (next lecture).
- The lighting model describes how light rays interact with a tiny **flat** surface patch and determines the amount and direction of reflection.
  - The reflected light is what we see (therefore needs to be drawn/rendered). If a surface does not reflect light, it is simply invisible.
  - Various models have been proposed. Among them, the Phong lighting model (and the various variations) is mostly used in CG packages and applications
- The shading model/algorithm determines how we evaluate the lighting model on geometric primitives, e.g., a triangle or a polygon. For this, we have a few options:
  - Flat shading,
  - Gouraud shading, and
  - Phong shading.
- These shading algorithms produce good visual results under certain conditions and are widely used (More on them next lecture).

Light Sources

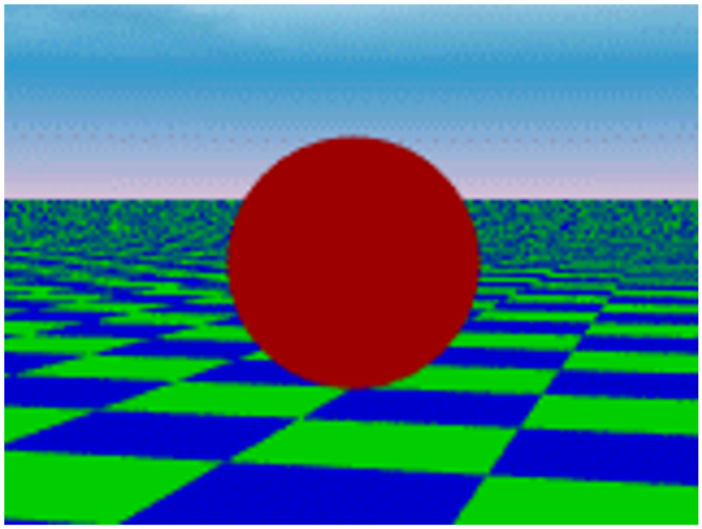
- Various light sources are used in CG to produce different lighting effects, e.g., sun light produce daylight effect, point light produces effect a lump, etc
- Most of the lights in CG have their physical counterparts, e.g., point light and a light bulb, but some do not, e.g., the ambient light.
- A light source has a number of properties:
  - Colour – spectral frequencies,
  - Intensity – radiance,
  - Geometry – shape (e.g., lamp shades), position and direction, and
  - Directional attenuation – intensity distribution.

Ambient Light Source

- Ambient light exists in almost every scene:
  - Objects not directly lit are typically still visible, e.g., the ceiling in this room, undersides of desks.
  - This is the result of light scattering – lights bouncing off intermediate surfaces.
- But ambient light is computationally too expensive to simulate.
- Therefore a *fictional* ambient light source is invented to account for the effect of scattered light in a scene:
  - No spatial or directional characteristics (therefore, no highlight)
  - Illuminates all surfaces with equal intensity, $I_{ambient}$

An Example

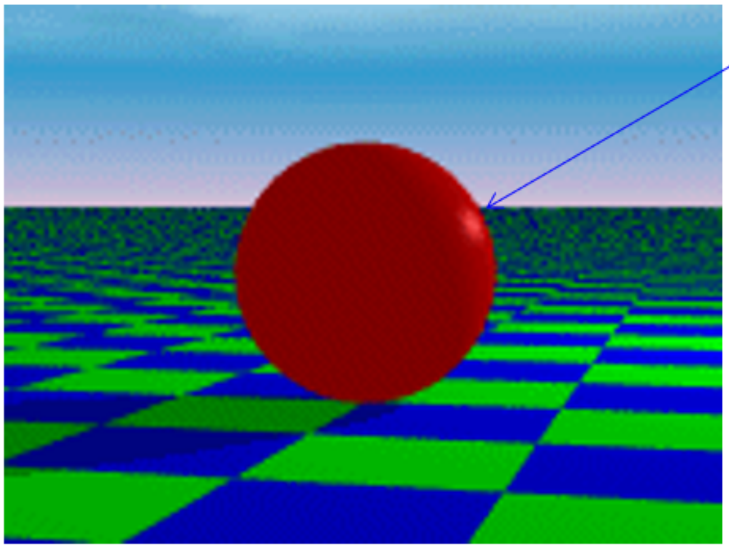
- A scene lit only with an ambient light source
  - Make the scene visible, but
  - Produce chalky scene and has no highlight



The default light in most CG software and APIs

Point Light Sources

- A point light source emits light equally (i.e., the same intensity) in all directions from a single point.
- It is characterised by:
  - A location/position,
  - An intensity (radiance),
  - Attenuation with distance (usually ignored)
- The directions (angles) of the light rays with respect to the surface normals vary.
- Light bulbs at some distance are typical point lights.
- An example of using an ambient and a point light source:

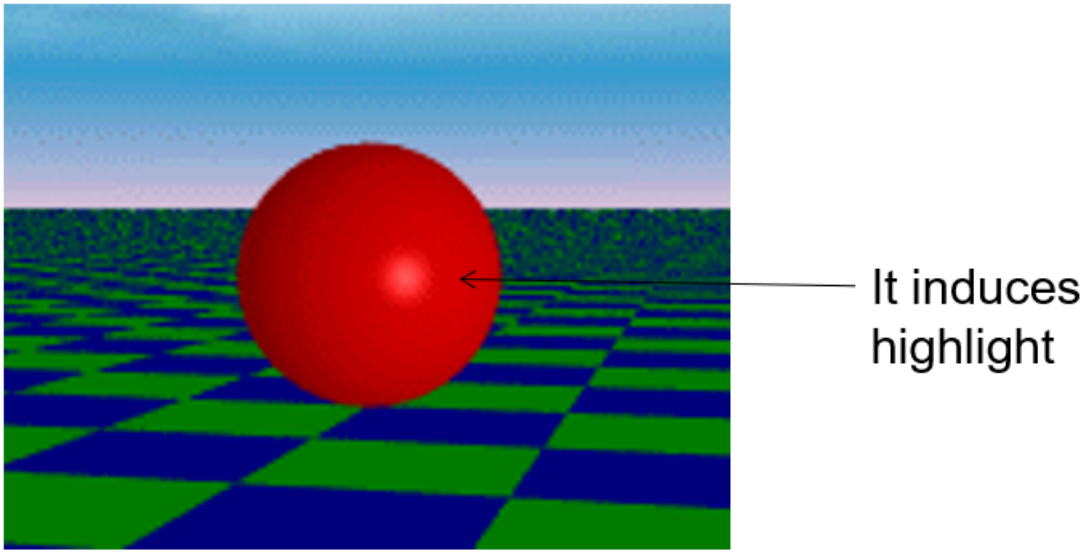


Because the light rays of a point light have directions, a point light causes highlight.

Directional Light Sources

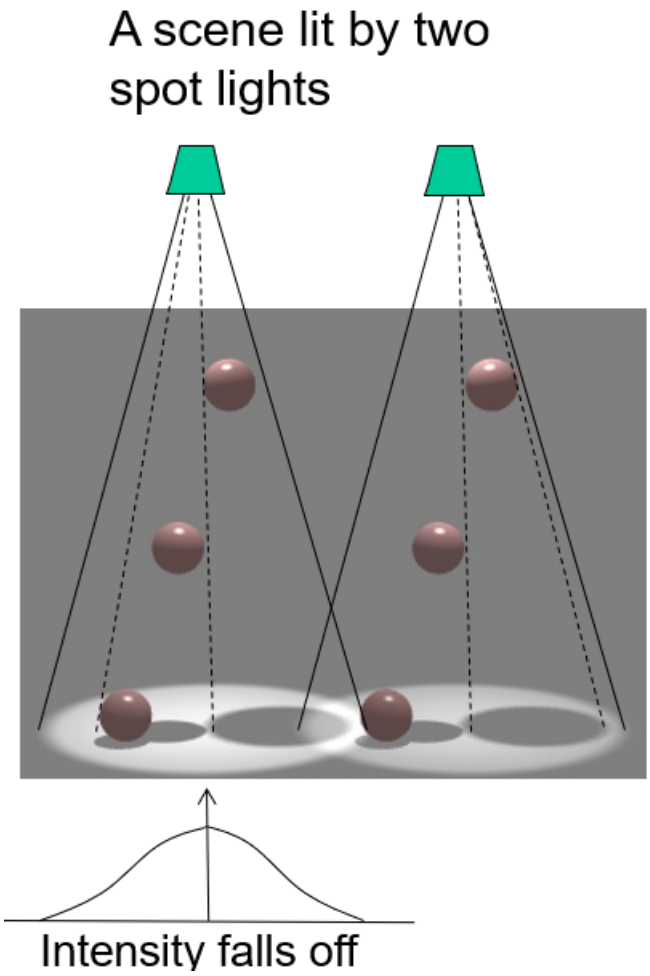
- For a directional light source, all light rays are parallel.

- It is characterised by
  - an intensity,
  - a direction (with respect to the surface normal),
  - Distance attenuation (usually ignored)
  - A point light **at infinity** (or far enough) could be regarded as a directional light source (e.g., sunlight) when depth of view is small.
- The same scene lit with a directional and an ambient light source.



Spotlight

- Spotlights are point lights whose intensity falls off directionally.
- It is characterised by a light cone that has
  - a position (of the apex of the cone),
  - a direction of the main axis,
  - a cutoff angle, α
  - a falloff function.



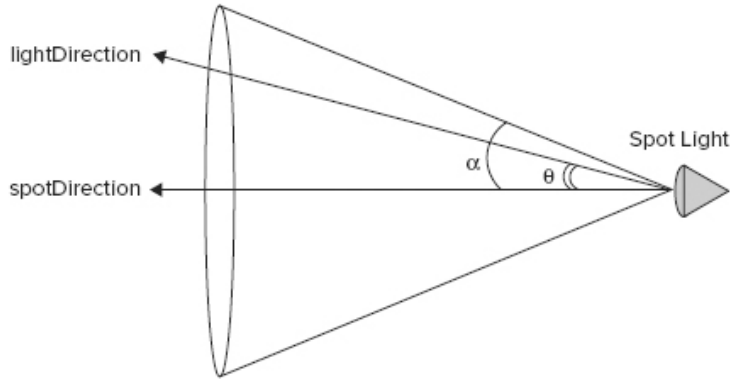
Falloff Function

- The *falloff function* of a spot light is used to describe the angular attenuation.
- Given a cutoff angle α, and the angle between the axis of the cone and the light ray, ϑ, the cutoff function is expressed as

$$I(\theta) = \cos^a(\theta)$$

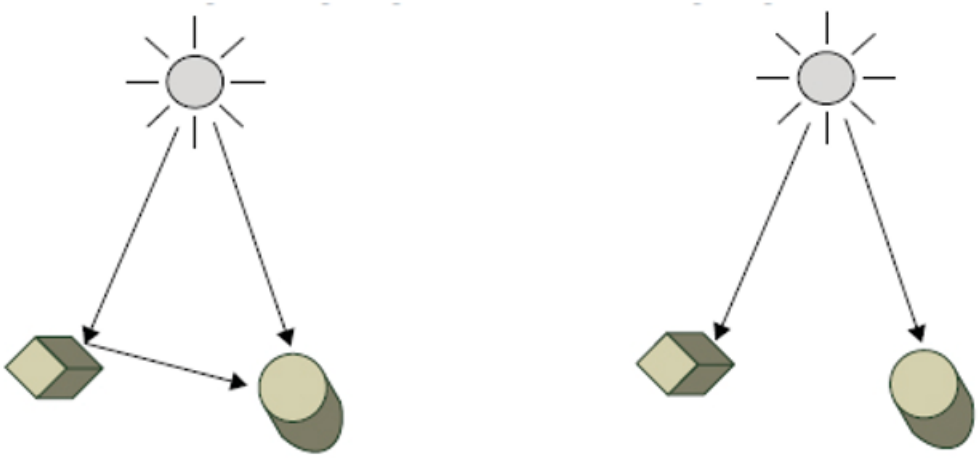
- where *a* is the *falloff exponent*. A large *a* makes the intensity decrease faster as the angle θ increases.
- If we use two *unit vectors spotDirection* and *lightDirection* to represent the directions of main axis and the light ray in question, we have

$$I(\theta) = (spotDirection \bullet lightDirection)^a$$



Lighting Methods

- Lighting is about how lights illuminate the scene objects.
- When simulating lighting in 3D graphics, one can choose to use one of two types of lighting:
  - Global lighting, or
  - Local lighting.



Global Lighting

- Global lighting takes into account the lights from the light sources **AND** the lights reflected from the objects in scene. E.g., a light source is reflected from an object; the reflected light will illuminate a second object; the light reflected from the second object then illuminates the 3rd object, and so on.
- An example of rendering (i.e., shade calculation) using global lighting is ray tracing.
- This technique tries to mimic the complicated interactions between light rays and the surfaces of objects.
- It can produce a highly realistic scene, but it requires a lot of computing resource to track down the paths of light rays, and therefore not very efficient. (It had been the default renderer in 3DS Max in the past, but Autodesk is no longer buying the license of Mentalray renderer)

Local Lighting

- In local lighting, only the lights that come directly from the light sources are accounted for in shading calculation.
- A defect of a *strict* local lighting model is that objects will not block light that hits them. This means that shadows are not automatically created in a local lighting model.
- Scanline renderer uses this approach. If you develop applications using APIs, this might be the only choice unless you develop your own rendering engine.
- In modern CG package, scanline renderer can produce shadows. In 3DS Max, you can enable shadow effect by changing the properties of light sources.

Reflection Models

- When considering lighting and shading issues, we usually focus on a **small and flat** patch of surfaces. The shapes of the surfaces are not considered (at this stage).
- The small patch is normally characterised by:
  - a location,
  - a direction (represented by its normal),
  - a reflectance spectrum (i.e., colour of the surface), and
  - a reflectivity.
- The reflectance spectrum and the reflective ability are determined by:
  - the atomic properties of the material (decides the colour), and
  - the micro-structures (e.g., matte or gloss) of surface (decide the reflectivity).
  - The true mechanism underlying light reflection is hard to model/describe.

BRDFs

- In some applications, an empirical functions called Bidirectional Reflectance Distribution Functions (BRDFs) is used for describing the reflectance spectrum and reflectivity of a material.
  - The BRDF of a material may be based on the results of numerous experiments.
  - Finding a good BRDF for a material, e.g., human skin, is difficult. Some research still need to be done.
- In practice, BRDFs are usually simplified by simpler reflection functions in which
  - the spectral properties of a material are simplified by assigning a colour to the material, and
  - the reflectivity is replaced by a (simpler) relationship between incident intensity (irradiance) and reflective intensity (radiance).

Phong Lighting

- The various reflection functions used in CG are called lighting models (or reflection models).

- Phong Lighting model (published in 1973) is one of them and is, arguably, the most popular for its simplicity and good visual effect.
- Other lighting models are similar and can be seen as variations of Phong lighting model.
- Phong lighting model breaks the reflection from a tiny flat surface into three parts :

  Total reflection =

  ***ambien**t reflection + **diffuse** reflection + **specular** reflection.*
- By this model, different materials have different combinations of these three parts.
- There is not much theoretical basis for this breakdown, but it produces acceptable results!
- To calculate each of the three reflection components, the model assumes that a light source has three corresponding components:
  - ambient light, I_a
  - diffuse light, I_d and
  - specular light, I_s
- It also assumes that a surface has different reflectivity corresponding to the types of lights:
  - the ambient reflectivity, k_a ,
  - the diffuse reflectivity, k_d,
  - the specular reflectivity, k_s, and the shininess of a surface, α, which is larger for smooth, mirror-like surfaces.
- With these assumptions, calculation of reflection can be easily done.

Ambient Reflection

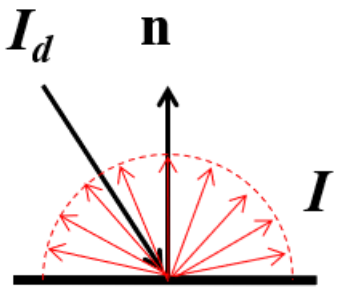
- Assume a global ambient illumination in the scene, I_a
- The ambient light reflected from a surface depends on
  - The surface properties, k_a
  - The intensity of the ambient light source (constant for all points on all surfaces).

The reflection function is linear (and simple): $I = I_a.k_a$

- Empirical, no theoretical basis whatsoever.

Diffuse Reflection

- Rough surfaces (at the microscopic level) reflect light in ALL directions. Typical surfaces of this category include chalk and matte surfaces.
- Because of the microscopic variations, an incoming ray of light would be reflected with equal intensity in any direction over the lit hemisphere.



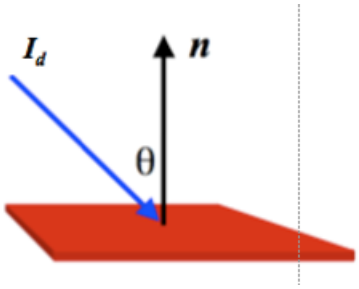
Lambertian Surfaces

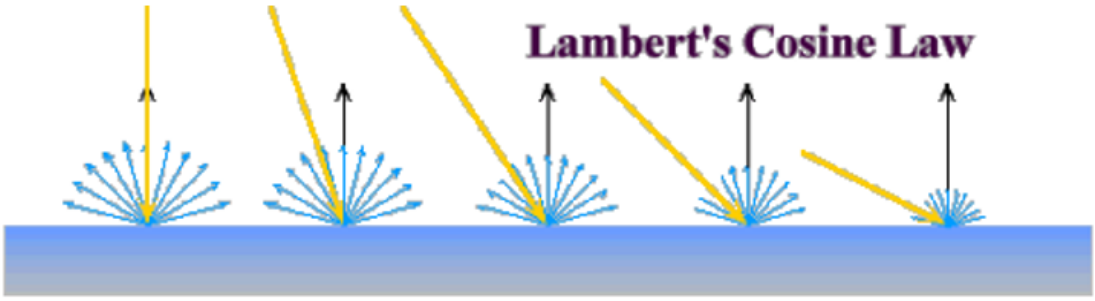
- Ideal diffuse surfaces are called Lambertian surfaces (chalk and most matte surfaces are very close to ideal diffuse surfaces).
- The reflection from a diffuse surface is calculated according to the *Lambert's cosine law*:

$$I = I_d k_d \cos\theta$$

where

  - k_d is the diffuse reflectivity of the material
  - I_d is the intensity of the incident light
  - θ is the angle between the surface normal **n** and the direction of incident of the light ray.

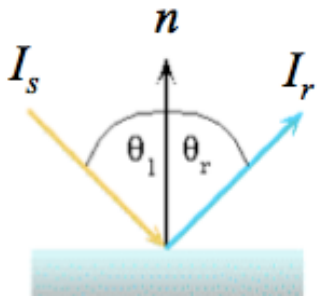




- Note that the intensity of reflection is **independent** of the viewing direction (because equal amount is reflected in all directions)

Specular Reflection

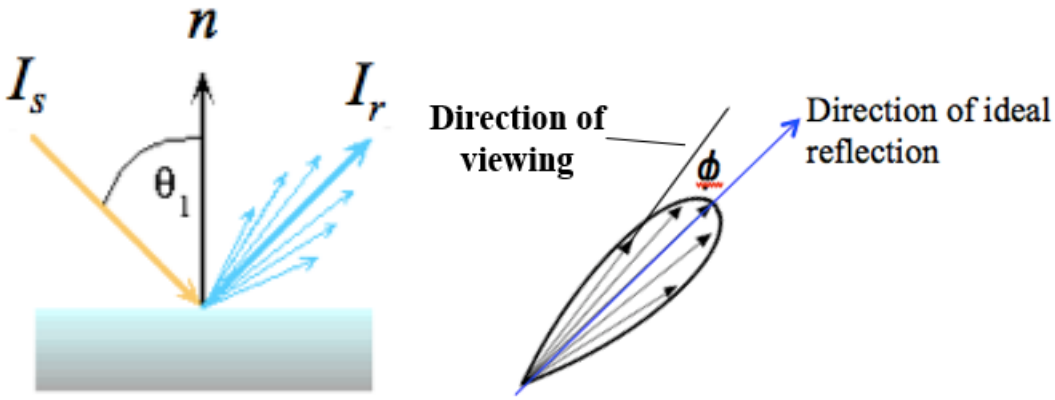
- Shiny surfaces, e.g.,, mirrors, polished metals, glossy car finish, etc., exhibit specular reflection.
- Light rays cast on a specular surface cause a bright spot known as a specular highlight.
- For ideal specular surfaces, specular reflection has these properties:
  - highlight appears as the colour of the light, NOT the colour of the surface,
  - highlight appears in the direction of ideal reflection, which is decided by the incident direction (For ideal specular surface, the direction of ideal reflection, θ_r, equals the incident angle, θ_i, as described by Snell's law),
  - highlight intensity equals the incident intensity, i.e., there is no energy loss in the reflection.



$$I_r = I_s$$

Non-Ideal Specular Surfaces

- Except for mirror-like surfaces, most real world surfaces are non-ideal, so the highlight appears softer and less defined (i.e., its appearance/visibility is not restricted to the direction of ideal reflection).



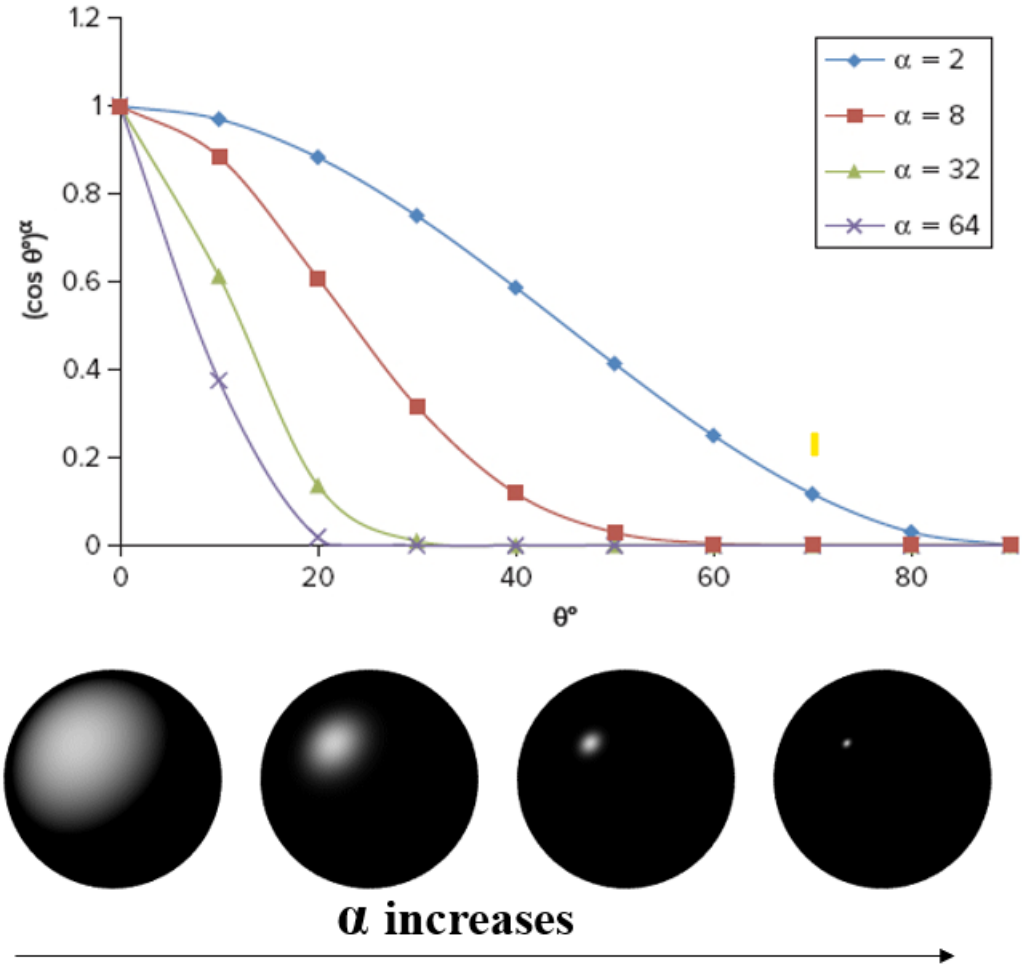
- Experiments had shown that most of the reflected light will travel in direction of ideal reflection, but some will go in the directions that are slightly ***off*** the direction of ideal direction. The intensity of specular reflection varies – as the off-angle $\boldsymbol{\varphi}$ increases, the intensity drops.
- Such properties of non-ideal specular surfaces can be captured by the formula:

$$I_r = k_s I_s (\cos\phi)^{\alpha}$$

where

- k_s is the specular reflectivity of the material,
- I_s is the intensity of the incident light,
- $\boldsymbol{\varphi}$ is the angle between the direction of ideal reflection and the direction of viewing,
- α characterises how fast the reflection intensity drops as $\boldsymbol{\varphi}$ increases and is used to indicate the shininess of a surface (ie., bigger $\boldsymbol{\varphi}$ represents smoother surfaces).

Shininess



Put All Together

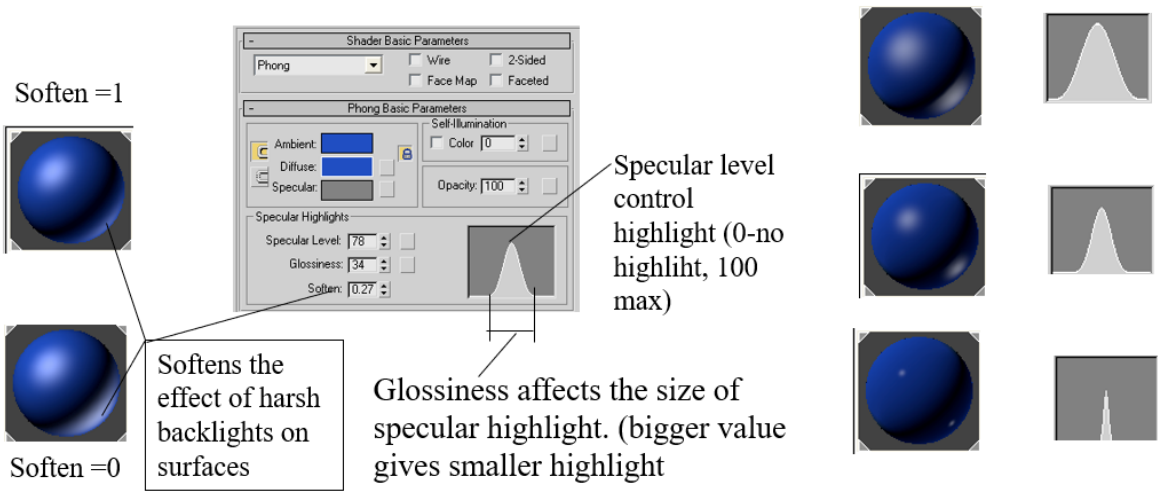
- Put all three components of reflection together, we have the formula for the Phong Lighting Model

$$I_{total} = k_a I_a + \sum_{i=1}^{all\_lights} \left(I_{i_d} k_d \cos\theta + I_{i_s} k_s (\cos\phi)^{\alpha} \right)$$

Ambient term **Diffuse term** **Specular term**

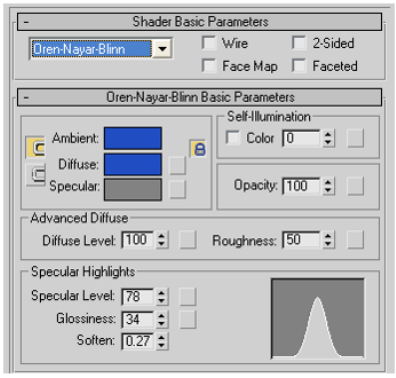
3DS Max: Material Design

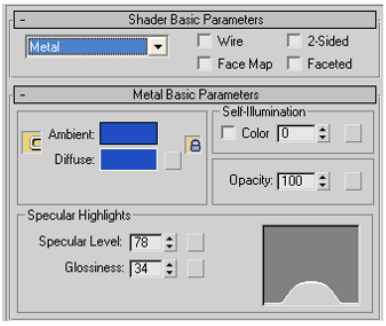
- Design a material in 3DS Max (or in CG in general) is basically a matter of determining the coefficients: k_a, k_d, k_s
  - Choose a material slot and select "Standard" for material type

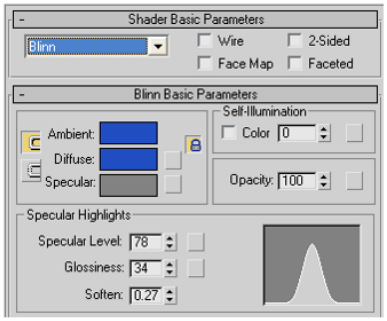


Other Lighting Models

- There exist many other lighting models that produce better results for special materials, but they work basically on the same principle of the Phong lighting model.







Shading

Introduction

- The realism of a rendering depends on
  - Geometric models (polygon mesh, NURBS surface, and etc),
  - Local v.s global lighting,
  - Lighting model that simulates
    - how light interacts with a surface,
    - surface properties (materials),
    - light sources (directional, point, etc).
  - Shading models (this lecture).

Recap: Phong Lighting Model

- Phong lighting (or reflection) model states that the light reflected from a point on a surface consist of three components:
  - Ambient reflection
    - Independent of viewing direction.
  - Diffuse reflection
    - Dependent on the incident angle (the angle between surface normal and the direction of light) but not on the viewing direction.
  - Specular reflection
    - Dependent on light source, incident angle and viewing direction.

Reflection Calculation

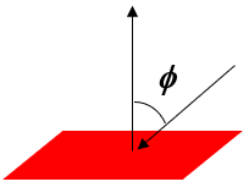
Total reflection =

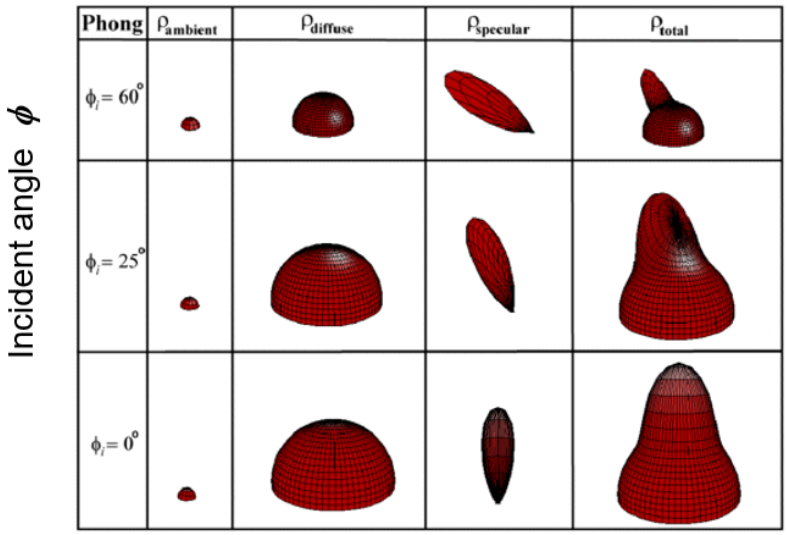
***ambient** reflection+**diffuse** reflection+**specular** reflection.*

$$I_{total} = k_a I_a + \sum_{i=1}^{all\_lights} \left(I_{i_d} k_d \cos\theta + I_{i_s} k_s (\cos\phi)^{\alpha} \right)$$

Ambient term — **Diffuse term** — **Specular term**

Visualisation of Lighting Model





- For matte materials, the diffuse reflection dominates. Little or no specular reflection will be found. However, for mirror surfaces, the reflection is mainly specular.

Shade Calculation

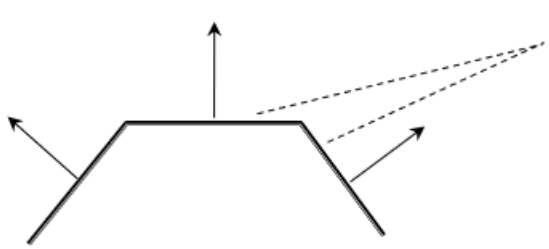
- In principle, by using a lighting model (e.g., Phong lighting model), one can calculate the colour of any point on a surface and therefore determine the colours for all the pixels representing the entire surface.
- However, in practice, such a strategy does not work well or unnecessary when computers are not powerful enough and real time performance is more important then realism.
- Different shade calculation (interpolation) methods have been developed. We call them shading models.

Shading Models

- In the last lecture, we mentioned that when determining the shades for the pixels of a flat patch(e.g., a triangle), we can use different shade-calculation methods:
  - Flat shading: treating the patch as a single point and evaluating the lighting model once to have a single colour for all the pixels of the entire patch.
  - Gouraud shading: applying the lighting model to the vertices of the patch to produce the vertex shades and then determining the colours of other pixels from them.
  - Phong shading: applying the lighting model to every pixel of the patch to get the colour for each of them.

Flat Shading

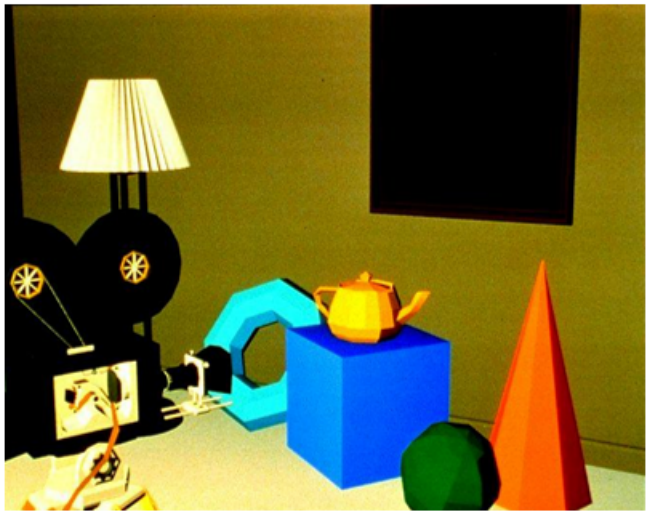
- Flat shading calculate a single shade for each polygon based on its surface normal.



- If a lighting model is chosen, e.g., Phong lighting model, we only need to evaluate the model once and assign the colour to all the pixels of the polygon.

$$I_{total} = k_a I_a + \sum_{i=1}^{all\_lights} \left(I_{i_d} k_d \cos\theta + I_{i_s} k_s (\cos\phi)^{\alpha} \right)$$

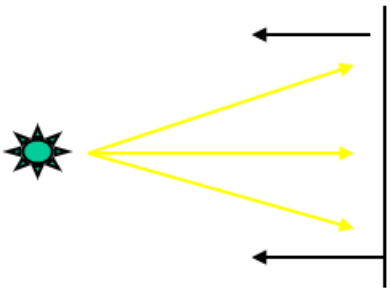
- Flat shading is very efficient.
- It works well under suitable conditions, for example, flat surfaces illuminated with directional lights and viewed at distance.

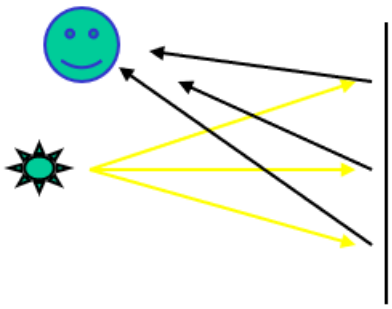


Result of Phong lighting + Flat Shading

Problems with Flat Shading

- In general, flat shading does not produce good visual realism even for objects that indeed have flat surfaces.
    - This is because that diffuse and specular reflections depend on incident directions of light and viewing.
    - For point light, the incident direction varies across the surface.
    - The viewing direction varies across the surface (if it is large).
- Result: Patchy surfaces with incorrect highlights.

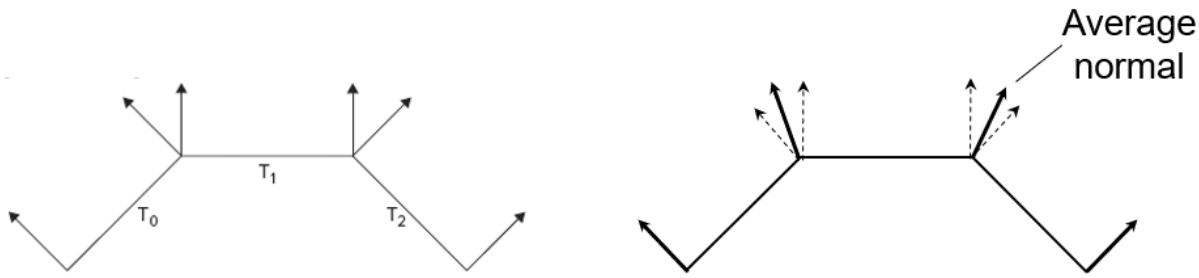




- When using flat shading, we have to notice that it only produce acceptable results under certain conditions:
  - When the light source is (infinitely) far away, so that the incident angle does not change across the surface.
  - When the viewer is far away, so that the viewing angles remain the same.
  - When the surface is indeed a flat surface (for example, one of the faces of a cube) and is not an approximation of a curved surface.
  - When the surface is matte (or less specular).
- It is used in applications where rendering efficiency is important.

Gouraud Shading

- The model (named after French computer scientist Henri Gouraud, 1971) calculates shades for the vertices of a polygon and then the vertex shades/colours are linearly interpolated over the polygon. It is also called per-vertex shading.
- This model uses the "average normals" of neighbouring polygons.

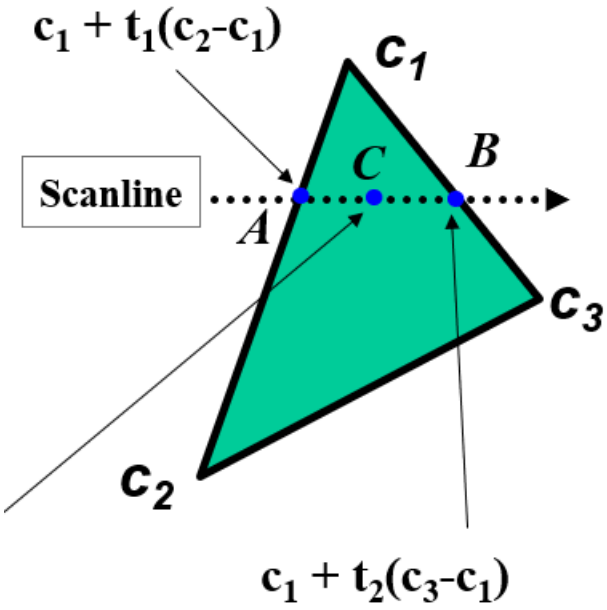


Why Average Normals

- The reason for using the "averaged normals" is obvious: if the polygons are approximating a curved surface (e.g., a sphere), the vertex normals should be orthogonal to the underlying curved surface instead of the flat polygons.
- In addition, the use of average normals gives a smoother transition of colour from one polygon to the adjacent polygons, and the seams between the polygons are less obvious.
- But the implementation of this model can vary: lighting calculation could be done on the vertext normals provided directly. If this is the case, to get a smooth shade transition across the edges, the normals should be carefully set.

Pixel Shade Interpolation

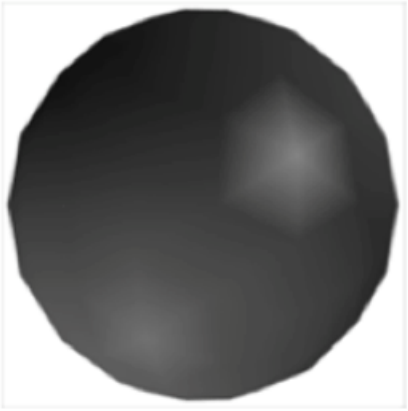
- The shades of other pixels of the polygon are calculated by linear interpolation of the vertex shades.
- Given the vertex colours, c_1, c_2 and c_3, to find the colour at a pixel ***C***, the interpolation is carried out in three steps:
  - First, two interpolations are done along two edges. This determines two colours for the two points on the edges along the scanline, i.e, ***A*** and ***B***
  - Then, interpolation is done along the scanline to find the shade for ***C***



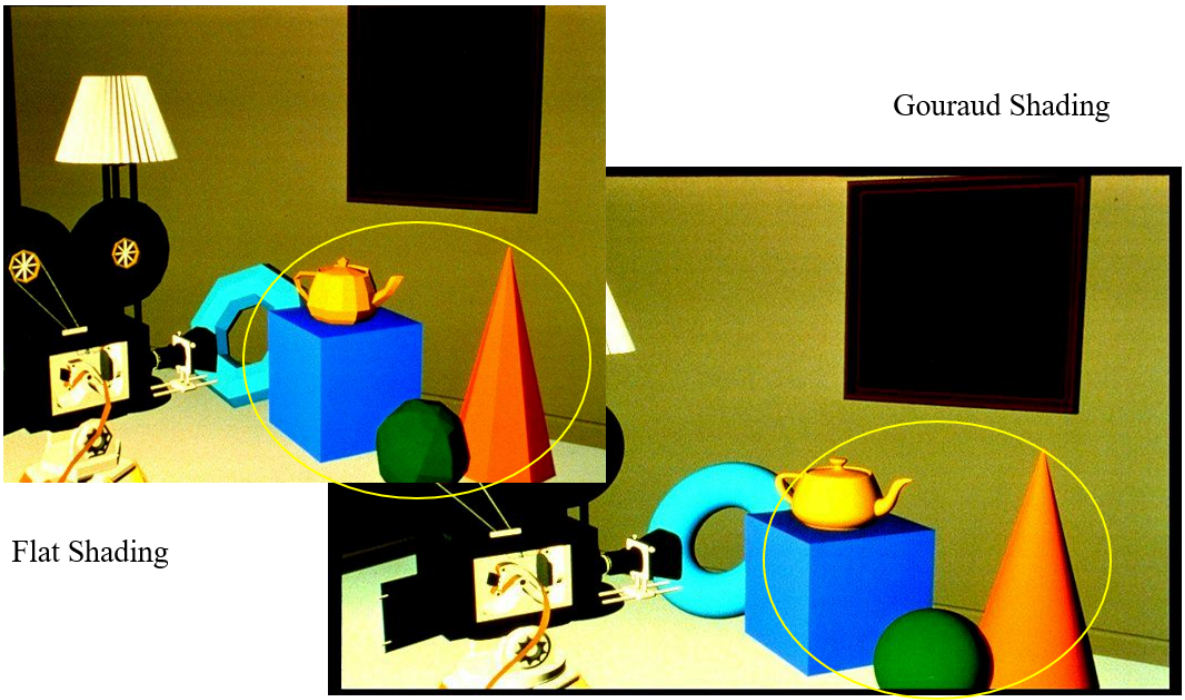
$$c=c_1 + t_1(c_2-c_1) + t_3(c_1 + t_2(c_3-c_1)- c_1 + t_1(c_2-c_1))$$

Gouraud Shading Evaluation

- In general, Gouraud shading produces a decent result for most matte surfaces.
- For shiny surfaces, the Gouraud shading will produce defects. Since the shading equation is only calculated at the vertices, a specular highlight that falls inside the triangle can be missed.



Gouraud shading applied to a sphere

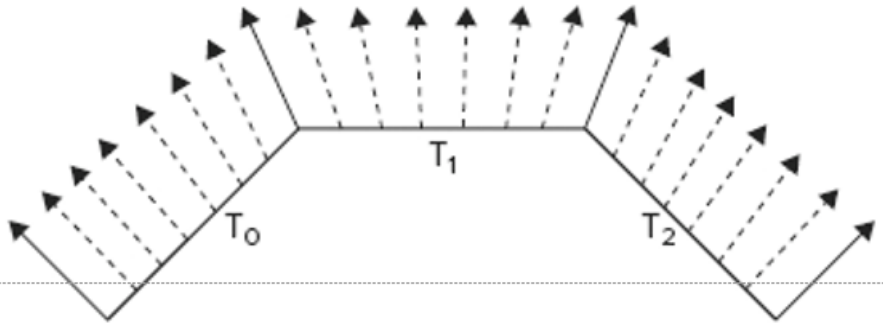


Gouraud Shading

Flat Shading

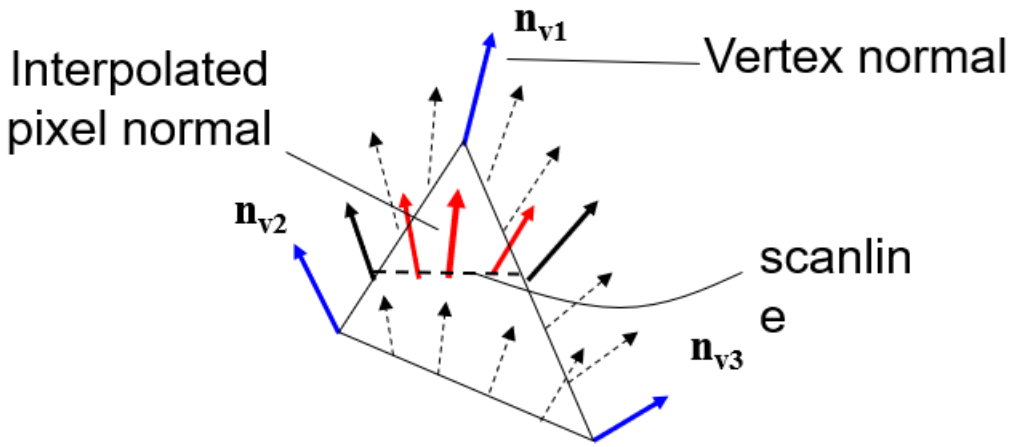
Phong Shading

- In Phong shading, each pixel/fragment has its own normal and the pixel normals are obtained by interpolating the “average” vertex normals.
- Phong shading interpolates vertex normals and evaluates the lighting equation for every pixel of a polygon. therefore it is also called per-fragment shading.
- It is computationally more costly than flat or Gouraud shading.



Normal Interpolation

- The normal at a pixel of a polygon is obtained by linear interpolation of the (average) vertex normals.

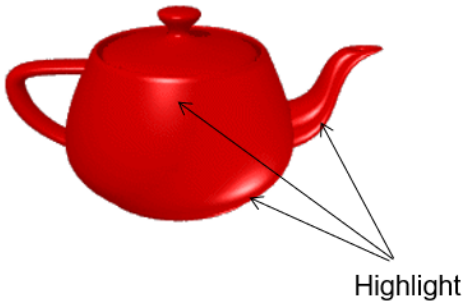


- Then the shade of the pixel is calculated using the normal and a lighting model (e.g., the Phong lighting model).

Phong Shading Evaluation

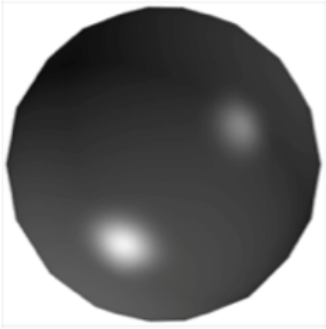
- Usually very smooth-looking results.
- Computationally much more expensive.

low polygon model
of a teapot

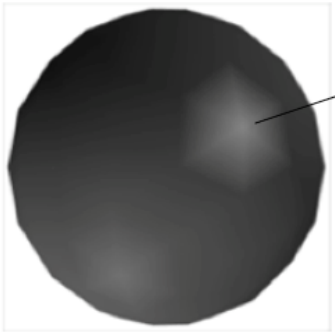


Rendering by Phong
shading

- Phong shading eliminates some defects that Gouraud shading cannot avoid:
  - produce well defined (not spreading) highlights,
  - produce specular highlight that falls within a polygon.



Phong Shading

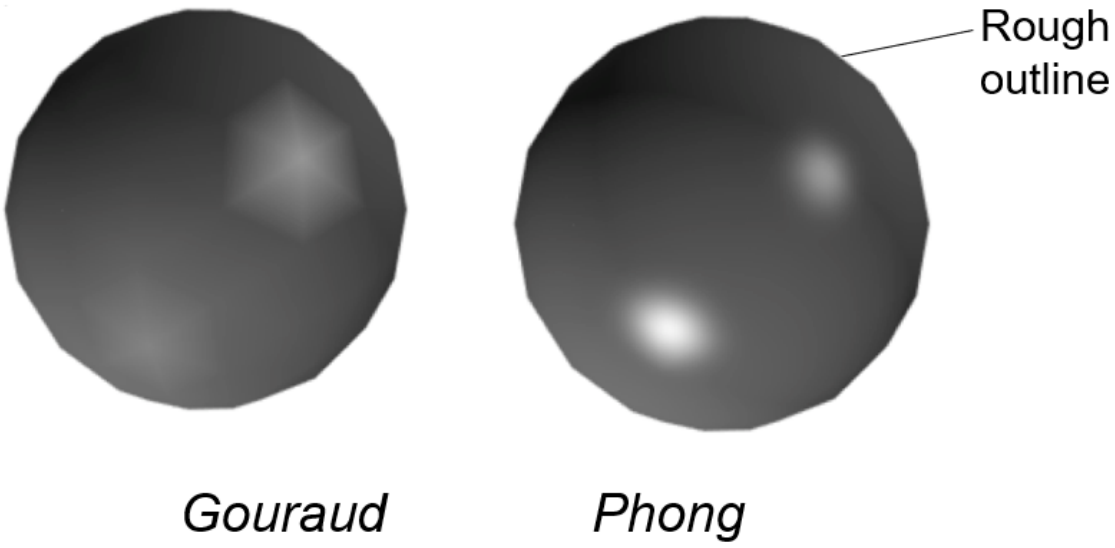


Gouraud Shading

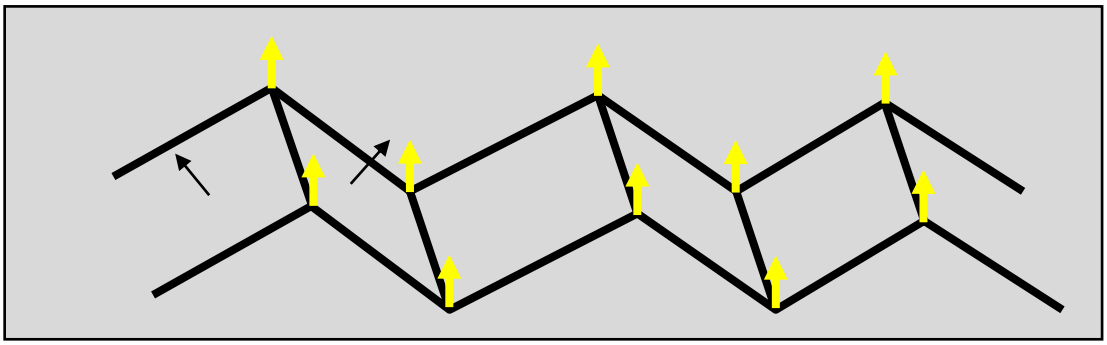
Notice that
highlights
are always
produced at
vertices

Limitations of Shading Models

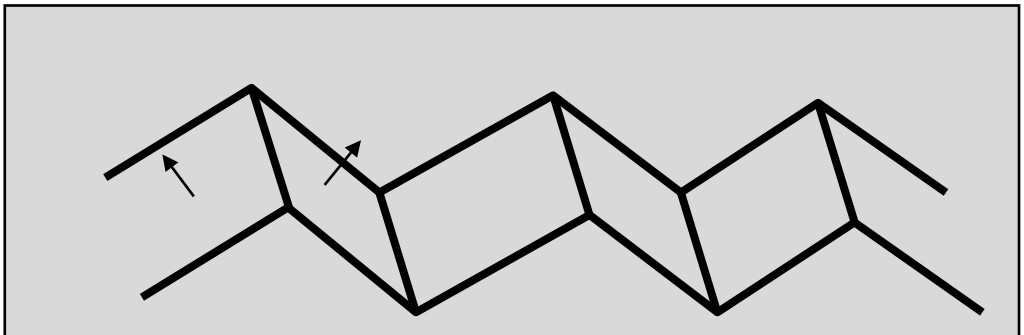
- Both Gouraud & Phong shadings use interpolation at certain stage, therefore they are called interpolated shading.
- These shading models cannot eliminate the polygonal (jagged) boundaries in silhouettes/outlines or shadows.



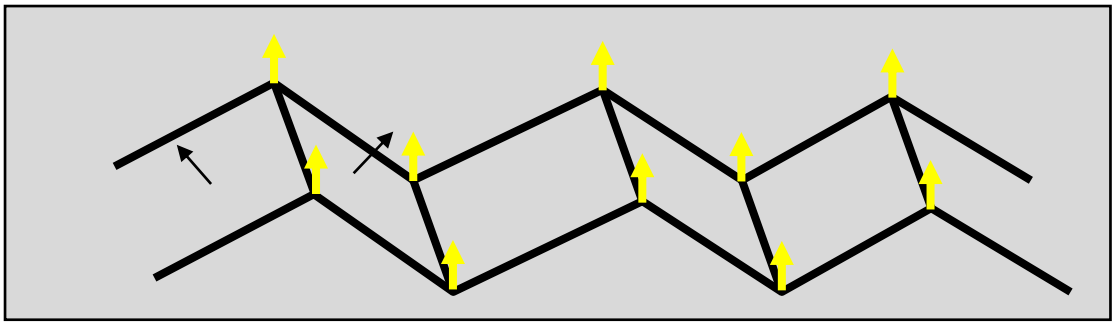
- Interpolated shading causes problems with some shapes.
- Consider the shading problem for following shape: the average normals (for Gouraud shading) and the interpolated normals (for Phong shading) are the same everywhere
- Both Gourand and Phong shading methods will produce the same colour for all the pixels – a strip of constant colour rather than a 3D surface.



- **Solution 1**: Use the surface normals – this is, in effect, the flat shading.
- By calculate the shades for each flat surface using its surface normal, we obtain alternating shades for the flat facets.

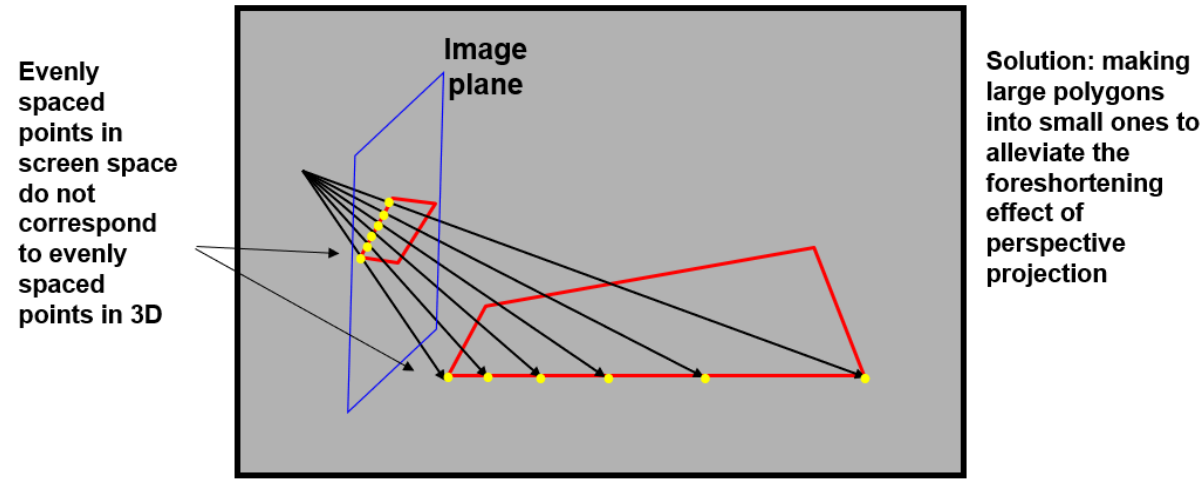


- **Solution 2**: Adding small polygon strips along edges and then use interpolated shading.



Perspective Distortion

- Linear interpolation in screen space does not corresponds to a linear interpolation in world space, i.e., the interpolation in world space is nonlinear.



Summary of Shading

- Flat (constant) shading
  - Approximation only holds for exceptional cases.
- Gouraud shading
  - Poorly interpolates specular highlights.
- Phong shading
  - produce better visual realism, but more expansive.
- Interpolation shading has its inherent problems.

Texture Mapping

Use Textures

- One of the dilemma in computer graphics is visual fidelity vs. rendering speed.
  - With refined geometric models, better visual fidelity will be achieved, but the rendering will take longer time,

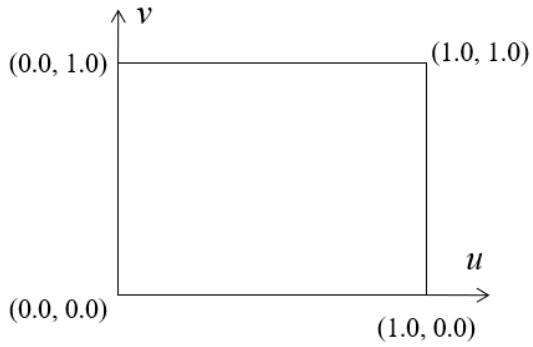
  - With coarse models, faster rendering is possible, but the quality will suffer.
- Another problem is some objects, especially the surface properties of the objects, are difficult to model
- Texture mapping is a techniques for solving the modelling and rendering problems of such objects.

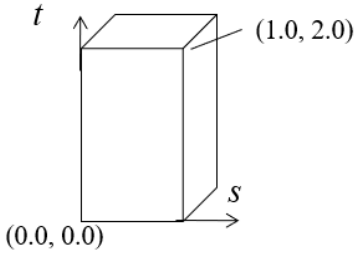
2D Texture

- The most basic form of texture mapping uses a single image as a texture – 2D texture.
- Regardless of what being used as textures, coordinate systems are needed to specify where texels should be sampled from the texture and where on the object they should be placed.
- Texels may be regarded as the pixels of a texture, but a texel may consists of many pixels of a texture image.
- To fetch a texel from the texture is called to sample a texture.

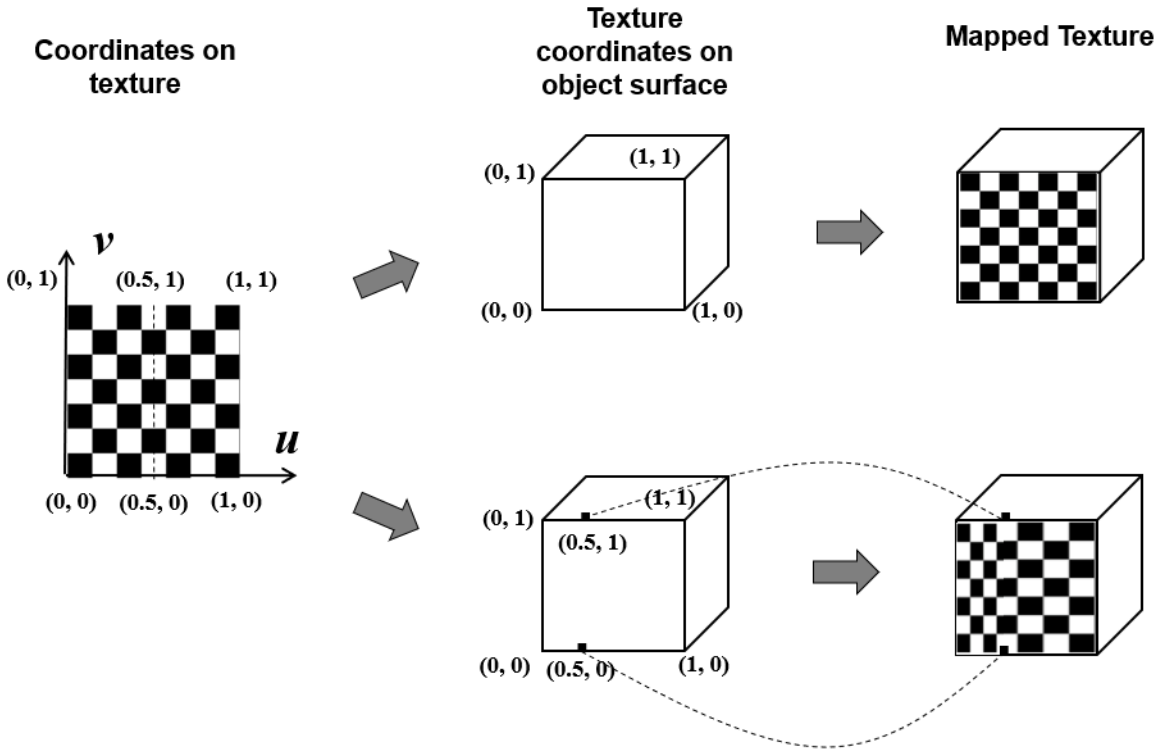
Texture Coordinates

- To map a texture to a surface, we need to establish the correspondence between the texels on the texture and the points (fragments) on the surface of a model.
- This is done by specifying texture coordinates on both the texture image and the object model.
- The texture coordinates for a texture are usually represented by a pair of values, (*u*, *v*), and called *u*, *v* coordinates.
- The texture coordinates for a model (it might be a surface of a single or a few polygons) is called *s, t* coordinates.
- Texture coordinates for a texture, (u, v), are normalised coordinates:
  - The lower-left corner of the texture is defined as the origin and has the coordinates (0.0, 0.0).
  - The upper-right corner of the texture always has the coordinates (1.0, 1.0), regardless of its size or whether or not the texture is a square.
- In contrast, the texture coordinates on a model, (s, t) could be greater than 1.0, e.g., (0.0, 2.0). In this case, texture will be repeated



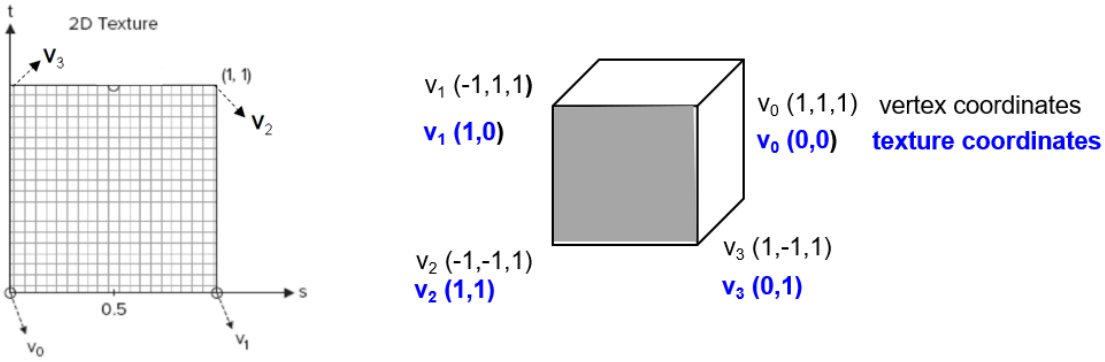


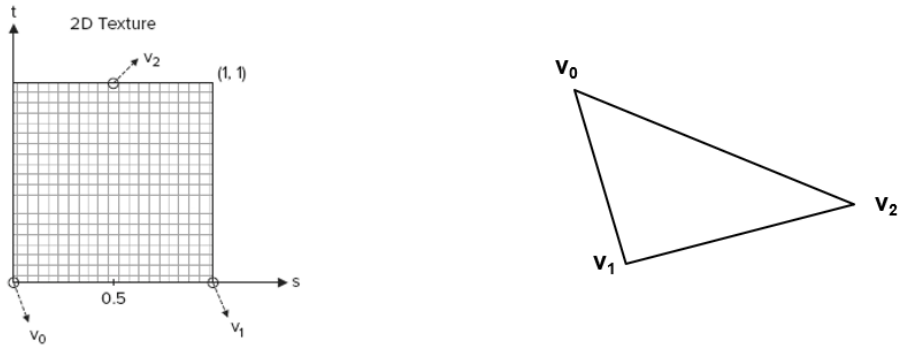
Map (u,v) to (s, t)



- The example shows clearly that the assignment of texture coordinates on the vertices affects how the texture being mapped.
- In general, it is difficult, or even impossible, to achieve uniform, distortion-free texture mapping on curved surfaces.
- Texture mapping algorithms that produce acceptable visual effect for some common shapes, such as triangle, cube, cylinder, sphere, etc, are available and implemented in graphics libraries and/or APIs.

Assign Texture Coordinates



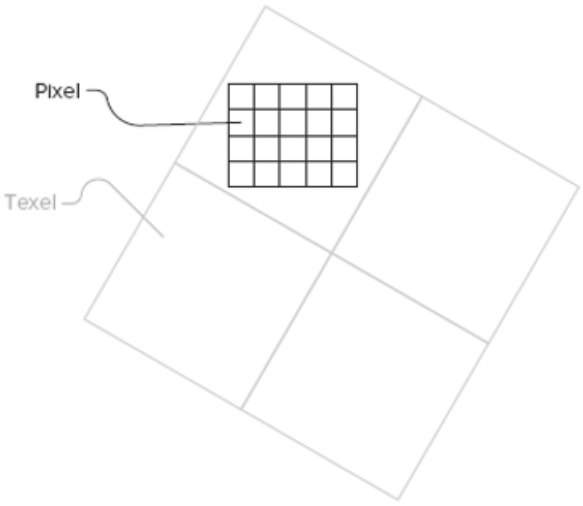


Texture Filtering

- Texture mapping produces good results only if the size and resolution of a texture matches the rendered image of the object.
- For example, a texture of the size 512 × 512 texels is mapped to a square surface (of any size). If the size of the rendered image of the square is close to 512 × 512 pixels, the texture will look accurate and natural. Otherwise some visual defects will appear.
- If the rendered image of a surface consists of more (less) pixels than the number of the texels on the texture image, the texture will need to be stretched (compressed) to properly cover the surface. This change to texture size is called magnification (minification).
- The process of calculating pixel colours from the texels of a magnified or minified texture is called texture filtering.
- The aim of texture filtering is to decide the “best” colour for a pixel in such cases.

Magnification

- In the case of texture magnification, the texture appears, visually, to have been magnified. That is, one texel covers (corresponds to) several rendered/on-screen pixels.

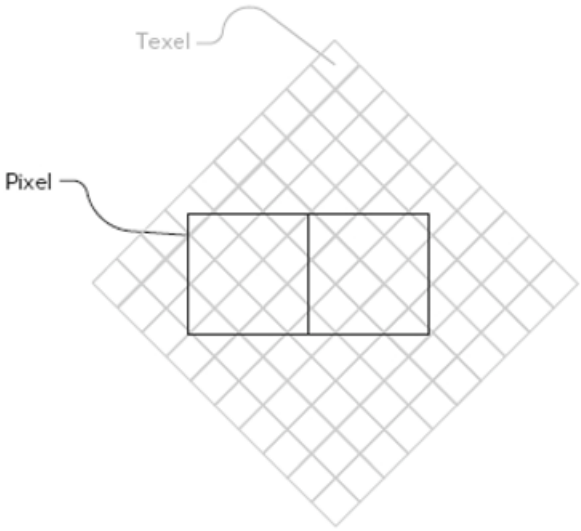


Filtering - Nearest or Linear

- The nearest filter takes the colour of the nearest texel to as the colour of the pixel.
  - This filtering is fast since only one texel is considered in colour calculation.
  - Many pixels fall upon one texel and they all take the same colour, which would result in a blocky appearance – an effect called pixelation.
- The linear filter assigns colour to a pixel by linear filtering or bilinear filtering - it takes a (distance-) weighted average of the four texels surrounding the texture coordinate (of a pixel).
  - Computationally more complex than nearest filtering,
  - Alleviates pixelation,
  - But results in blurred texture.

Minification

- In minification, several texels correspond to one pixel on the screen. This means that the colours of several texels would affect the color of one pixel.



Filtering - Nearest or Linear

- As in the case of magnification, the nearest neighbor filter fetches the colour from the texel that is closest to the current pixel (i.e., its texture coordinates).
- A defect of the nearest filtering is aliasing – the smooth curves, boundaries or lines become jagged.

Aliased and anti-aliased letters

- The linear filter uses the weighted average colour of the four closest texels as the color for a given pixel.
- The linear filter can give a slightly better result than the nearest filter, but the aliasing problem still exists.

Mipmapping

- As the distance between an object and the viewer/camera can change, we cannot decide beforehand when to apply magnification or minification filtering to reduce pixelation or aliasing.
- A better approach is to use a set of textures of different sizes and dynamically choose a texture size that can roughly maintain the one-to-one correspondence between the pixels and the texels.
- This techniques is called mipmapping – using a pre-calculated, optimised collections of images that accompany a main texture, intended to increase rendering speed and reduce aliasing or pixelation artifacts.
- The image collection used in mipmapping is called mipmap chain.

- A mipmap chain has images of several levels of resolutions. In practice, *at each level, the image size is half the size of the previous level*.
- The textures do not have to be square, but the chain continues until the last texture has the size 1 × 1, e.g., a mipmap chain could contain the textures of sizes 256 × 256, 128 × 128, 64 × 64, 32 × 32, 16 × 16, 8 × 8, 4 × 4, 2 × 2, and 1 × 1.

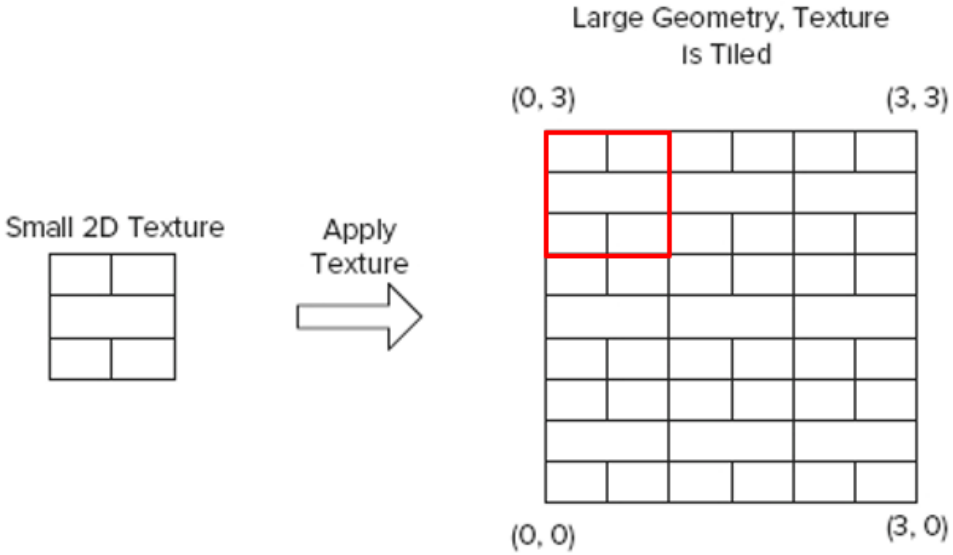
- A complete mipmap chain occupies approximately one-third more memory than without mipmapping.

Texture Wrapping

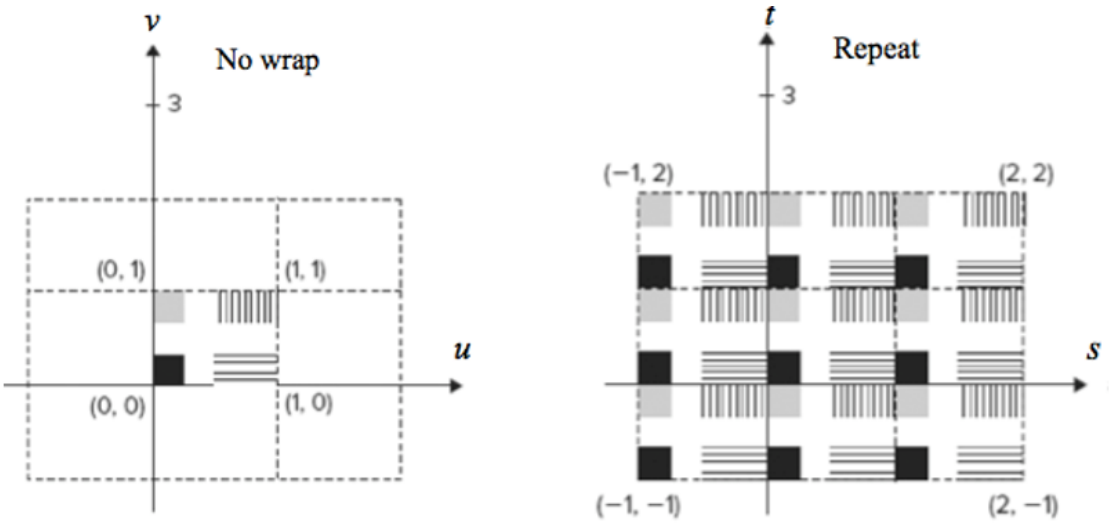
- As discussed previously, textures are described by a coordinate system: its lower-left corner has the coordinates (0, 0) and the upper-right corner has the coordinates (1, 1), regardless of its size or whether or not it is a square.
- What if we assign a pair of texture coordinates, (0.0, 2.0), at a vertex?
- This can be handled by texture wrapping. Texture wrapping can be done in the following ways:
  - repeat/tiled
  - mirrored repeat
- Note that independent, different wrap modes can be applied to s-direction and t-direction of the texture coordinates

Repeat

- E.g., according to the texture coordinates of the vertices provided, the original texture is repeated three times in both *s*- and *t*-directions.

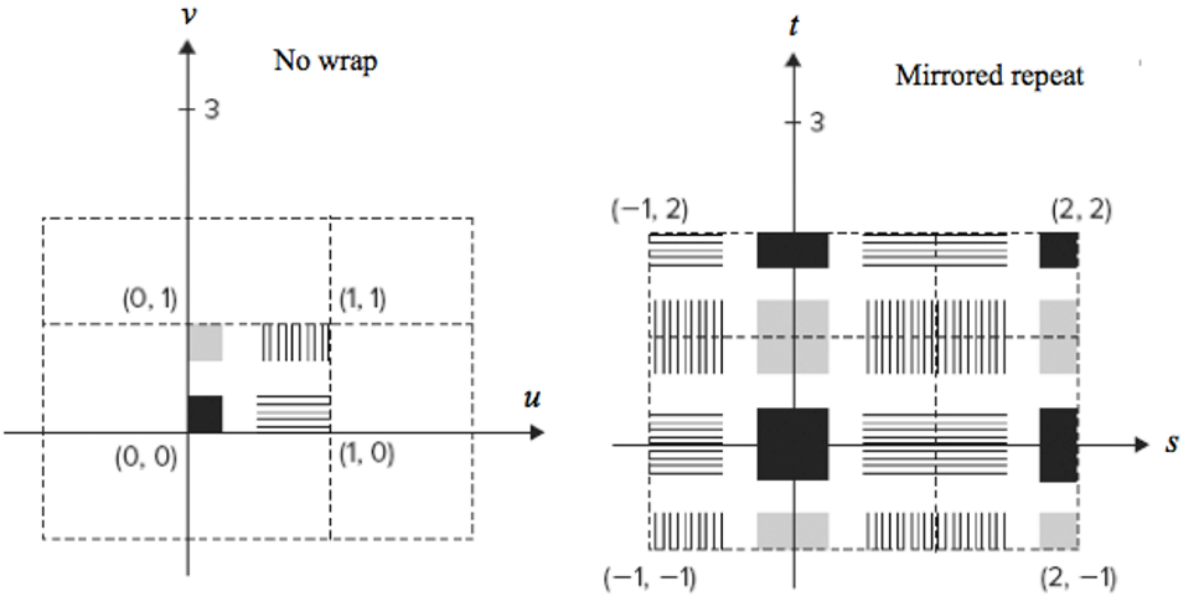


- Use negative coordinates (s, t)



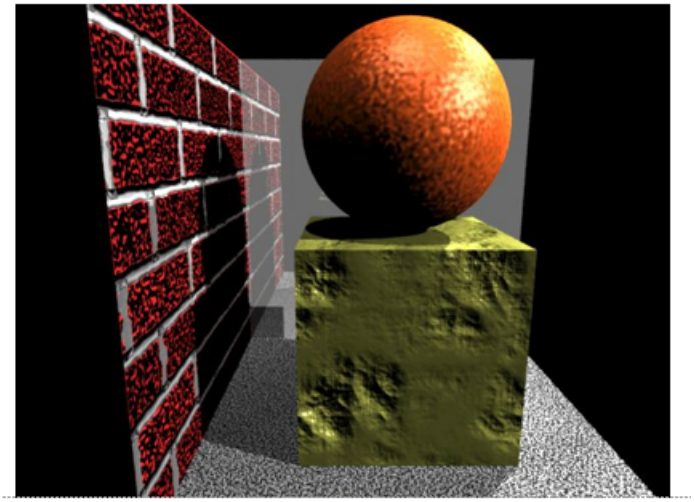
Mirrored Repeat

- In this mode, texture is mirrored while being repeated:



Bump Mapping

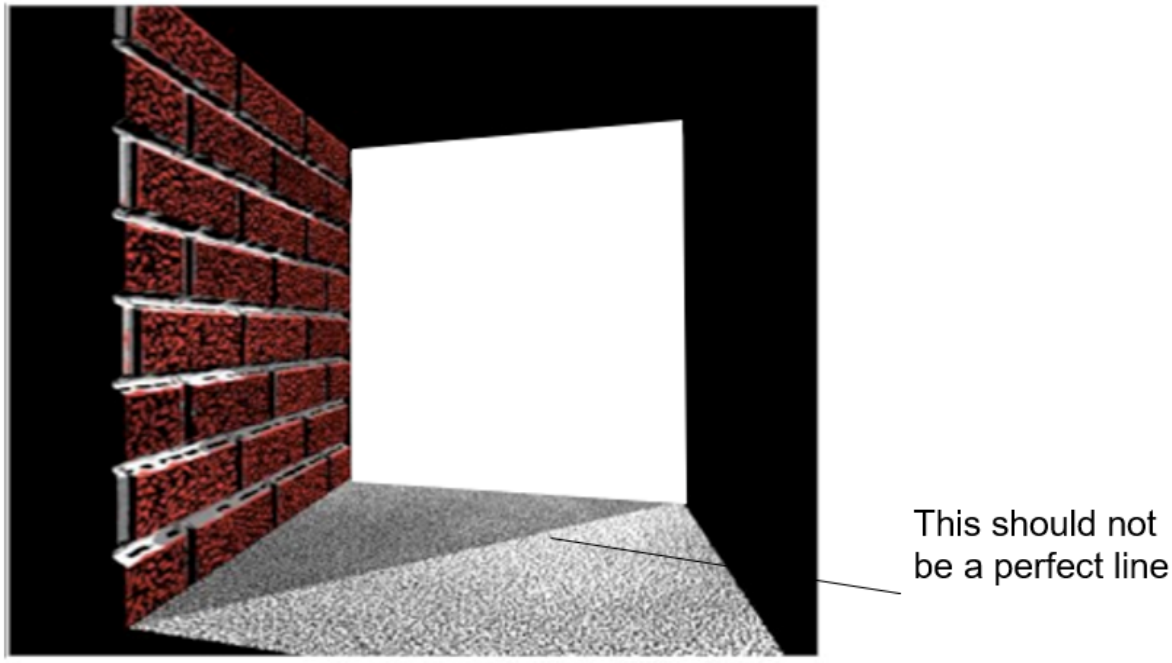
- Problems of texture mapping
  - Texture mapping does not produce uneven surfaces.
  - Texture mapping does not produce irregular boundaries of objects or their shadows



- A solution to these problems are to use textures to modify the surface geometry.
- The values of texel (e.g., intensity values of texels) can be used to modify surfaces in several ways:
  - Modify the surface normals of polygons, e.g., use the partial derivatives of bump map to modify the directions of surface normals. This is the bump mapping in its usual sense.
  - Control the height field of rough surfaces, e.g., to modify vertex coordinates – this is also called displacement mapping

Displacement Mapping

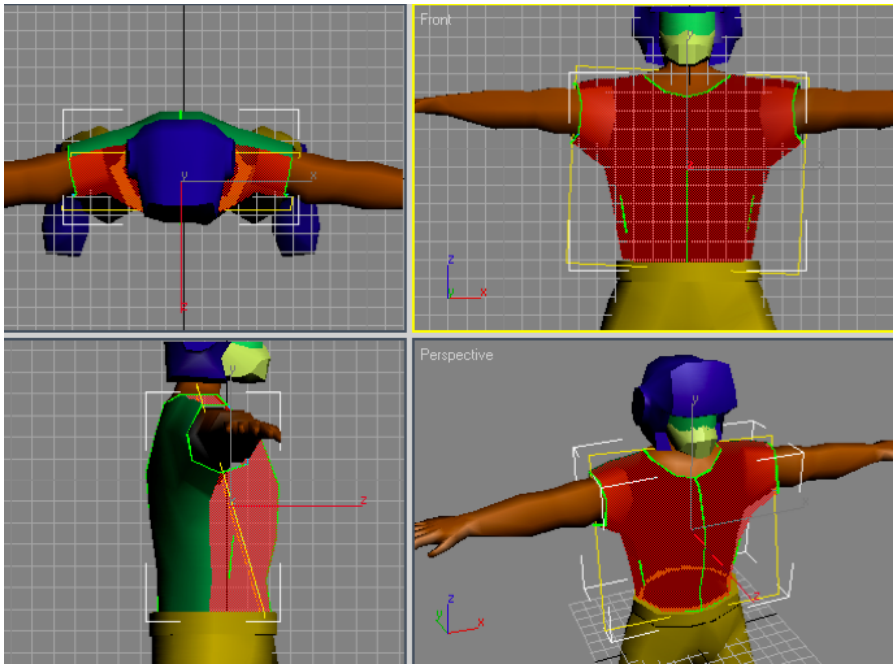
- Bump mapping modifies the normals, but does not change the actual geometry (the minute surface structures)
- This can lead to inconsistent shadow, e.g.,



- Displacement mapping use texture to modify the geometry
- As a result, a large amount of geometric data (polygons) are introduced.

Texture Mapping in 3DS

- The Unwrap UVW modifier is used to assign planar maps to sub-object selections (e.g., polygons, faces).
- E.g., map a T-shirt texture to the top of the character.



Raytracing

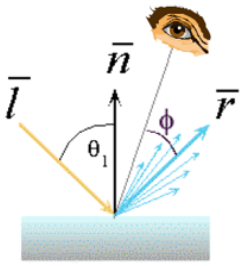
Lighting & Shading Review

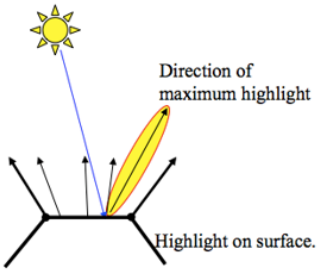
Lighting

- The way of evaluation of reflection from a small flat patches.
- Colours & intensities of reflection depend on surface properties (material) and light sources.

Shading

- Efficient ways of computing colourss for EVERY pixels of polygons - interpolation





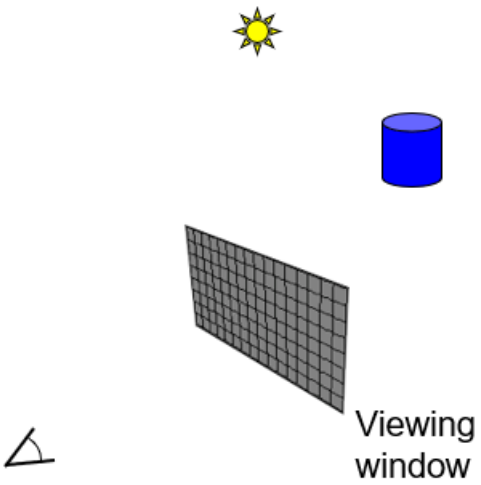
Rendering

- Rendering is the process of image creation, i.e., creating images from given viewpoint, object models and light sources using appropriate lighting and shading models.
- It involves two types of computation:
  - determining where on a surface of an object to perform shade computation. Only the points that correspond to image pixels will be evaluated. This can be done through, for example, the rasterisation process of the scanline renderer.

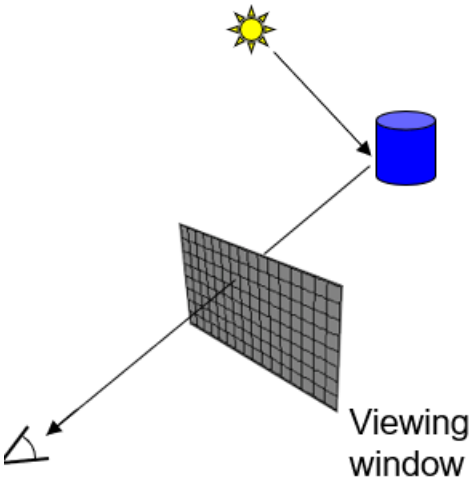
- computing pixel shade using appropriate lighting and shading models

Typical Scenario

- Consider the scenario of standard perspective viewing:
  - A scene (object models, lights, etc),
  - A viewer (camera), and a viewing window (image plane – where image will form)
- We want to create an image of what the viewer sees through this window – to *render* the scene.

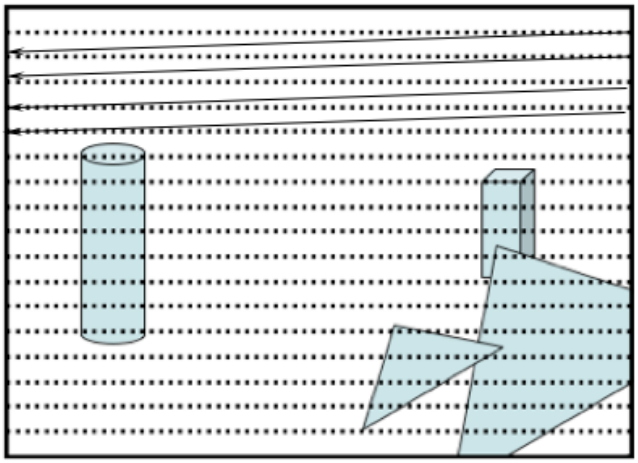


- First, the locations of the pixels on imaging plane tell us where we should evaluate the shades.
- For a given image pixel, its shade is the light reflected by the corresponding point on the surface – this tells us where on the object to evaluate the lighting equation.
- To create an image, we have different methods (i.e., *rendering methods*) to go through all the pixels (usually one-by-one)
  - Scanline rendering (or scanline renderer),
  - Raytracing (ray tracers).



Scanline Renderer

- Evaluate pixel shades one-by-one along horizontal lines (called scan lines) by using appropriate lighting and shading models, e.g., the Phong's.



- The rendering process can be summarised as:
  - At each pixel, determine ALL 3D points corresponding to the pixel and store their coordinates in memory. If multiple 3D points correspond to a single pixel, the one closest to the viewer will be visible (Z-buffer algorithms).
  - For each 3D point, perform lighting/reflection calculation to get the shade for that 3D point.
  - Store the calculated shade of the pixel in the memory (frame buffer).
  - Draw an image from frame buffer.
- The scan line renderer is a part of standard graphics pipeline.

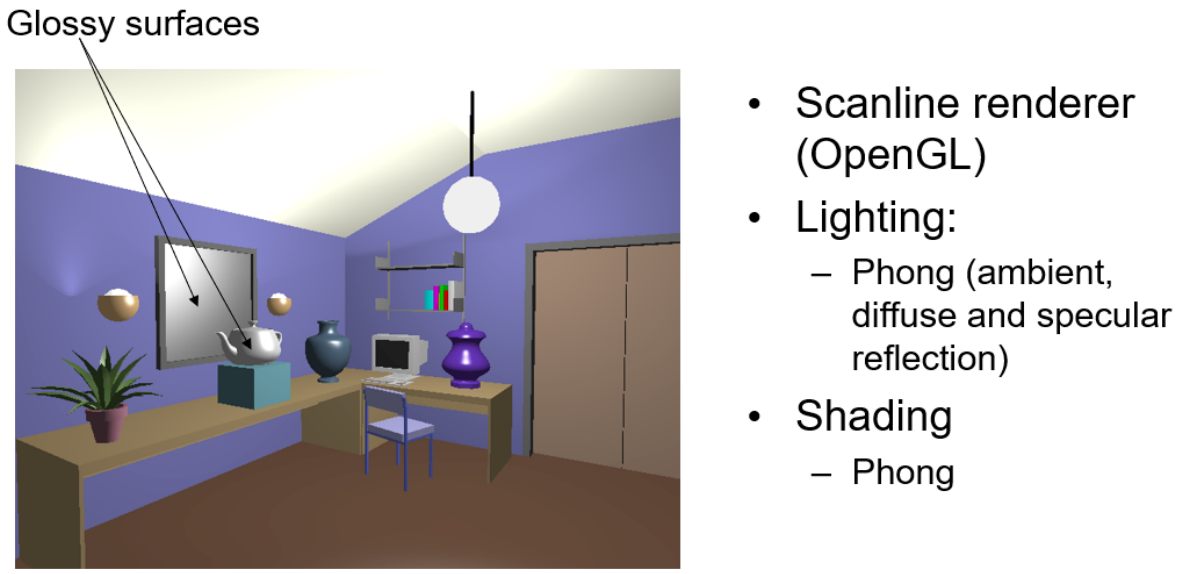
Some Observations

- In this process,

- Only the surfaces (polygons) facing the viewer is considered and their shades are evaluated (Hidden and back-facing surfaces are ignored),
- Only the lights that come directly from the light sources are considered (inter-object reflections are ignored).

- Pros & cons
  - Simple and efficient – a standard rendering method.
  - Some important aspects are ignored
    - Inter-object reflections among mirror-like objects are missing.
    - Refraction is ignored - transparent objects
    - Shadows cast by invisible objects are ignored

An Example



More Realistic Rendering

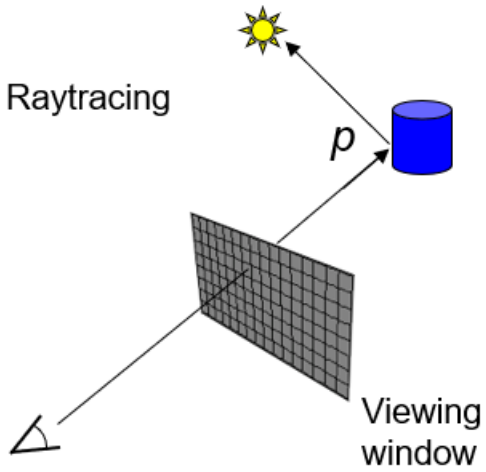
- To achieve a higher level of realism, we need a better rendering method to pick up the optics ignored by the scanline renders, i.e.,
  - inter-reflection between glossy objects, e.g., mirrors and polished surfaces.
  - refraction from transparent objects, e.g., glass, water, etc.
  - shadows cast by visible and invisible objects
- The above requirements implies we need a better way to simulate the true optics.
- Raytracing provides a conceptually simplest solution to this problem.

Renderings from Raytracing Renderers

- More realistic:
  - glossy surfaces: the mirror, floor board, tabletop, etc
  - Inter-reflection between the mirror, chrome teapot and floor
  - Shadows

Raytracing

- Raytracing abandons the concept of scanlines – when working with a pixel, a ray tracing algorithm does not follow the order of scanline.
- In fact, it deals with pixels in blocks. This is reasonable - if a pixel takes the shade of some object, it very possible that the nearby pixels also represent the same object.

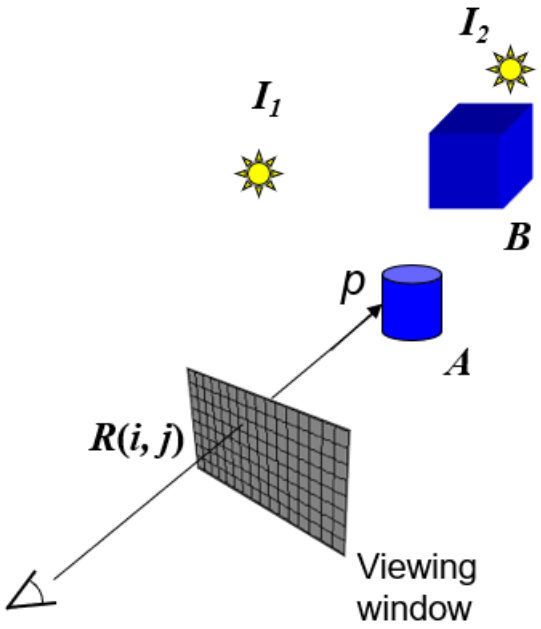


Principle of Raytracing

- Ray tracing also use the positions of pixels on imaging plane to decide where on the object the shade is evaluated.
- It determines the point on the object by shooting out a ray from the eye through the centre of each pixel (grid) and tracing the path of light backwards toward the scene.
  - It much easier to trace the paths backwards.

- In practice it is more common to shoot out a number of rays per pixel (and use the average shades).

- We call such rays primary rays (in comparison with secondary rays).



Destinations of Primary Rays

- A primary ray ends in one of three ways:

  1. **goes into infinity**. Such rays contribute nothing to image creation, so we do nothing about it.
  2. **reaches a light source**. That means the viewer can see the light source, so the light source decides the shade of the pixel.
  3. **hits an object.** That means the viewer sees a point of the object and we need to calculate the reflection from or shade of that point.

- The shading calculation for the first two cases are simple:

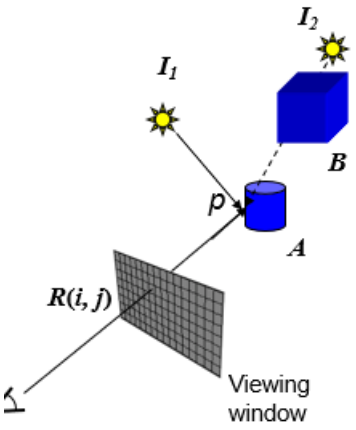
  1. In case 1, we have a dark pixel, and
  2. In case 2, the shade of the pixel is the colour & intensity of the light source.
  3. Case 3 is more interesting (and a bit complicated in comparison).

When a Ray Hits an Object

- The shade of the pixel is determined by the intensity of reflected light at that point. This reflection can be evaluated using appropriate lighting/reflection model, e.g., Phong's.
- Of course, the reflection depends on:
- Light sources

- The amount of light reaching this point from the various light sources (including the scattered lights from other objects).

- Material. The reflective and colour properties of the surface (material).



Secondary Rays

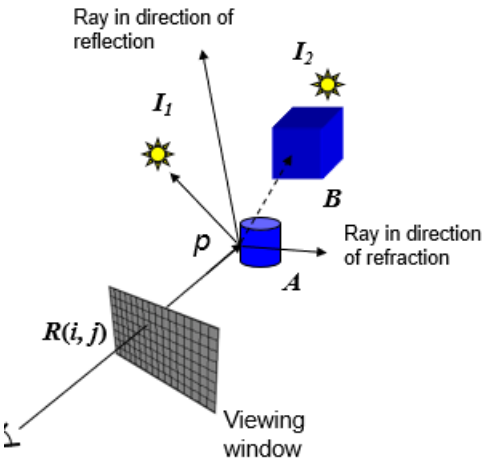
- To calculate the shade for a point on object ***A***, we need to send out rays from that point – these rays are called secondary rays (v.s. primary rays).

- Three types of secondary rays are sent out:

  - rays to **each** light source to see which light source can be reached (free of obstacles that block lights)

  - a ray in the direction of reflection if the surface is glossy (reflective).

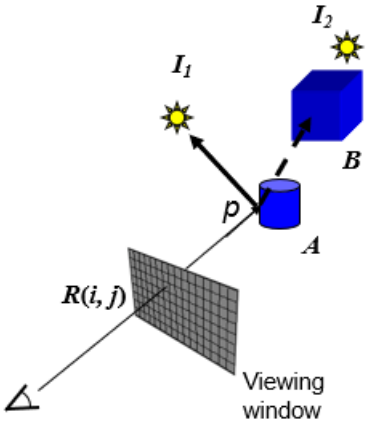
  - a ray in the direction of refraction/transmission if the object is transparent.



If the object is opaque and with a matte surface, then contributions from reflection and refraction will be zero or very small.

Secondary Rays: Rays to Light Sources

- If a ray succeeds in reaching a light source, then the point is illuminated by the source (e.g. I_1)
- Otherwise it is not illuminated by the light source, and hence is in the shadow of a blocking object. (e.g. I_2 and object B)
- Therefore the rays to light sources are called *shadow rays* (or *shadow feelers*).

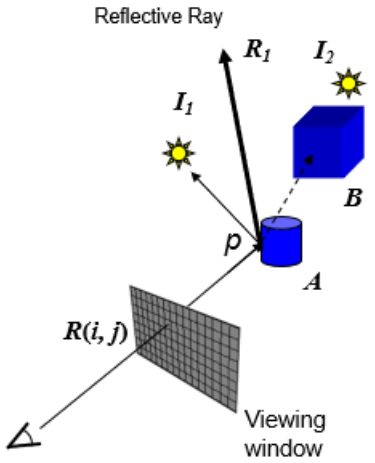


Shade Calculation

- If a point is illuminated by a light source (e.g. I_1), the reflection from that point can be calculated from the known position of the light source, the surface normal and the direction of the viewer.
- We have known this already, e.g., use the Phong lighting model to do the job.

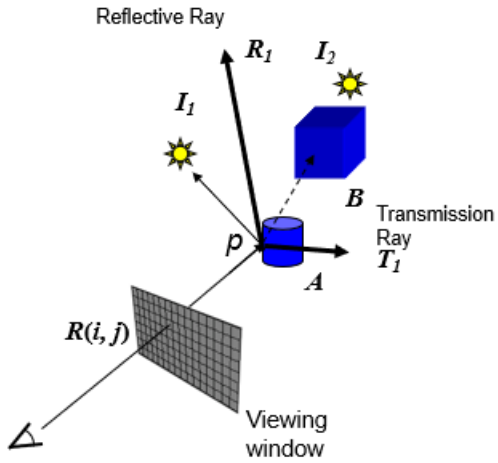
Secondary Rays: Reflective Ray

- The ray cast in the direction of reflection is used to check if any light comes from the direction of reflection.
- If the reflection ray hits an object, the contributions (to the shade of the pixel) from the object has to be considered.
- Doing so makes it possible to produce the mirror effect.



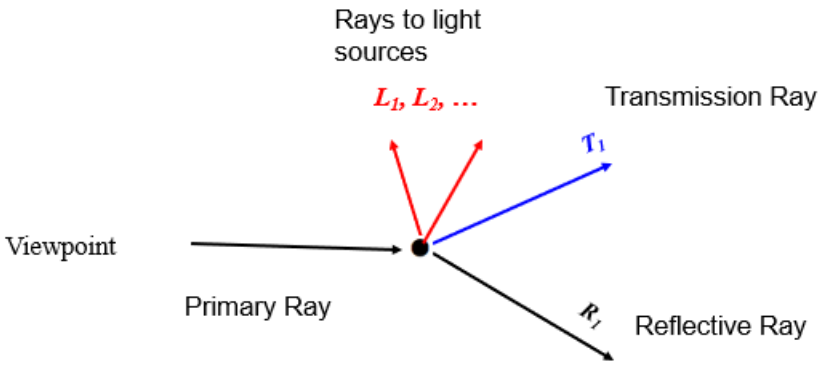
Secondary Rays: Transmission Rays

- Similarly, the ray cast in the direction of refraction is for checking contributions from light rays that come through refraction (i.e., rays reach the point by passing through object *A*).
- This makes it possible to produce the lens/transparency effect. More on this topic later.



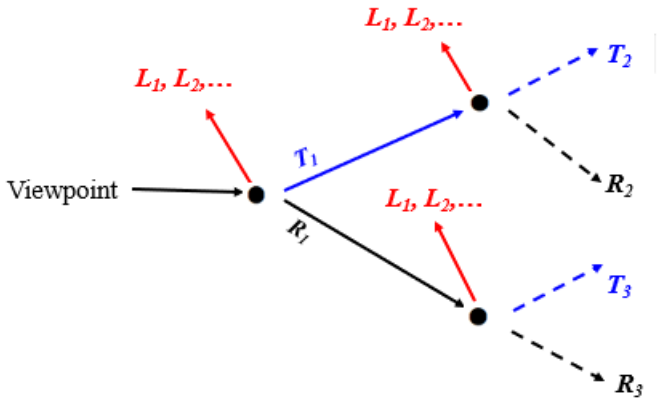
Ray Diagram

- The process has shown that a primary ray spawns a few secondary rays when it hits a surface.
- The structure of the rays can be illustrated by a simple diagram called ray tree.



Tracing Deeper

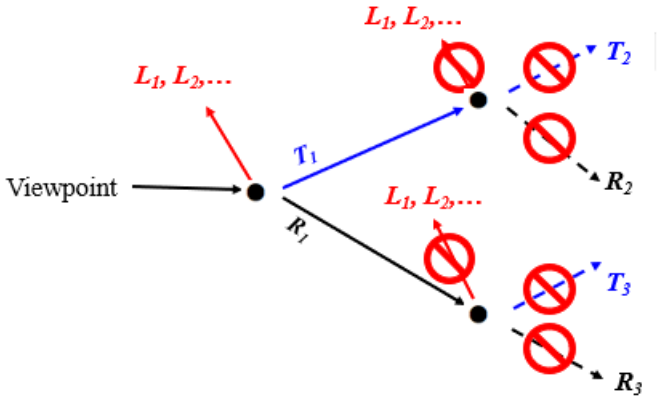
- In reality, the transmission and reflection rays will not stop when they hit an object. They will keep going in space.
- To trace the reflective and transmission rays further down their paths, we treat them in the exactly the same way as we do with the primary rays.
- The ray tree becomes:



- Now the ray tree is formed of two levels of simple ray-tracing – we say the ray-tracing depth/level is 2.

Simple Ray Tracing

- If we determine the shade of a pixel by tracing all the secondary rays and stop tracing the transmission and reflection rays any further, then we have performed the simple ray tracing (or standard ray tracing as called by some).



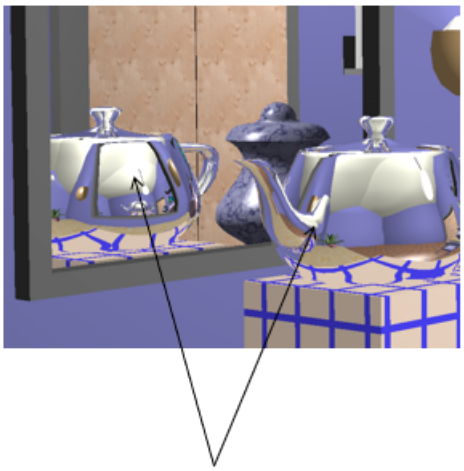
Simple Ray Tracing v.s. Scanline Rendering

- Simple/standard ray tracing produces renderings no better than those produced by a scanline renderer.
- It can pick up the shadows cast by the objects.
- But modern scanline renderers can handle shadows, too (as a property of light).
- Inter-reflection has not been properly handled
- But much slower than the scanline renderer.
- Ray tracing does not need:
    - polygon clipping,
    - z-buffer depth ordering,
    - perspective transform.
- But it needs to determine where the primary rays end.

- Time consuming.
- Binary tree or octree algorithm can find a use here

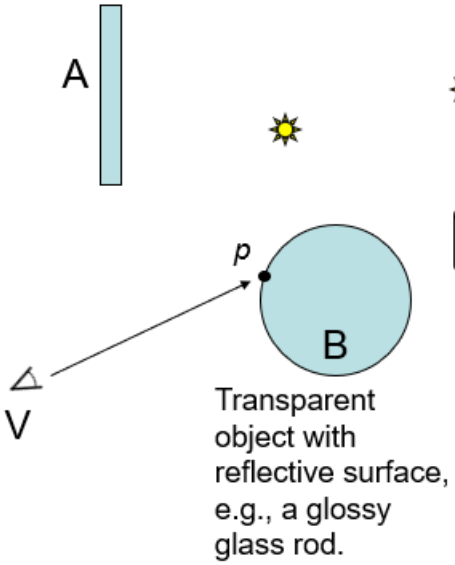
Tracing T and R Rays

- When mirror-like objects face each other, we expect to see the effect of "images of images" – the result of inter-reflection.
- The simple/standard ray-tracing model cannot produce such optical effect.
- To handle the inter-reflection among shining objects, we need to trace the T and R rays further down.



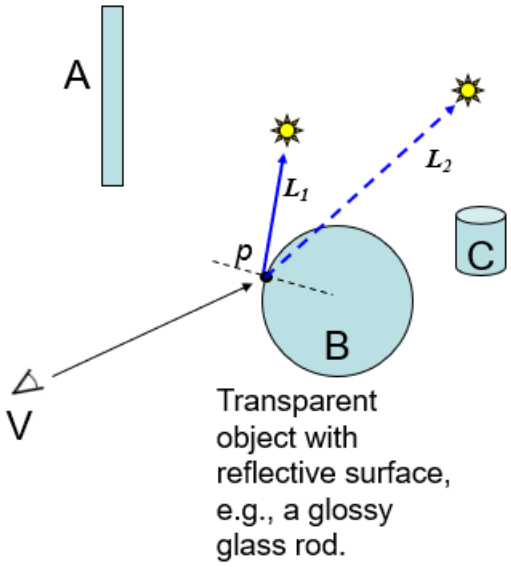
"Image of image"

Tracing Reflective and Transmission Rays



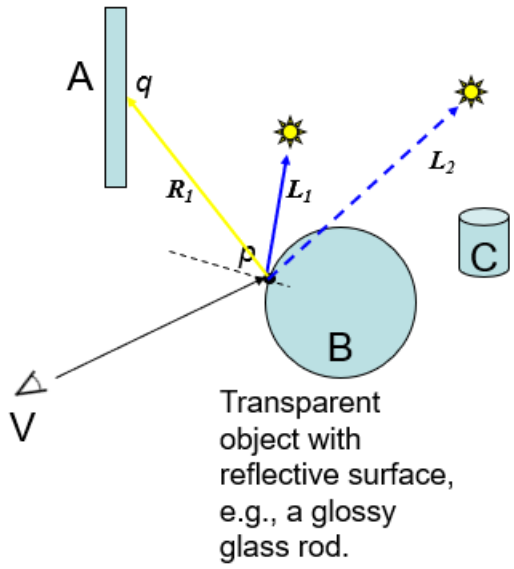
Transparent object with reflective surface, e.g., a glossy glass rod.

- Suppose B is a *transparent* object with *reflective* surface, e.g., a glass rod.
- Viewing B from viewpoint V, we expect to see
  - The image of A - result of reflection of the reflective surface (mirror effect).
  - The image of C - result of refraction (lens effect).

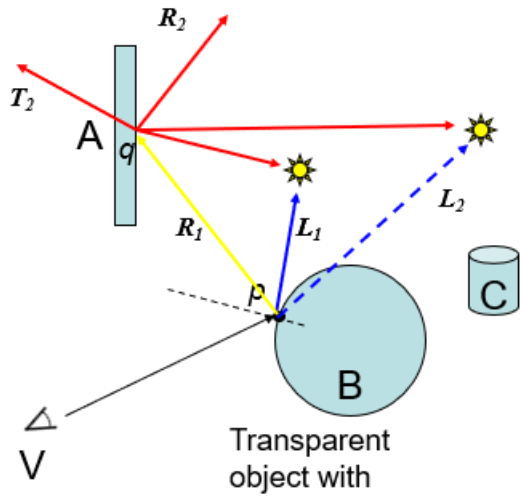


- First, two rays to the light sources, L_1 and L_2, are fired from *p*.
- L_1 contributes to the shade of *p*.
- L_2 does not (due to self-occlusion).

Trace Reflective Ray



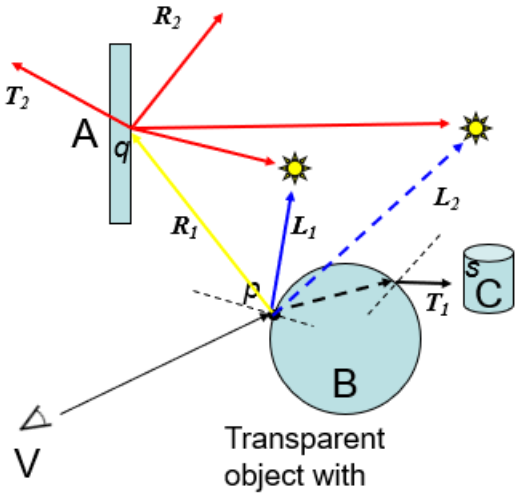
- Then a reflective ray ***R₁*** is fired in the direction of reflection at ***p*** (yellow path) which reaches object ***A*** at point **q.**
- The shade of point **q** on **A** will contribute to the shade of **p** on **B,** so it has to be evaluated.



Transparent object with reflective surface, e.g., a glossy glass rod.

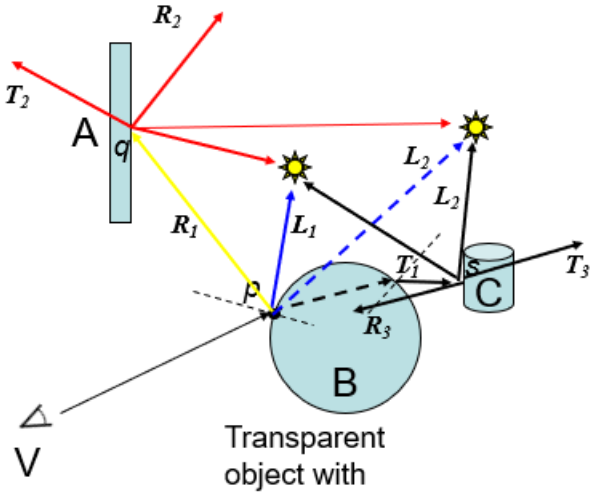
- To do so, we treat the ray R_1 as a primary ray and perform standard ray-tracing on it.
- This is represented as the red paths. The reflective and transmission ray , R_2 and T_2 may not be useful at this level depending upon the material properties of object **A** (e.g, when **A** is an opaque object with rough surface).

Trace Transmission



Transparent object with reflective surface, e.g., a glossy glass rod.

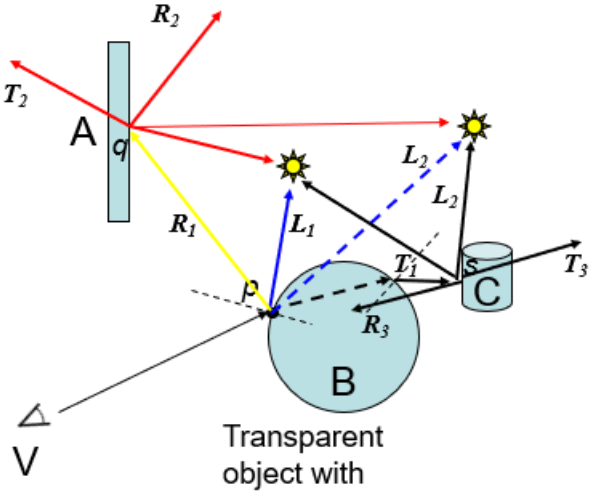
- Similarly, a ray, T_1, is fired in the direction of refraction/transmission .



Transparent object with reflective surface, e.g., a glossy glass rod.

- Performing standard ray-tracing on T_1 ,at point **s**, we have 4 new rays:
  - two to light sources,
  - one in the direction of reflection, R_3 , and
  - one in the direction of refraction, T_3.

Observations



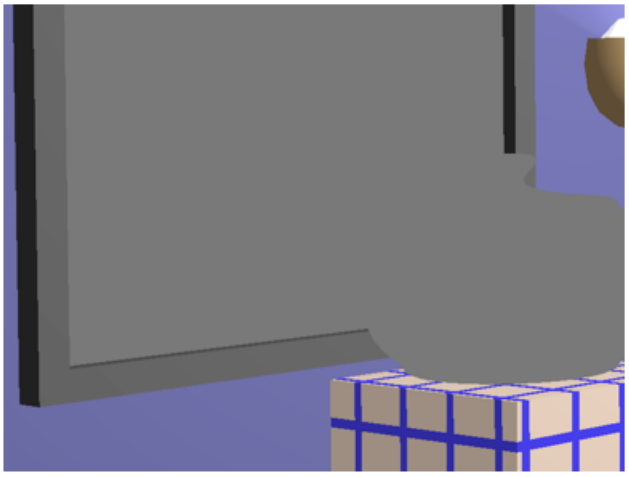
Transparent object with reflective surface, e.g., a glossy glass rod.

- To take reflection and refraction into account, we have recursively applied the standard ray-tracing algorithm on rays, R_1 and T_1.
- If necessary, this process can be further applied to the secondary rays spawned at point **q** on object **A** (i.e., R_2, T_2) and point **s** on object **C** (i.e., R_3, T_3).

Where to Stop?

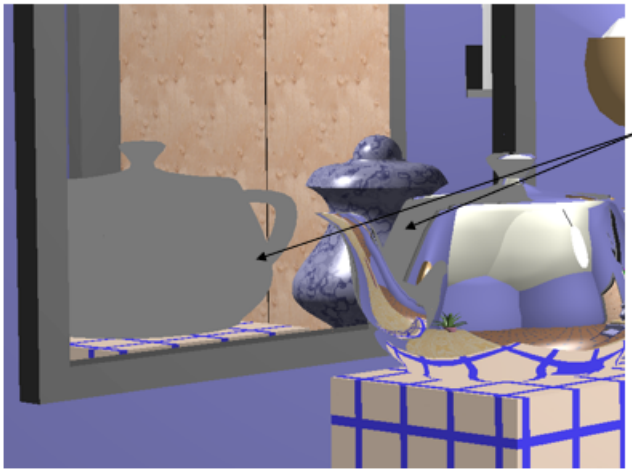
- In real world, the optical inter-reflections repeat themselves endlessly.
- But computers cannot do this without limit.
- We do not need to trace a depth of more than 10 (a depth of 6 will produce impressive result).

Test Scene



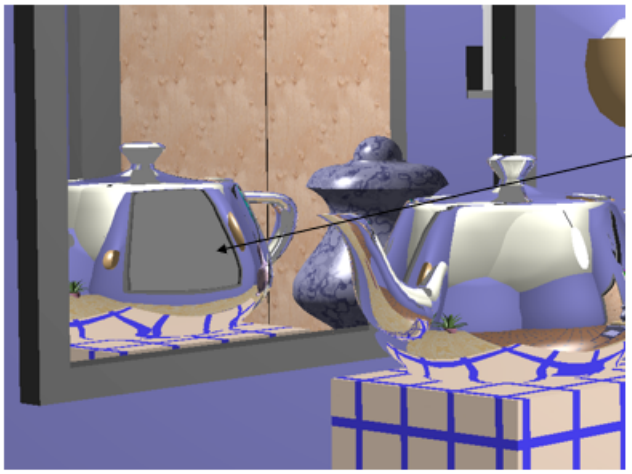
Standard ray tracing

Tree depth =1



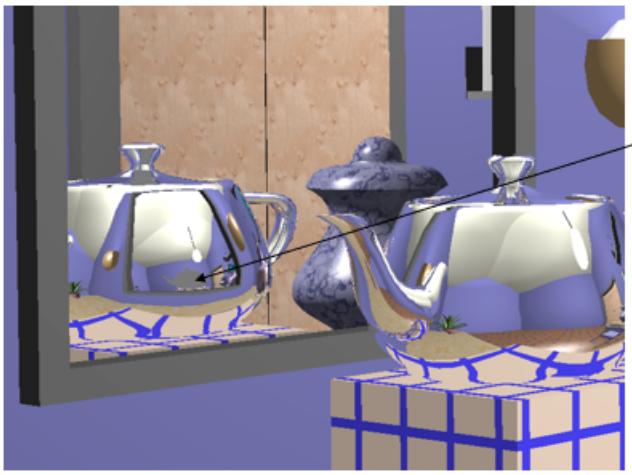
Standard ray tracing (tree depth = 2).

Only ambient shade on reflection of mirror and teapot.



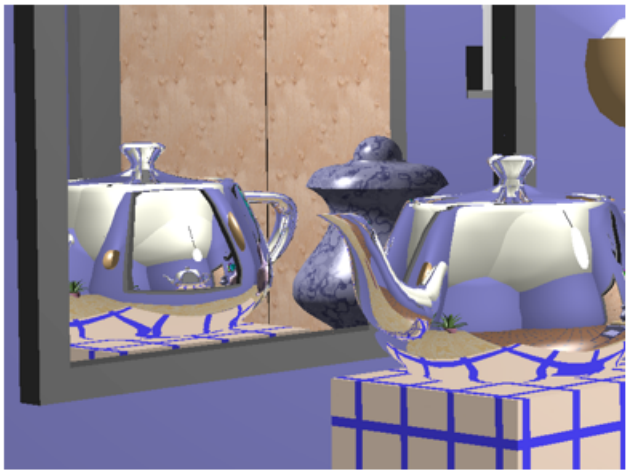
Ray tree depth = 3.

Only ambient shade on reflection of mirror in teapot.

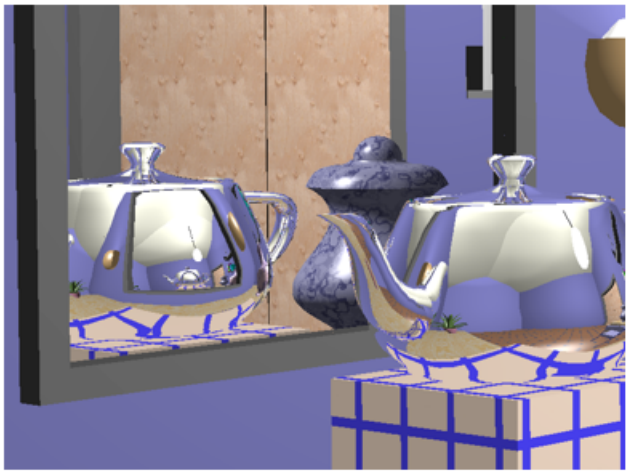


Ray tree depth = 4.

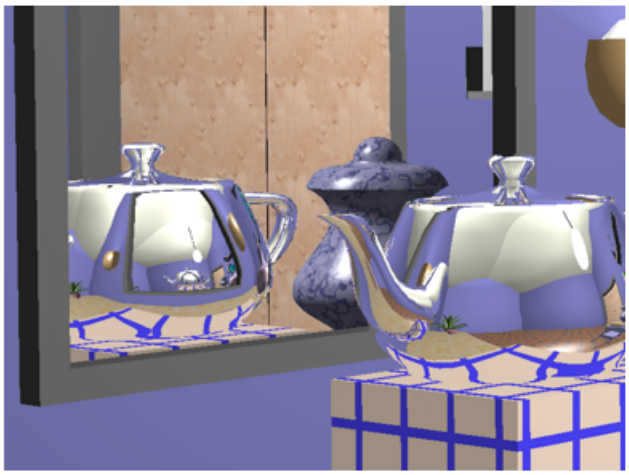
Only ambient shade on reflection of teapot in reflection of mirror in teapot.



Ray tree depth = 5.

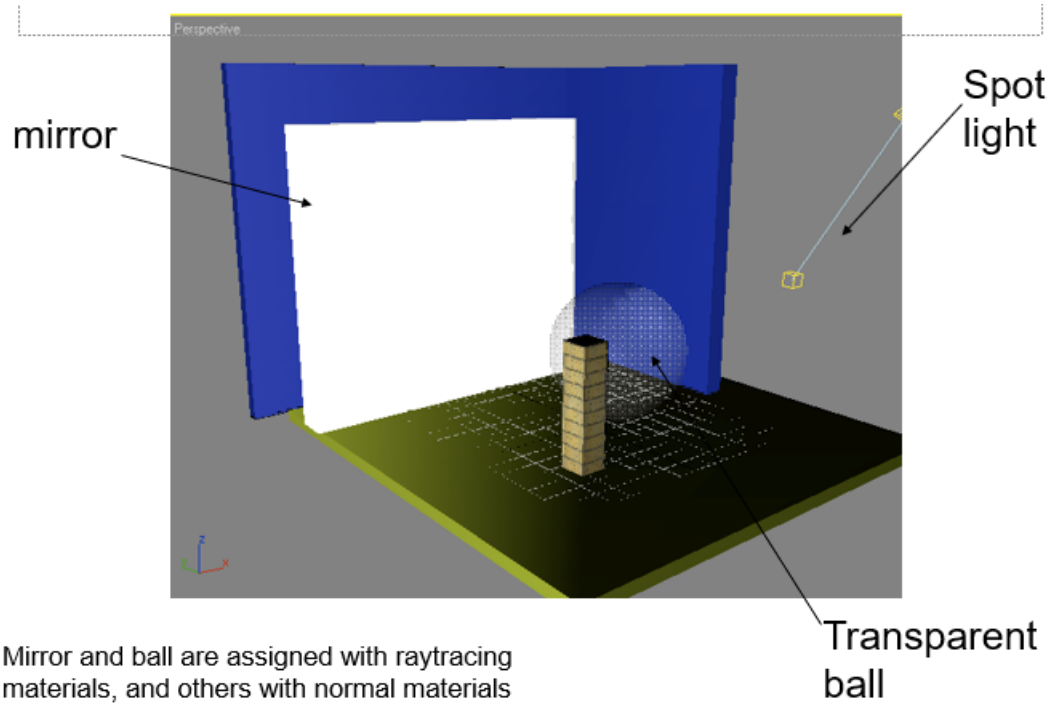


Ray tree depth = 6.



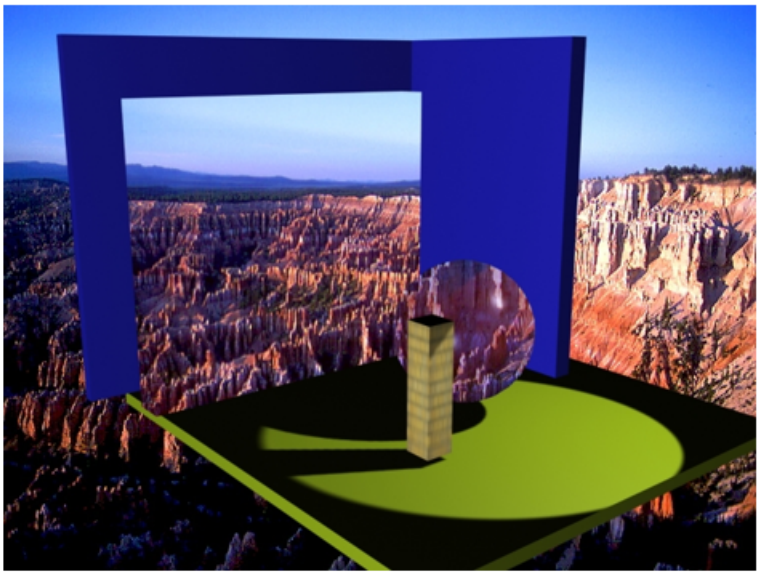
Ray tree depth = 7.

3DS Max Renderer – Preview Window



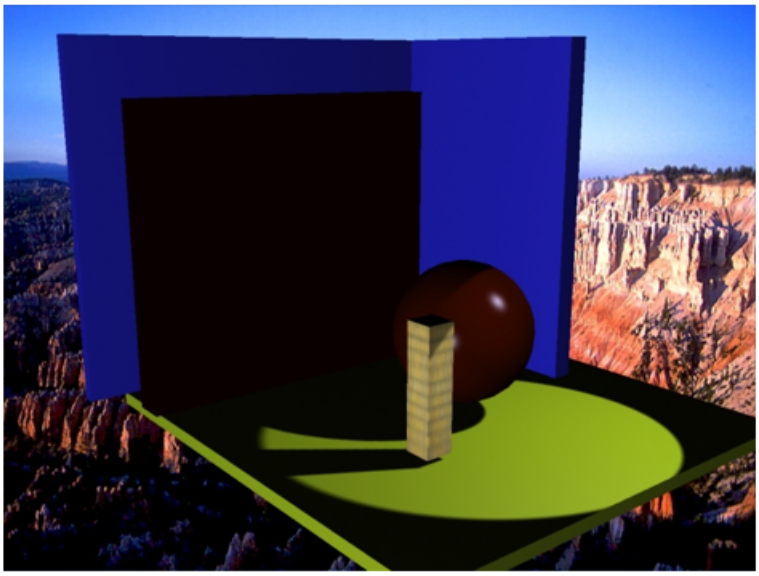
Mirror and ball are assigned with raytracing materials, and others with normal materials

3DS Max Renderer – Default Scanline Renderer

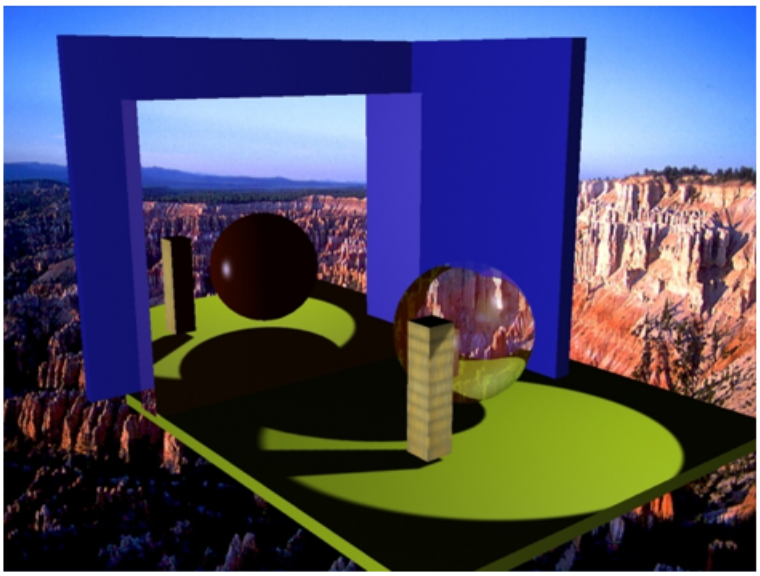


It implements:
1.shadows as a property of light sources,
2.reflective and refractive properties in materials

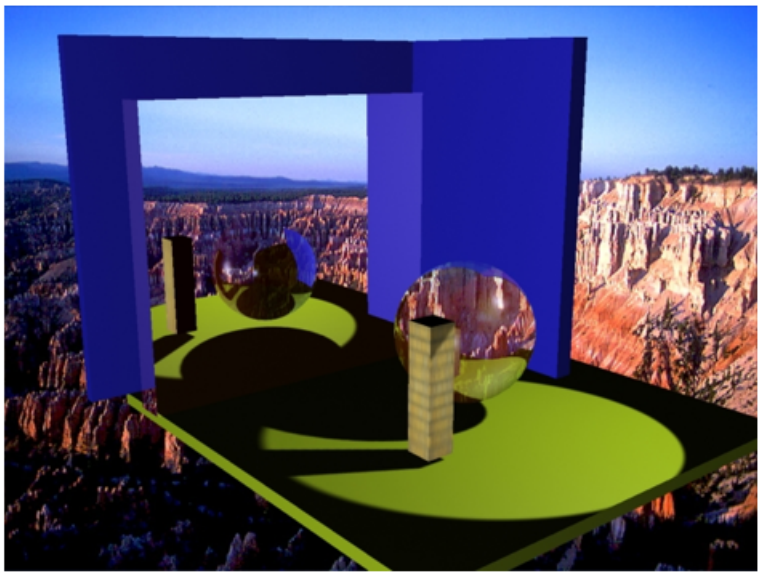
3DS Max Renderer- Mental Ray Renderer



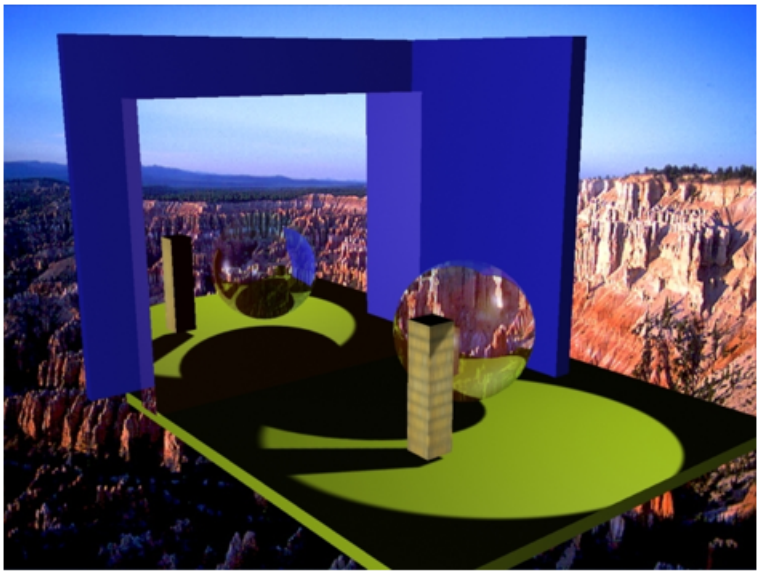
raytracing Depth = 1



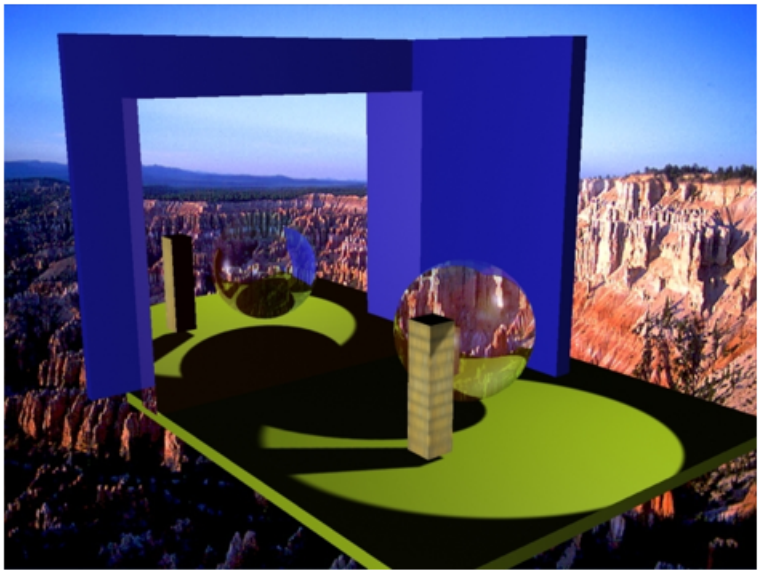
raytracing Depth = 2



raytracing Depth = 3



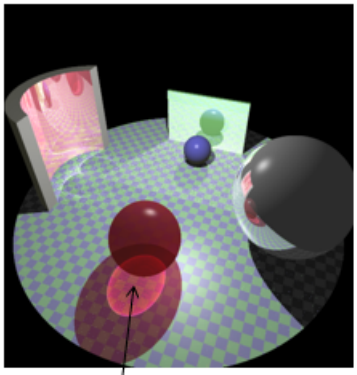
raytracing Depth = 4



raytracing Depth = 7

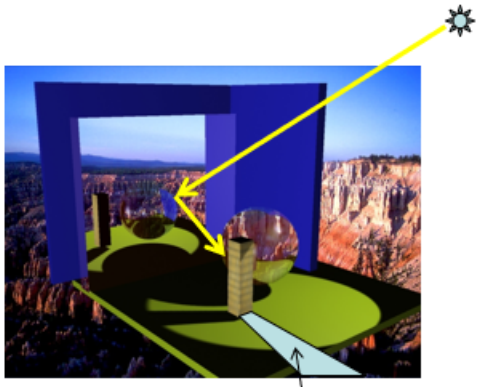
Some Limitations

- The ray tracing scheme we have considered produces much better renderings than scanline renderers, but it is not perfect.
- First, the rays are traced from the eye, which means
  - Refraction is not physically correct (incident angle and refraction angle are different),
  - Unable to render the caustic (focusing) effects produced by lens, which requires tracing starts from light source.



Caustic due to the red transparent ball

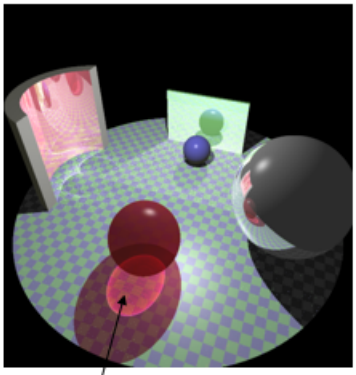
- Secondly, the light rays reflected by objects do not cast shadows, because shadow rays are cast only to light sources.



In real world, shadow would be cast by the light reflected by the mirror.

Other Raytracing Methods

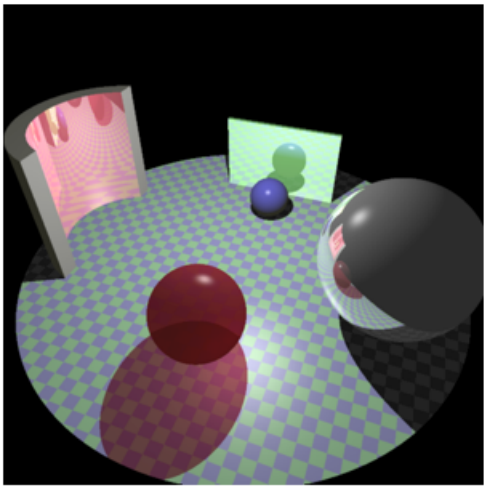
- Bidirectional ray tracing: tracing is initiated from both the viewer and light source sides.
- Strategy
  - Cast limited number of light rays (called photons) from each light source and trace their paths (Why limited number? in which direction?)
  - Accumulate number of hits (photons) on surfaces
  - Trace rays from viewer's eye, and determine the pixel shading according to the number of hits collected at a particular surface point



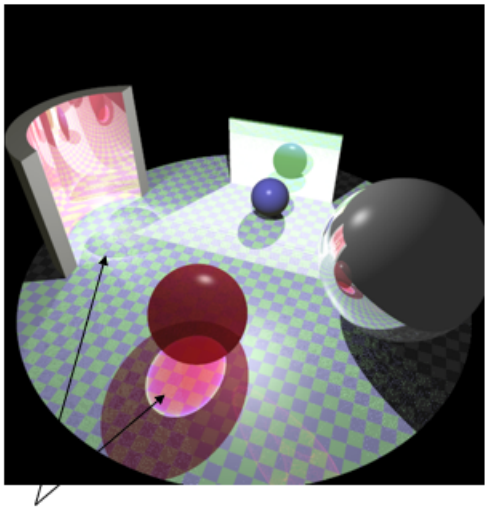
Caustic due to red transparent ball

Bidirectional Example

Conventional raytracing

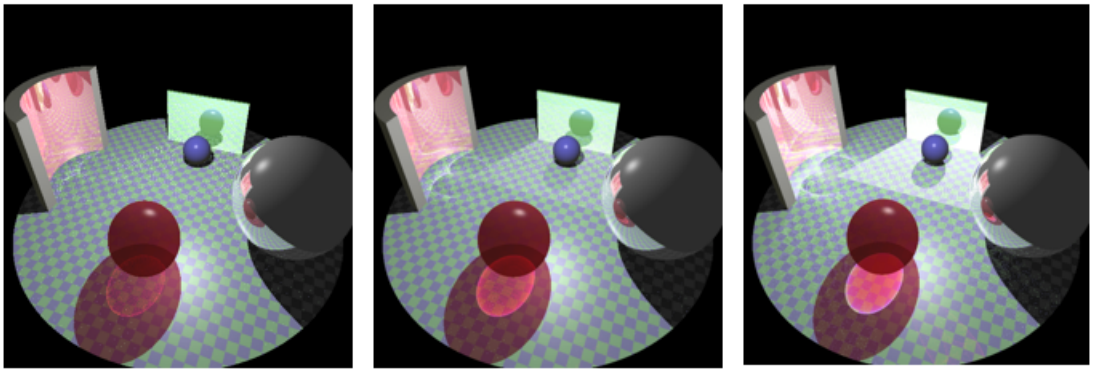


Bidirectional



Caustic due to red transparent ball and the cylindrical mirror

Effects of Number of Rays



200 rays used in forward ray tracing (i.e., from the light source)

400 rays used

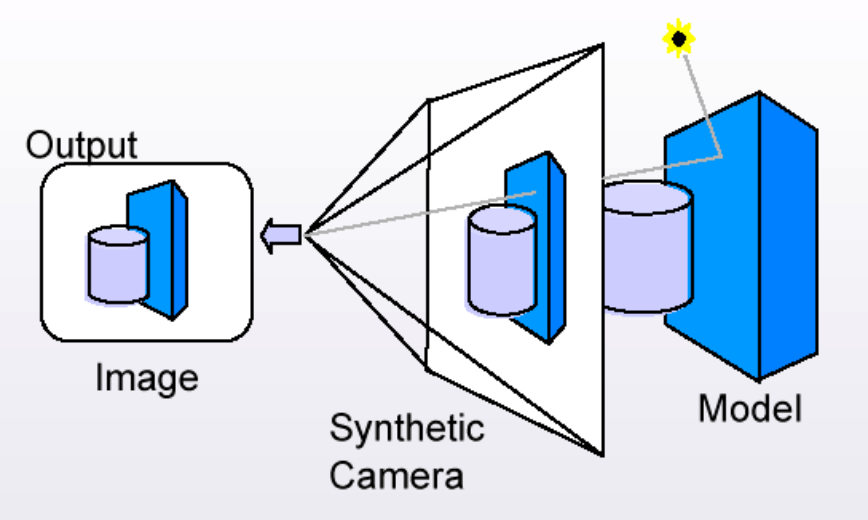
800 rays used

Introduction to Image Based Approaches

- 3D graphics v.s. photographs
- Image-based modelling (IBM)
  - 3D graphics v.s. computer vision
  - Some techniques
- Image-based rendering (IBR)
  - Issues & techniques
- Some examples

3D Computer Graphics

- 3D graphics is about creating images from models - as if we are using a virtual camera



- It is very flexible

  - We can create images for things that do not exist.
  - We can create image from view angles which are difficult/impossible to achieve by using real cameras.
  - We can navigate a scene in whatever possible ways.
  - etc.
- However, all these come at certain costs:
  - Laborious modelling work.
  - Quality of rendered images are not as realistic as photos.

It takes long time to render, etc.

Photographs

- Photographs are taken by real cameras
  - Easy to take, quick to display and high visual fidelity.
- However
  - It is not always possible to take a photo.
  - Even for a simple object, it can produce infinite number of different photos: you can never be sure you have right photos that you need.
  - Not convenient for interactive applications such as navigate a scene – the real camera must follow a path in a scene (which is hard sometimes – even by cameras onboard drones).

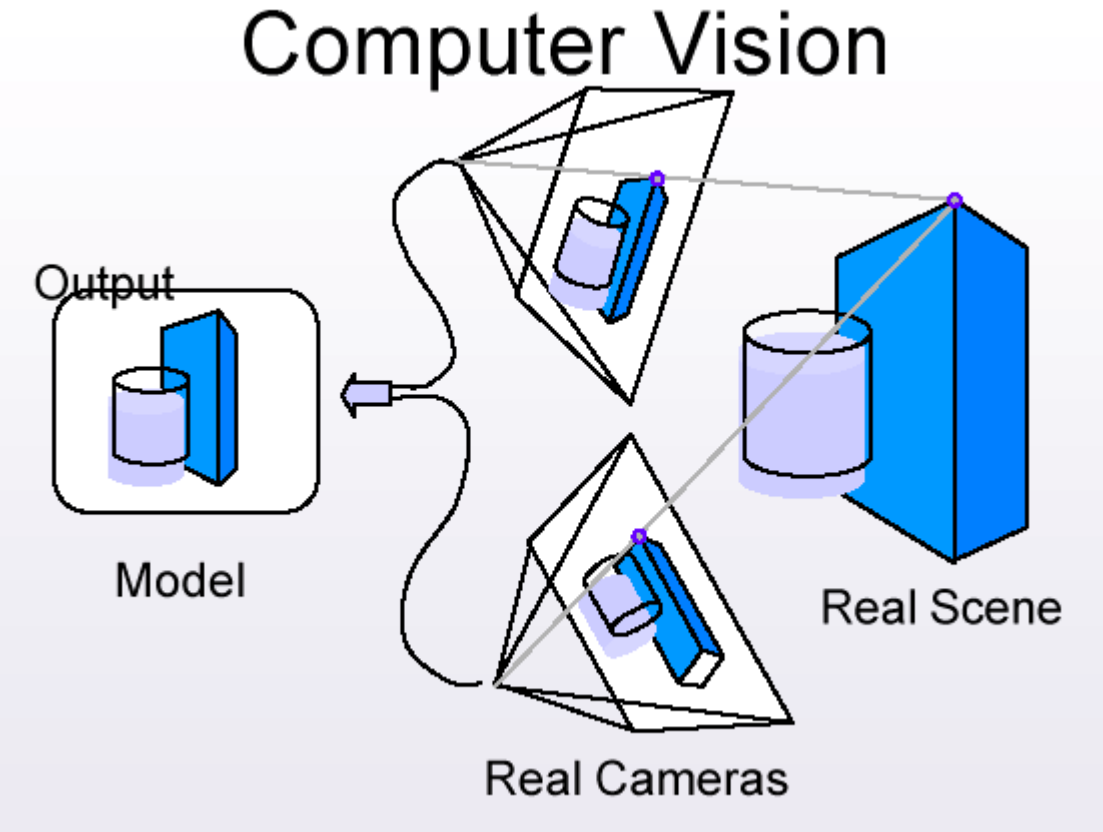
Image-Based Approaches – Motivations

- Image-based methods are about getting the best from both worlds
  - Reduce the workload of modelling.
  - Get the visual realism of photos at high speed
  - Keep the flexibility of 3D graphics, e.g., easy production of photos (renderings) at any view angles, and easy 3D navigation.
  - Two main categories:
  - Image-based modelling (IBM)
  - Image-based rendering (IBR)

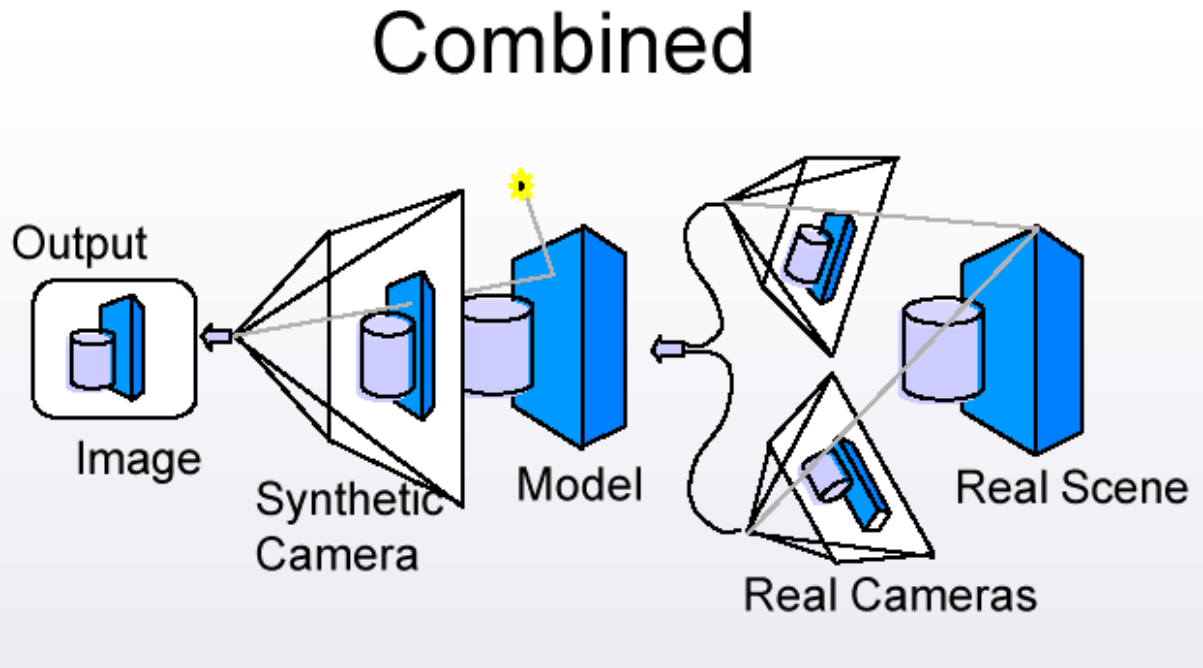
Image-based Modelling

- Aims of image-based modelling (IBM): automatic (or nearly automatic) creation of 3D models from images to save work of manual modelling.
- Principles of IBM: computer vision and other techniques are used for acquiring 3D models.
- The way it works is different from 3D laser scanners.

3D Model Acquisition Using Computer Vision – Stereo Vision Approach



IBM – Combines Vision and Graphics

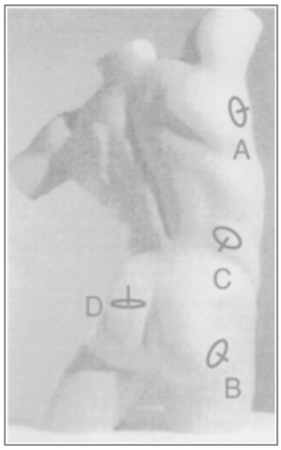


Different Visual Cues Can be Used for IBM

- In image-based modelling, various computer vision methods can be used (or have been attempted) for acquiring 3D object models.

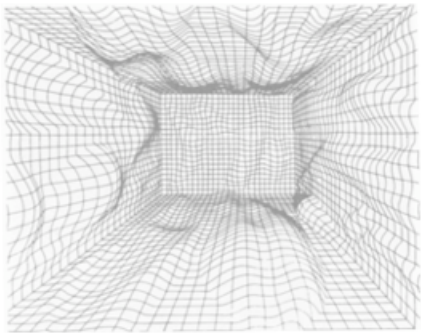
- The methods for creating 3D models of objects vary in principle, efficiency and accuracy.
- Various visual cues can be used, alone or in combination. This category of methods was given a name of shape-from-X, where X could be any of the various visual cues:
  - Shading, texture, motion, stereo, silhouette/contour, and etc.

Shape-from-Shading



- One can judge the direction of surface normals according to shading info.
- Normal directions plus other constraints allow shape of the torso to be recovered

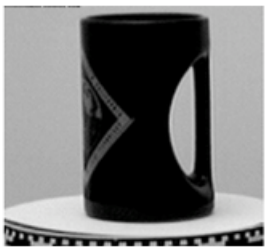
Shape-From-Texture



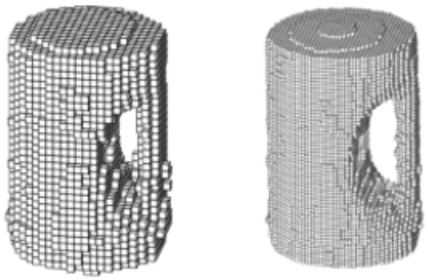
- If one assume that the elements of texture (texels) have the same shape and size, then depth can be recovered

Shape-From-Silhouette

- Shape-from-silhouette system consists of a motion platform (turntable) and a stationary video camera.
- As the object spins on the turntable, video images are captured, the turntable angle is computed, and silhouettes are computed.



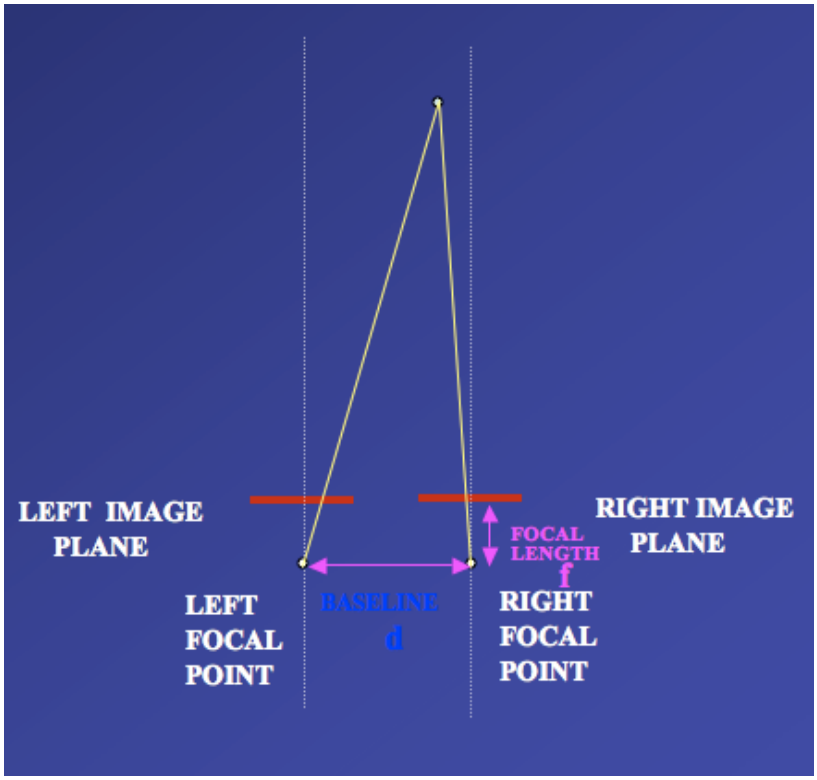
- Each silhouette, together with the corresponding centre of projection of the camera, defines a generalized cone in space within which the object must lie.
- The intersection of these cones in 3D defines a bounding volume for the object.
- Different techniques can be used for computing and representing this volume, such as an octree representation.
- Obviously, such approaches are not suitable for hollow objects or objects that have a lot of self-occlusion.



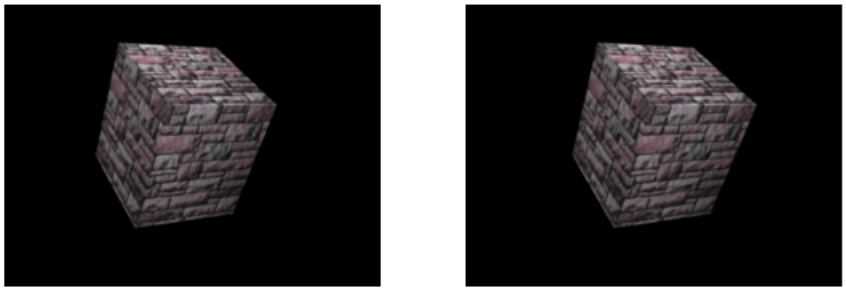
Lower-resolution octree model (5 levels) and higher-resolution octree model (6 levels)

Shape from Stereo Vision

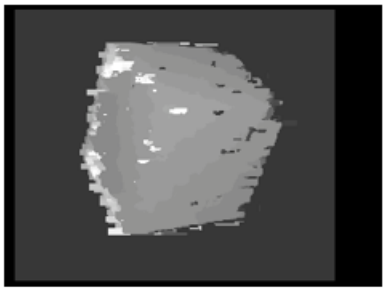
- Passive 2-camera system for triangulating 3D position of points in space to generate a depth map of an object or a scene.



Example



Focal length f=50mm, baseline=100mm



Disparity map

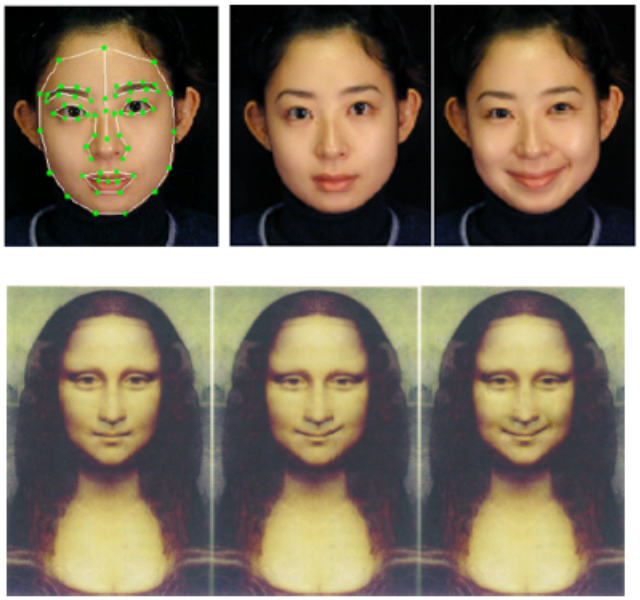
Image-based Rendering

- Image-based rendering tries to avoid direct use of accurate 3D models in creating photo-realistic renderings.
- There are two types of IBR applications
  - Traditional image-based rendering

- Create novel views from images: This includes techniques such as image warping, view interpolation
- No geometric modelling is involved

– Augmented reality – Insert virtual objects into photos/videos of real scenes so that the photos/ videos contain the objects that do not exist in reality

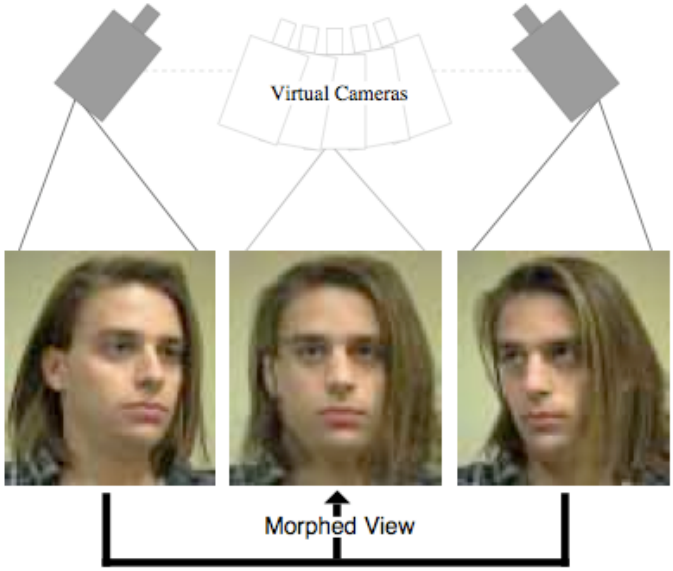
Image Warping



- Usually use a single image.
- Good visual effects can be achieved without much computation work.
- The results may not be very accurate.
- The assignment of the correspondence between the images is usually done manually and is hard to automate.

Image Morphing

- Involves two or more images of the same object from different view directions.
- Morphing calculation requires the followings:
  - Two images and their projection matrices – the perspective projection that simulate the optics of the cameras.
  - Correspondences between pixels/features in the two images, which can be established through feature-matching or user-interactions.
- No *a priori* knowledge of 3D shape information is needed.



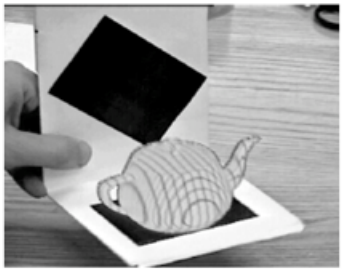
Augmented Reality (AR)

- AR refers to applications when synthetic objects are rendered into videos of real-world scenes.
  - E.g., a piece of furniture, a digital creature or actor needs to be rendered seamlessly into a video of the real scene.
- The task involves two sub-tasks:
  - Insertion of virtual objects with correct viewing conditions (i.e., correct sizes, orientations and positions). This is to guarantee the geometry of the virtual objects visually correct.
  - Ensuring the synthesized objects be lit consistently within their environments – to guarantee visual realism. This requires a proper simulation of the interplay of lights between the objects and their neighbours – this problem is called image-based lighting.

Insert Virtual Objects

- To insert synthesized objects with correct viewing positions, we need to know the camera parameters with which the photos/videos are taken. E.g., the focal length, position and orientation of the camera.
- Finding out camera parameters from images is a typical computer vision problem – camera calibration.
  - Some features on the images could be used to calibrate the camera.
  - Some good (computer vision) algorithms exist, e.g., Tsai's method.

Example

- A virtual object (a tea pot) is added to image/video of a real scene.
- The corners of the black polygons have been used to define the coordinate system for specifying the position and orientation of the teapot.

Image-Based Lighting

- IBL is about assigning correct illumination to virtual objects
- The aims of IBL is to ensure the virtual objects to be lit consistently within their environments, which is important for good visual realism.
  - See the example in the previous slide
- Image-based lighting is difficult
  - Much more difficult than the standard lighting problems.
  - We don't have the information of light sources or the models of the objects in the scene. It is difficult (if possible at all) to recover them from photos/videos.
- Two sub-problems:
  - how to measure and model the 3D light structure from photos/videos,
  - how to estimate the interplay of light among the scene objects without knowing their models.
- We use an example to *illustrate* the issues of and possible solutions for image-based lighting.

Example

- A photo of a real scene (left image). Virtual objects (a white ball, etc, right image) will be inserted into the photo/video.


- We want the insertion to be seamless:
  - correct view angle,

- correct lighting,
- correct reflection of other scene objects on the surface of the inserted object.

Camera Parameters

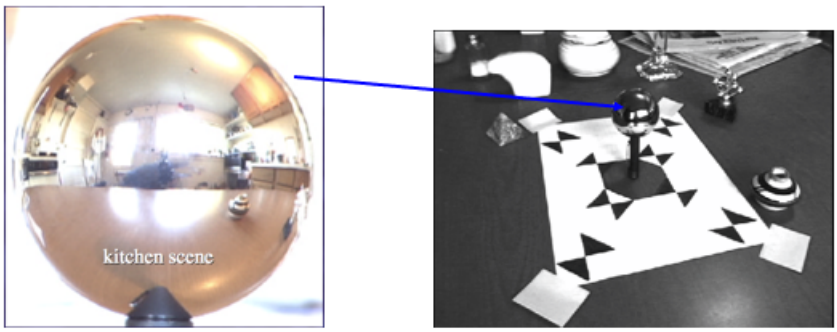
- Camera parameters can be recovered if we have access to the physical camera and scene.
- In this example, special scene objects (special patterns – the black triangular shapes) and a camera calibration algorithm are used for recovering camera parameters.

Illumination Estimation

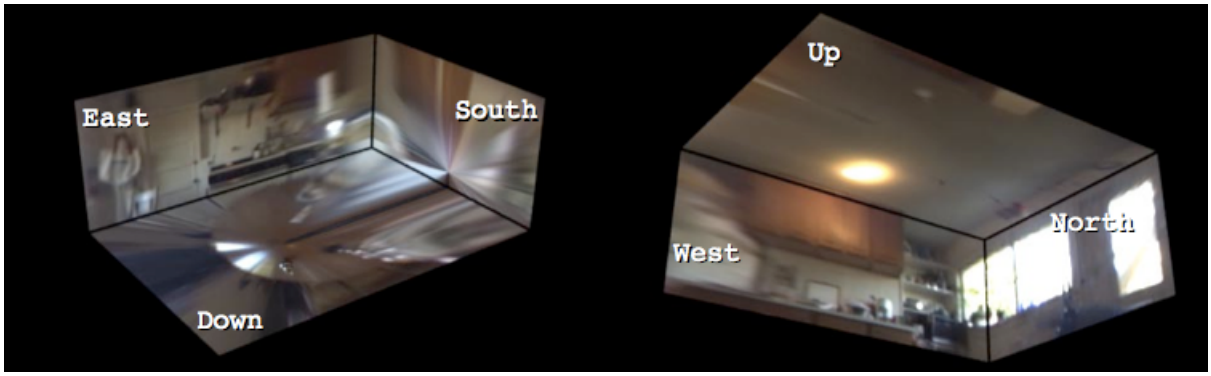
- To estimate lighting condition, one needs to know:
  - light sources (intensity and positions).
  - scene structure (what is the shape of the ? Where are the light sources? what other objects are in the room? What are the shapes, location and the surface properties of these objects?).
- Answers to these questions are impossible without an assessment of the physical environment.

Acquiring Lighting Model

- However, if one has the access to the physical environment, a light probe (a mirrored ball) can be used to record lighting conditions in the environment.
- The image of the light probe gives the information about the light sources and their locations.



- Mapping the image from a light probe onto a box (assuming the room has a normal box-like structure) to create an environment map.



- The environment map can be mapped onto virtual objects.
- The method has many limitations, such as the need for installation of a physical probe, no information of light and 3D structure, etc.
- The example shows what need to be done to achieve high quality of augmented reality.
- We still do not have a good solution to the problem of recovering illumination from images/videos.

Finally

- You won't find the materials for this lecture in the major graphics texts since the problems are largely still research matters.
- If interested, I recommend you check the SIGGRAPH papers and course notes.

Computer Animation Overview

Applications

- Special Effects (Movies, TV)
- Video Games
- Virtual Reality
- Simulation, Training, Military
- Medical treatment
- Robotics
- Visualization
- Communication

Types of Animation

- According to the way that motions are created, animation can be classified into:
  - Keyframe animation
    - interpolation
  - Character animation
    - Hierarchical modelling and kinematics
    - Motion capture
  - Physics simulation
    - Rigid body simulation
    - Laws of physics (dynamics, mechanics)
  - Procedural
    - L-system, fractal geometry, wave functions
    - Plants, water, clouds, gaseous phenomena

Theories and Technologies Behind Computer Animation

- Robotics
  - Kinematics of hierarchical structure
- Mechanics
  - Rigid body mechanics
  - Biomechanics
- Dynamics
  - Fluid dynamics
  - Aerodynamics
- Motion capture technology
  - computer vision, image processing, object tracking, etc.
- Artificial intelligence
- path planning, swarm intelligence, flocking, etc

Animation Tools

- Maya
- 3D Studio
- Lightwave
- Blender

Many more…

Industrial Computer Animation Production Pipeline

- Production pipeline: 3 stages
  - Pre-production stage
  - Production stage
  - Post-production stage

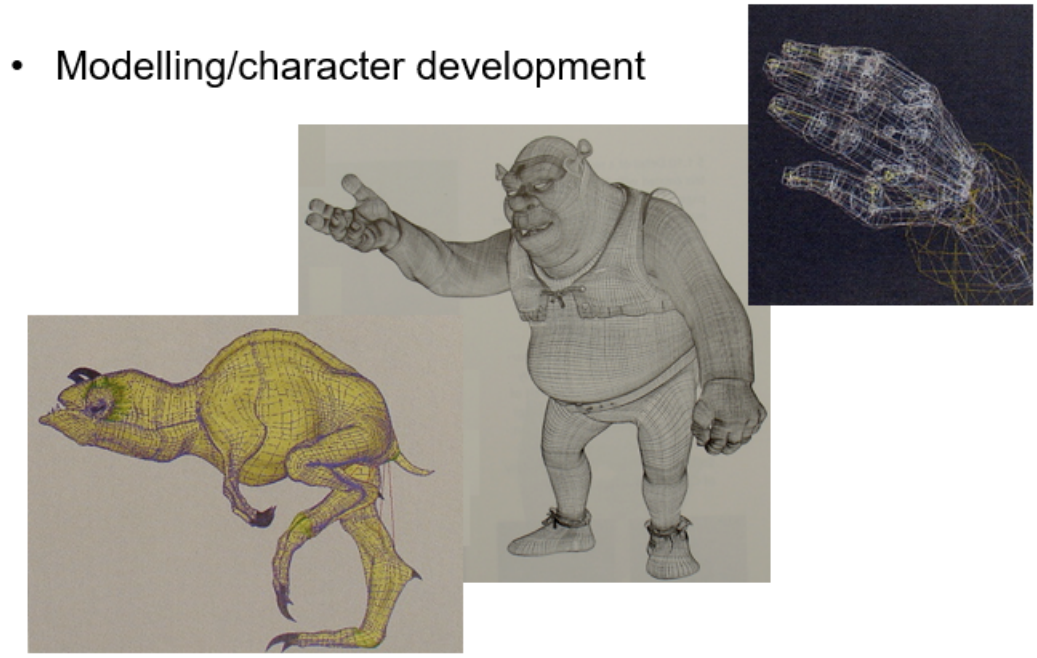
Stage 1: Pre-Production

- Conceptual Design
  - Story/game idea development, scriptwriting/game design documentation
  - Storyboarding - translate story into images

- Overall style, visual look, colour scheme - by painters, sculptors, illustrators, etc

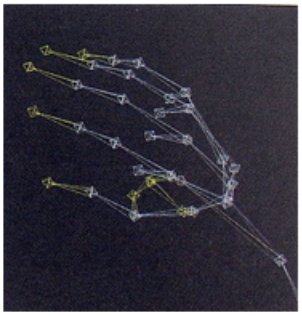
Stage 2: Production

- Modelling/character development



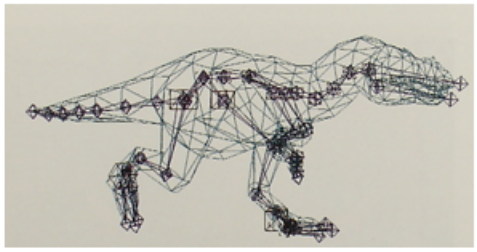
- Materials, textures, hairs, etc.

- Rigging: Design skeletons for the characters, IK computation



Muscles and **bones** used for rigging Shrek

- Animation
  - Keyframing
  - Motion capture
- Skinning: attach skins to the skeletons



- Other aspect of scene
  - Lighting, sound, …

- Rendering
  - Traditional graphics pipeline (polygons), or
  - Ray-based rendering.

- Dialogue, soundtracks, etc.

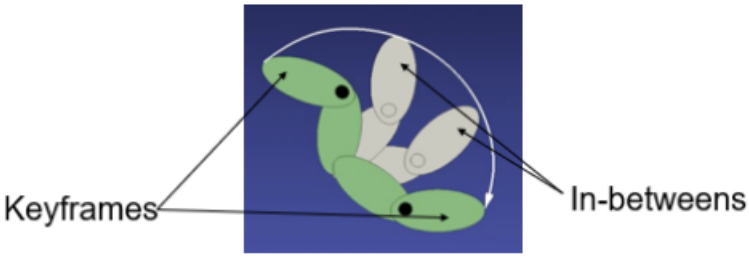
Stage 3: Post-Production

- Digital editing, recorded on video film for release

Keyframing and Principles of Animation

Keyframing

- A traditional animation technique.
- Artists to generate “key” frames/pictures at particular moment in animation timeline - this called “keyframes”.
- Additional "in-between" frames are drawn by less experienced persons.

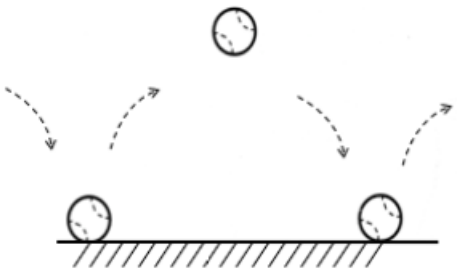


Interpolation

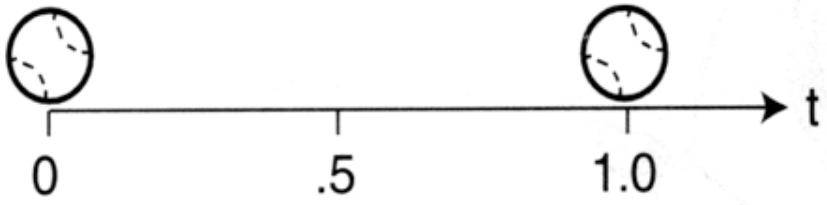
- Nowadays, the in-between frames are drawn automatically by computers
  - "Averaging" the parameters and attributes contained in keyframes, e.g., shape, position, pose or surface attributes, e.g., colours.
- The techniques for creating the in-betweens are usually interpolations.

A Bouncing Ball

- Suppose we know the extreme positions at 3 instants along the timeline of a ball that bounces on the ground. How should we create an animation to simulate the motion of the ball?

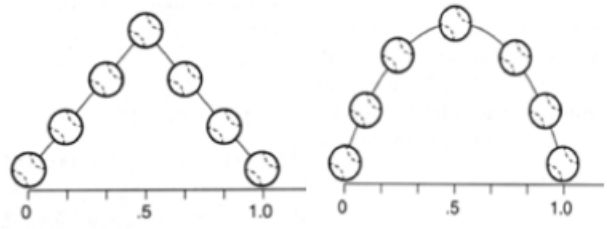


- The three extreme positions are used as keyframes – they characterise the path of the ball.
- What shape should the path have?

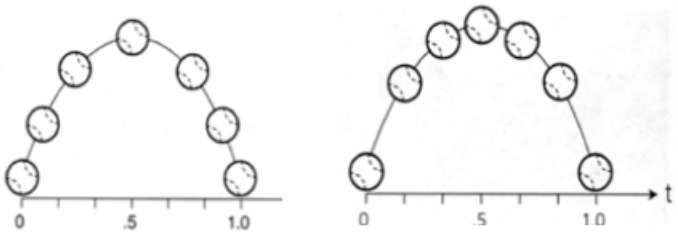


- A path contains location (we ignore the spin of the ball) and timing information.

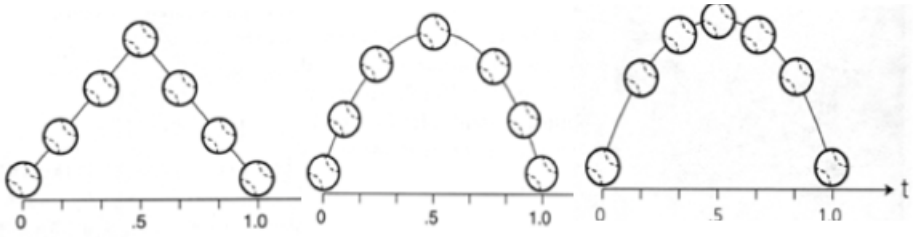
- Suppose we draw the ball at equal time intervals, which path is more realistic? Why?



Speed of the Ball

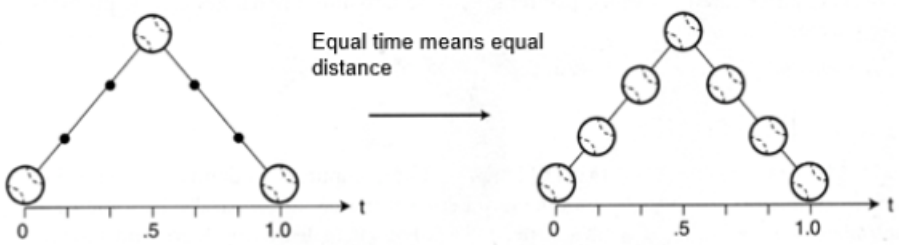


- Some factors to consider:
  - Requirements on motion accuracy (For example, is the ball an insignificant element of a bigger system, e.g., particle system?);
  - Easiness of obtain the data of the true motion of an object;
  - Computing power, etc.
- Depending on applications, any one of the following paths might be acceptable.

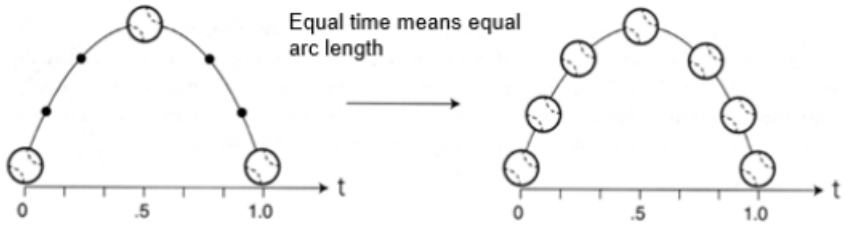


Linear Interpolation

- Linear interpolation creates inbetween frames by assuming an object travels *equal distance (arc length) in equal time* along straight or curved paths.
- In the case straight line path, this implies: (1) constant speed (2) object (abruptly) changes its direction of motion at keyframes.

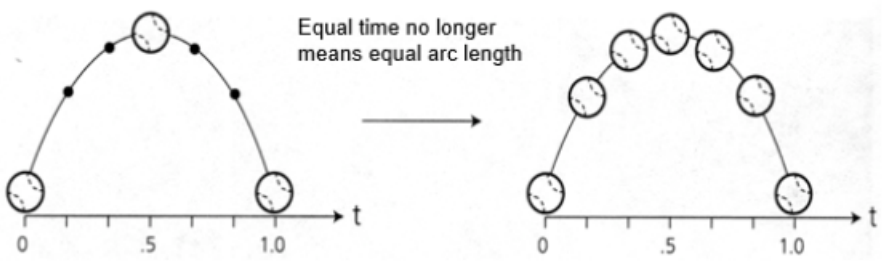


- In the case of *curved* paths, the speed (the distance traveled per unit time) is constant, but direction of travel changes.
- Note: When referring to **both** direction **and** speed, we use the term velocity.
- Result
  - The shape of the path is correct - the parabolic path of projectile under the influence of gravity.
  - But the speed is incorrect. E.g., the ball's speed at top should be close to zero.



Nonlinear Interpolation

- In nonlinear interpolation, an object no longer travels equal distance in equal time.
- The velocity (speed and direction) of an object changes, i.e., it has accelerations and decelerations.



- In animation, these acceleration/decelerations are called, and implemented through, “easing” - “ease-in” and “ease-out” (or “slow-in” and “slow-out”) around a keyframe.
- Notice that the ease-ins and ease-outs are usually not dynamically correct/accurate. But the result is visually acceptable.
- As you may have noticed, in 3DS Max, the nonlinearity of paths are controlled through manipulating the “tangents” at keyframes (Try to use the 3ds curve editor in practical sessions).

Principles of Animation

- Disney Studio developed some principles of animation (starting in the 1930’s) for traditional hand-drawn cartoon character animation:
  - for making good animation
    - Fluid, natural, realistic motion
    - Effective ways of telling the story

    - for training young animators better and faster
- These principles are applicable to all sorts of animation, computer animation included.

12 Animation Principles

- Extremes
- Squash and Stretch
- Anticipation
- Staging
- Ease-In and Ease-out
- Arcs
- Exaggeration
- Timing
- etc

Extremes

- Extremes refer to extreme positions, angles, etc.
- They usually define the poses or actions of a character. E.g. for a jump:
    - the start
    - the lowest crouch
    - the lift-off
    - the highest part
    - the touch-down
    - the lowest follow-through
- Extremes are used to define keyframes.
- In comparison, the frames in between the keyframes (called “inbetweens”) do not introduce a lot new info about an action.

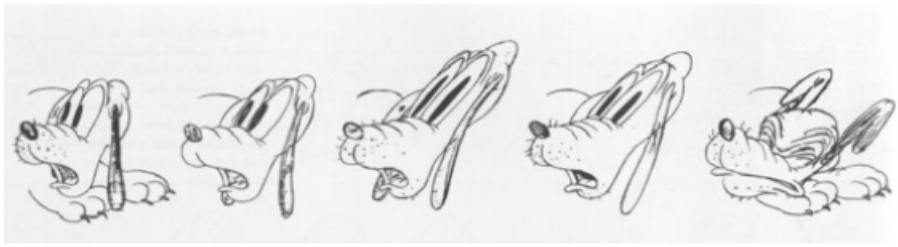
Arcs

- Natural motions tend NOT to be in straight lines, instead should be curved arcs.
    - Just doing straight-line interpolation gives robotic, weird movement.
    - That's why linear interpolation doesn't always work.
- They are also part of physics
    - gravity causes parabolic trajectories;
    - joints cause circular motions;
    - etc.

Squash and Stretch

- Rigid objects look robotic – let them deform to make the motion more natural and fluid.
- Accounts for physics of deformation
  - Communicates to viewer what the object is made of, how heavy it is, etc.
- It might be used for exaggeration of impact force, hurt, pain etc., e.g. banging a face against a wall.
- Volume conservation in deformation: if you squash in one dimension, stretch in another to keep the mass/volume constant.

Information conveyed: material property (elasticity), forces, feelings (pains)



Information conveyed: mass, speed, emotion (fear), etc.

- Stretch also accounts for persistence of vision (after effect of human vision)
  - An image persists in our vision for a while – fast moving objects leave blurred streaks.
  - Without this attribute, we would have problems to watch films/videos.

Anticipation

- Helps to guide the audience's eyes to where and/or what action is about to occur.
  - Signals what is about to happen, and where it is going to happen.
- Often physically necessary, and also indicates how much effort a character is making (by using exageration).
  - The preparation before a motion. E.g. crouching before jumping, pitcher winding up to throw a ball, etc.

Anticipation tells the characters' intention

The intention here is obvious

Staging

- Staging is about guiding the attention of the audience to the focus of the scene - center of interest.
- But remember:
  - Audience can only see one idea at a time.

- Avoid confusing the audience by having two or more things happen at the same time.

- Staging is also about creating contrast.
  - Object of interest needs to be contrasted against rest of scene.
  - Pick strongest and simplest technique. E.g. Still object vs. busy background.

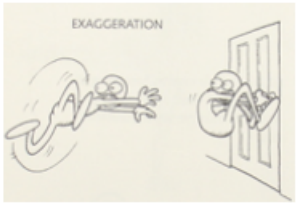
- Staging is also about using background information (object) to enhance the mood and intention of a scene.

Happy

suspenseful

Exaggeration

- Don't worry about being physically accurate: convey the correct psychological impression as effectively as possible.
- Exaggerated motion often created special effects



Secondary Motion

- Secondary motion is the movement that's not part of the main action (e.g., a dog shakes its head), but is physically necessary to support it (e.g., ears flaps as a result of head shaking).
- Animator has to give the audience an impression of reality, or things look rigid and robotic.

Notice how the cape/clothes moves

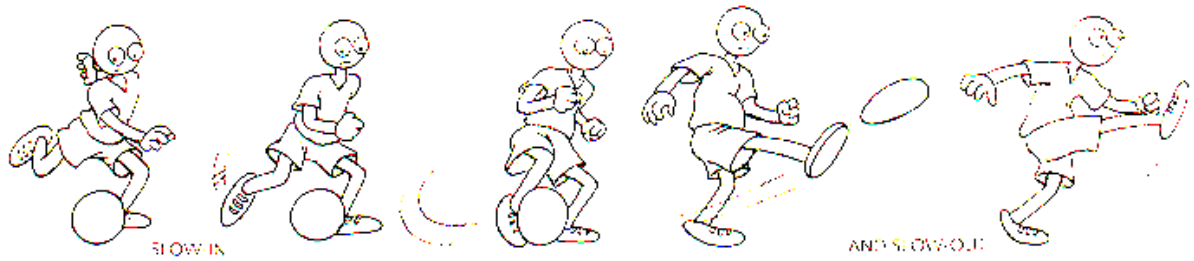
Timing

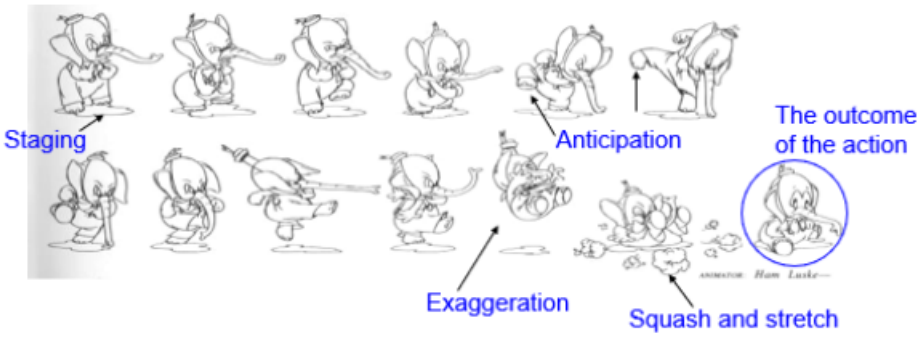
- Timing is about the precise moment and the amount of time that a character spends on an action.
- Tempo (pace) and rhythm
  - A slow tempo expresses seriousness, fatigue, caution, intimacy.

- A lively tempo may express happiness and nervousness.

Slow-in and Slow-out

- As having been motioned, slow-in/out are used as a simulation of physics
  - More physics: objects generally smoothly accelerate and decelerate, depending on mass and forces.
  - First, second, and third order continuity.
- It is also used as a techniques for emphasizing the key frames, the most important or "extreme" poses
  - Character spends more time near those poses, and less time in the transition.
  - Audience gets better understanding of what's going on at the key moments.



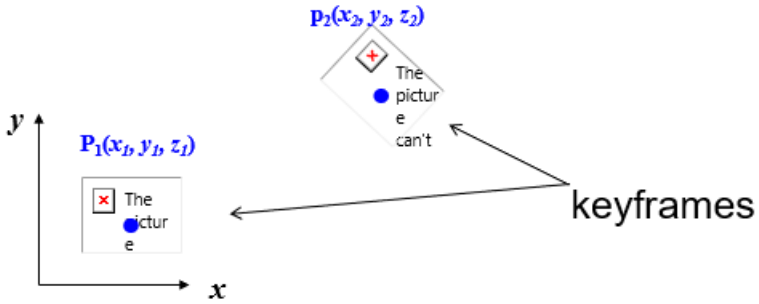


Position and Rotation Interpolation and Transformations

Position and Orientation

Position and Orientation

- If the object to be animated is rigid, an inbetween is defined by a position and a orientation.

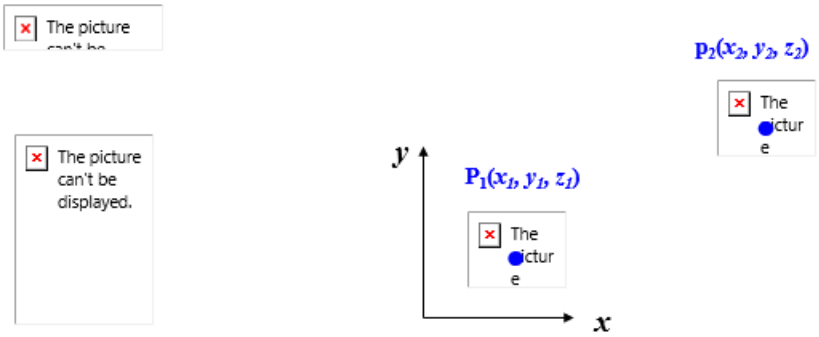


- Therefore the problem of finding inbetweens becomes one calculating the position and orientation at a given instant of time.

Position Interpolation

- Suppose (1) we calculate positions and orientations by linear interpolation;(2) calculation is done separately for position and orientation.
- If we represent a position by its coordinates, i.e., writing a position/point P as

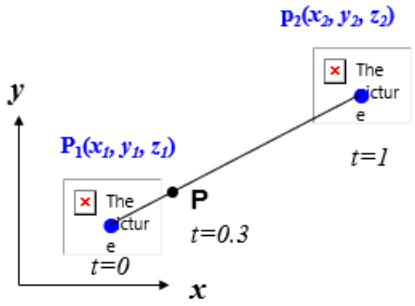
or



- Given two positions P_1 and P_2, the linear interpolation for a position P between them is given by:

  $$\mathbf{P} = (1\text{-}t)\,\mathbf{P_1} + (t)\,\mathbf{P_2}$$

  – Where *t* is a parameter ranging from 0 to 1 and changing t from 0 to 1 will trace out all the points along the line connecting P_1 and P_2.
- Consider a 2D example: Given two points $\mathbf{P_1}(3,8)$ and $\mathbf{P_2}(6,10)$, find out the point where t=0.2.



The picture can't be displayed.