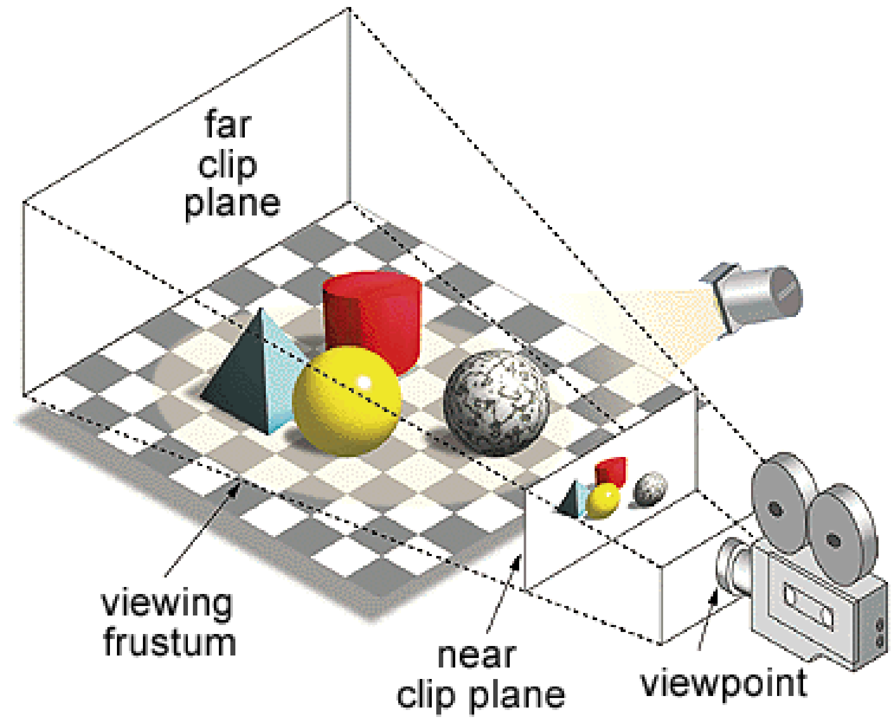
# Scene Projection

- Objects in 3d (world) space are *projected* onto the camera's *near clipping plane*
  - Resulting in a 2d image

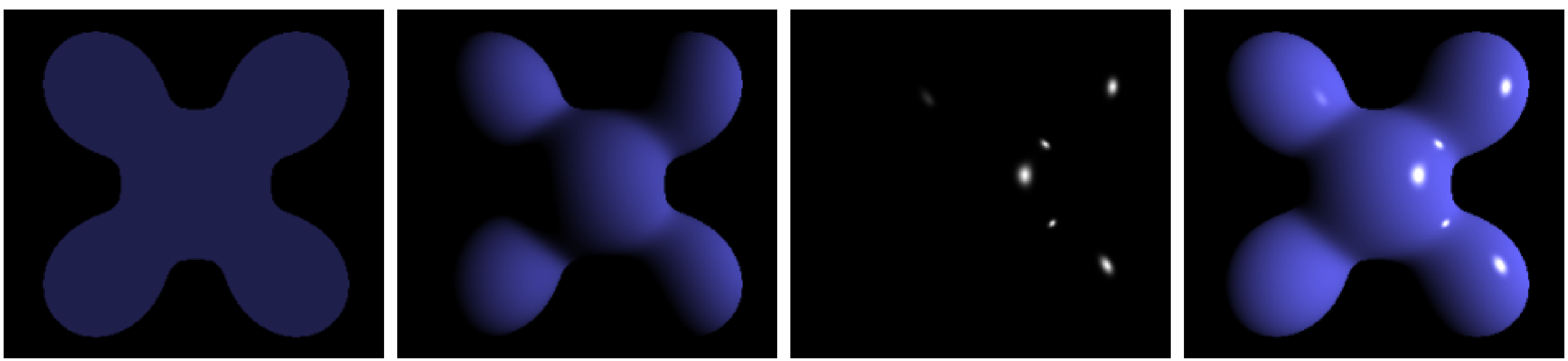



  - For each point on the object
    - Transform into camera space
    - Multiply by *camera projection matrix*
  - Doesn't scale well with continuous points
    - Mesh vertices work much better

*Image from freedictionary.com*

# Lighting Calculation

- *Phong shading* and *Phong reflection*
  - 1973 Ph.D. Thesis, standard simple lighting model
  - Roughly, ambient light is the same everywhere
  - Diffuse light spreads out in all directions after reflection
  - Specular light reflects towards the viewer (creates highlights)

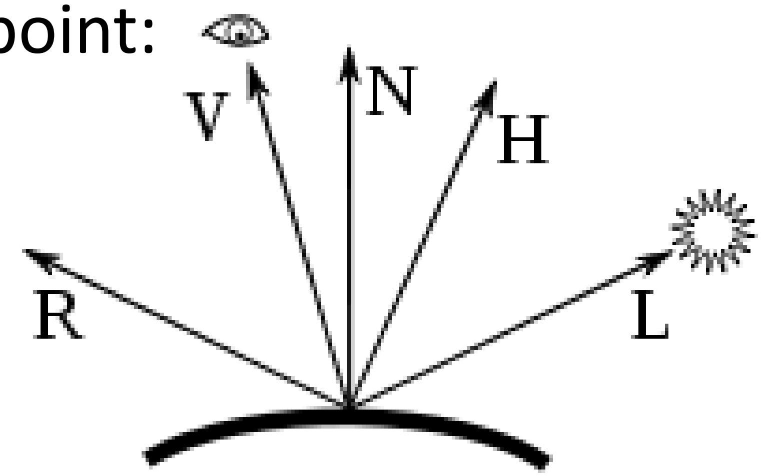


Ambient + Diffuse + Specular = Phong Reflection

*Images from wikipedia.com*

# Lighting Calculation

- Calculate intensity at a surface point:
    - L: vector to a light source
    - N: surface normal
    - V: vector to the viewer
    - R: direction of the light reflection



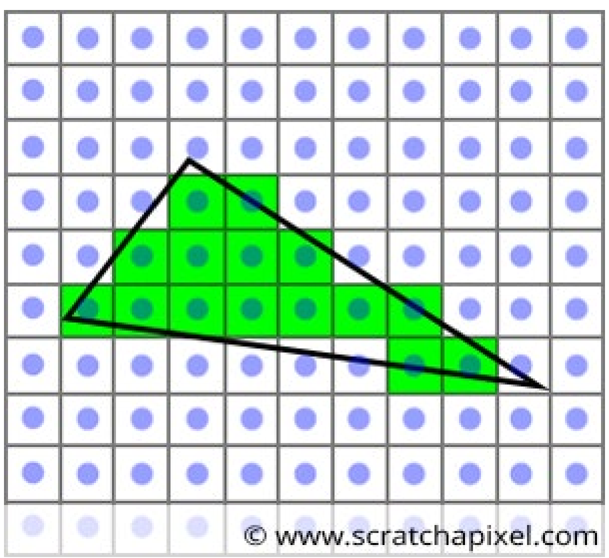

  - Ambient light is just a constant
  - Diffuse light is proportional to L dot N (how it hits the surface)
  - Specular light is proportional to R dot V (how directly it goes towards the viewer)

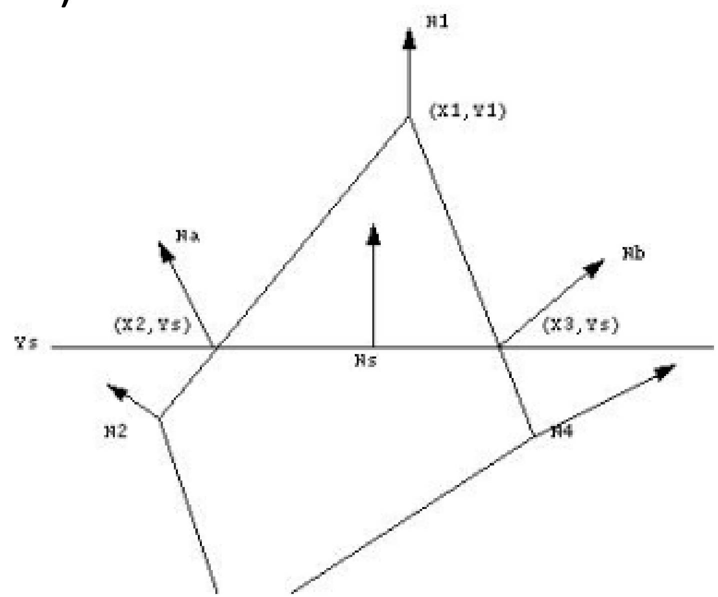
$$I_{\mathrm{p}} = k_{\mathrm{a}} i_{\mathrm{a}} + \sum_{m \in \mathrm{lights}} (k_{\mathrm{d}} (\hat{L}_m \cdot \hat{N}) i_{m,\mathrm{d}} + k_{\mathrm{s}} (\hat{R}_m \cdot \hat{V})^{\alpha} i_{m,\mathrm{s}}).$$

*Images from wikipedia.com*

# Rasterization

- We only care about the discrete pixels on the screen
- Given only the vertices of a polygon
  - For each horizontal *scan line*
    - Interpolate *vertex normals* along polygon edges (Na, Nb)
    - Interpolate across scan line (Ns)




*Images from wikipedia.com*

# What Color?

- Phong shading calculates light *intensity*, which is multiplied by the color of the light and the color of the surface
- *Vertex coloring*
  - Specify the color at each vertex and interpolate
  - Hard to be very precise
- *Texture mapping*
  - "Wrap" an image onto the object like a sticker, specifying the color at each point

# Texture Mapping

- For each vertex in the object
  - Specify a 2d coordinate in the texture image
  - Called U,V coordinates
  - Similar to flattening a globe out into a wall map
    - *UV unwrapping* supported by 3d modeling software

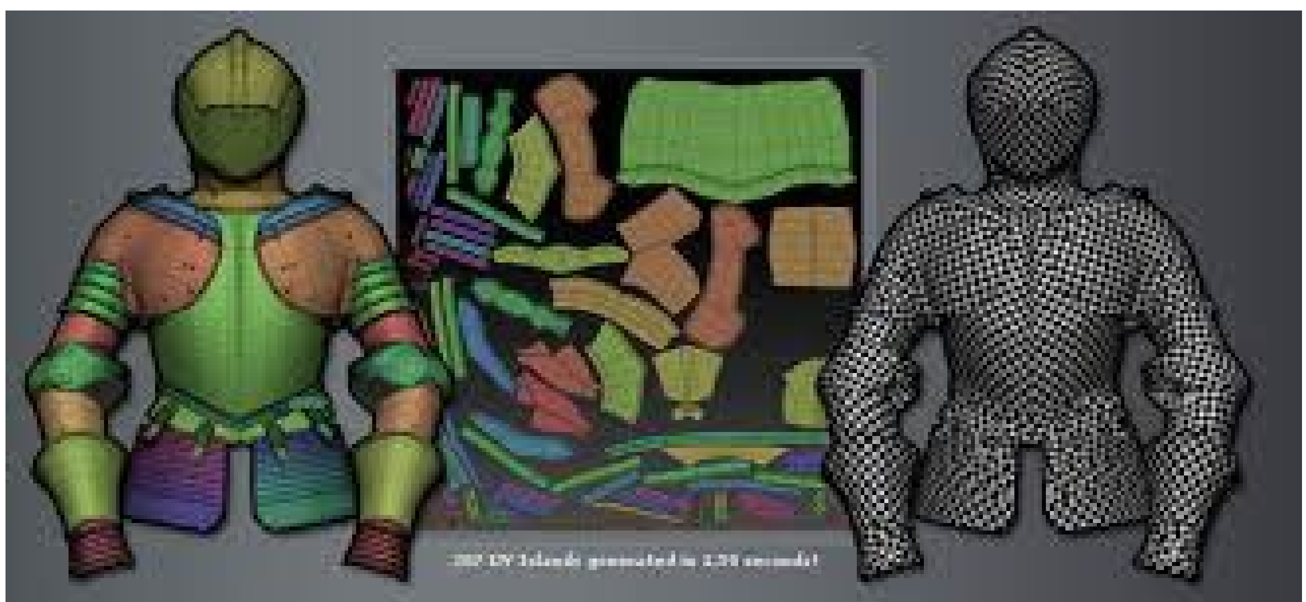


*Image from pixologic*

# Bump Mapping

- Same idea as texture mapping
- But use the image pixel values as normal vectors to create the illusion of surface variation

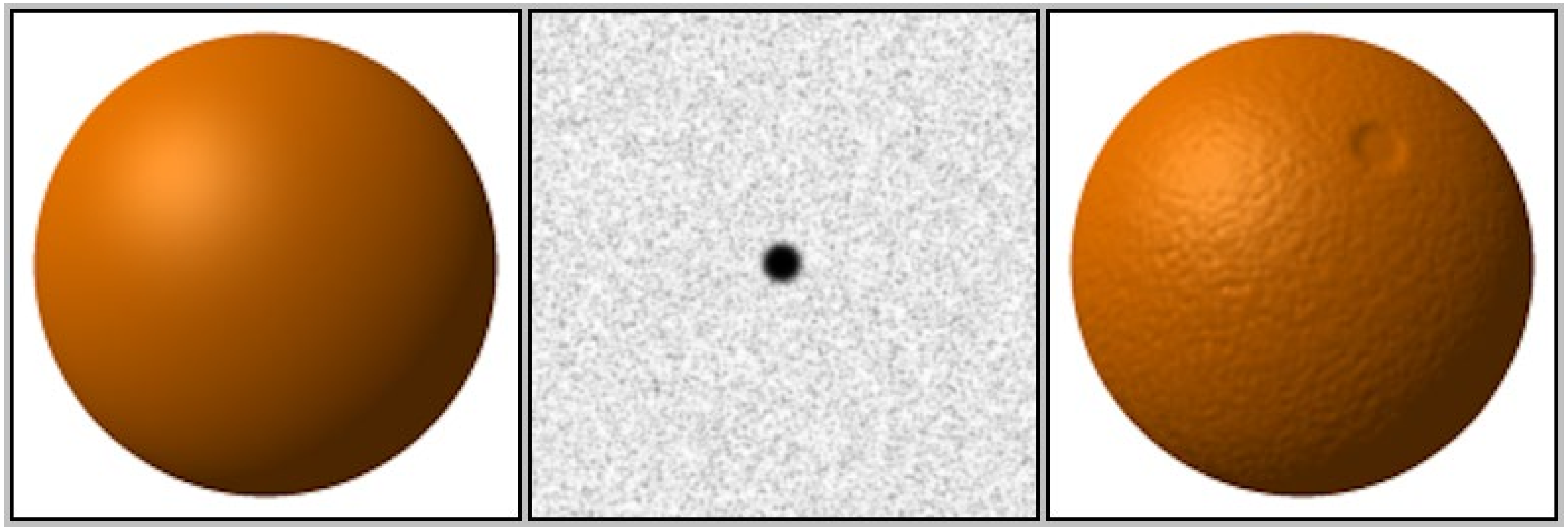
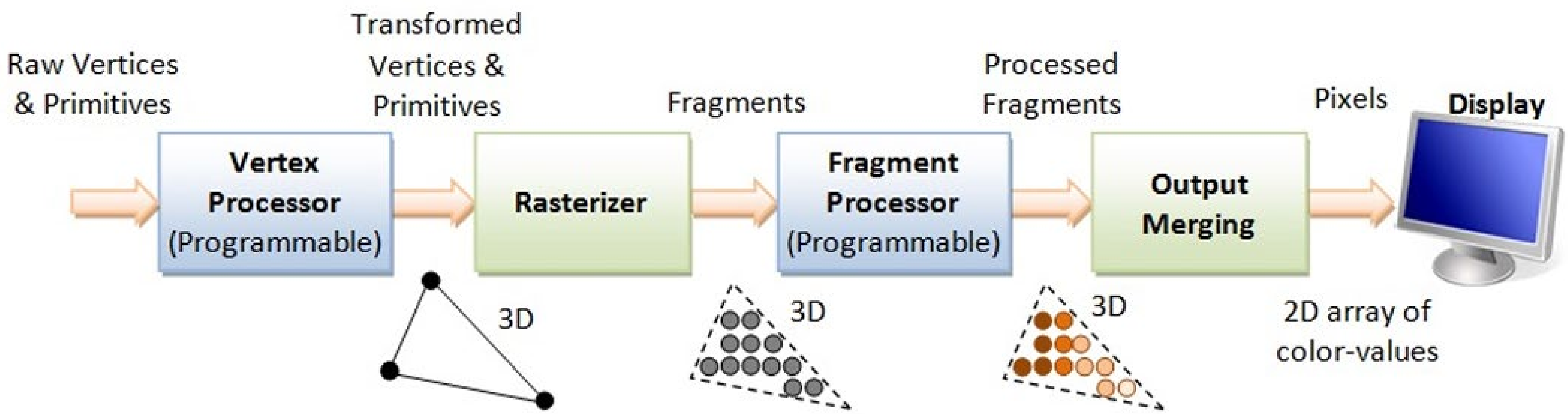


*Image from wikipedia.com*

# Rendering Pipeline

- Original hardware acceleration was rasterizing




Images from: https://www.ntu.edu.sg/index/ehchua/programming/opengl/CG_BasicsTheory.html

# 3D Transforms

- Matrix multiplication!

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{bmatrix}$$

- Matrices can represent transform, rotation and scale

$$\begin{bmatrix} 1 & 0 & 0 & Translation.x \\ 0 & 1 & 0 & Translation.y \\ 0 & 0 & 1 & Translation.z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

$$\begin{bmatrix} Scale.x & 0 & 0 & 0 \\ 0 & Scale.y & 0 & 0 \\ 0 & 0 & Scale.z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

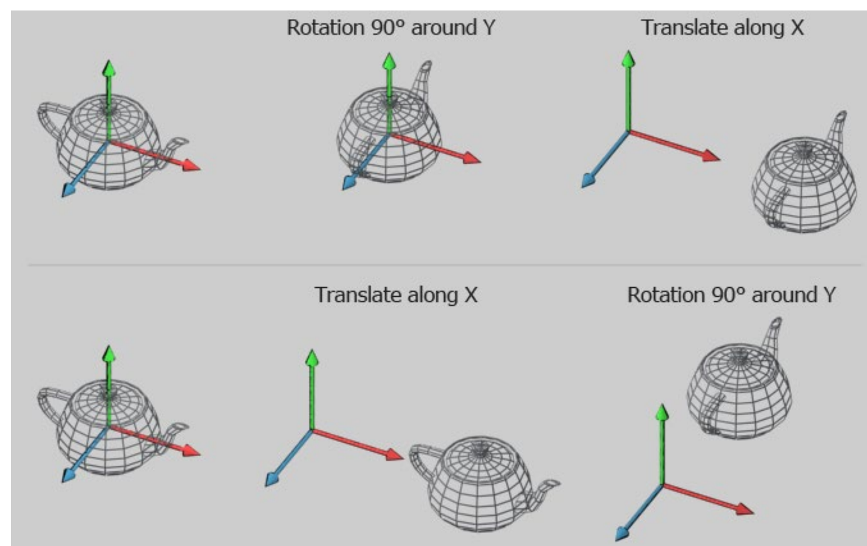
$$\begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

Images from: http://www.codinglabs.net/article_world_view_projection_matrix.aspx

# Model Space vs. World Space

- The vertices of the kettle are specified in *model space*
  - Distance from the origin of the kettle
- The kettle rotation is also in model space
  - Spins on its own axis no matter where it is in the world
  - (The top example is right)
- Matrix multiplication is *not commutative*
  - T*R*V not R*T*V
- But it is *associative*
  - Precalc T*R for all V




Images from: http://www.codinglabs.net/article_world_view_projection_matrix.aspx

# 3D Transforms

- Matrix multiplication is *associative*
  - You can pre-multiply any number of transforms

$$\begin{bmatrix} 1 & 0 & 0 & 1.5 \\ 0 & 1 & 0 & 1.0 \\ 0 & 0 & 1 & 1.5 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(180) & -\sin(180) & 0 \\ 0 & \sin(180) & \cos(180) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} \cos(90) & 0 & \sin(90) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(90) & 0 & \cos(90) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 1.5 \\ 0 & -1 & 0 & 1.0 \\ 1 & 0 & 0 & 1.5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

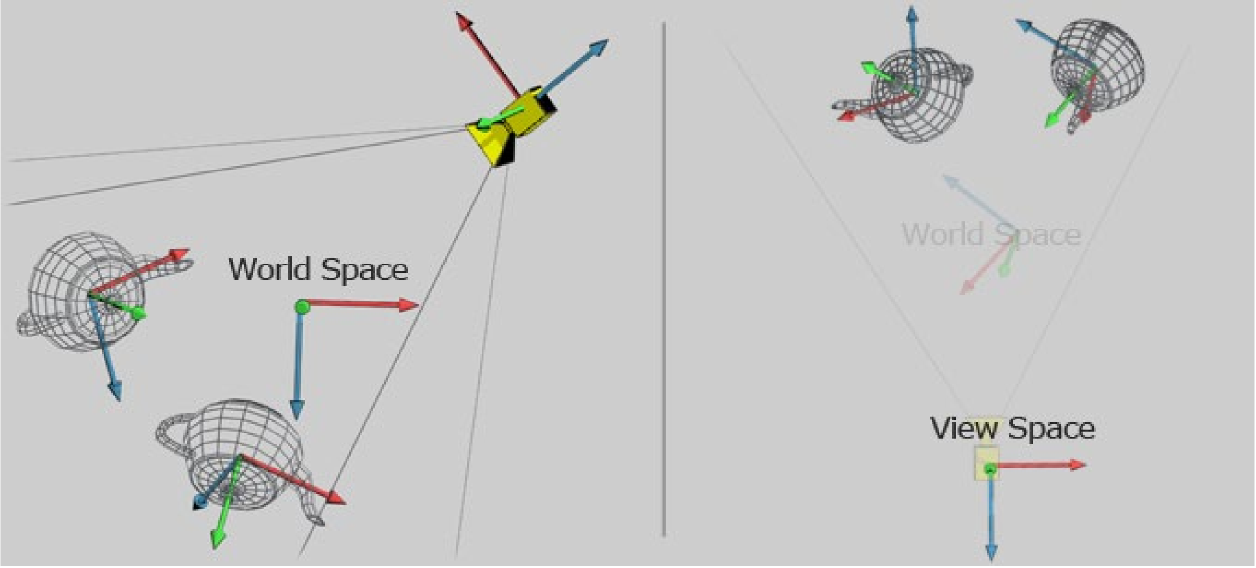
  - Then apply the resulting matrix to all the points in an object

$$\begin{bmatrix} 0 & 0 & 1 & 1.5 \\ 0 & -1 & 0 & 1.0 \\ 1 & 0 & 0 & 1.5 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1.5 \\ 0 \\ 1.5 \\ 1 \end{bmatrix}$$

Images from: http://www.codinglabs.net/article_world_view_projection_matrix.aspx

# World Space vs. View Space

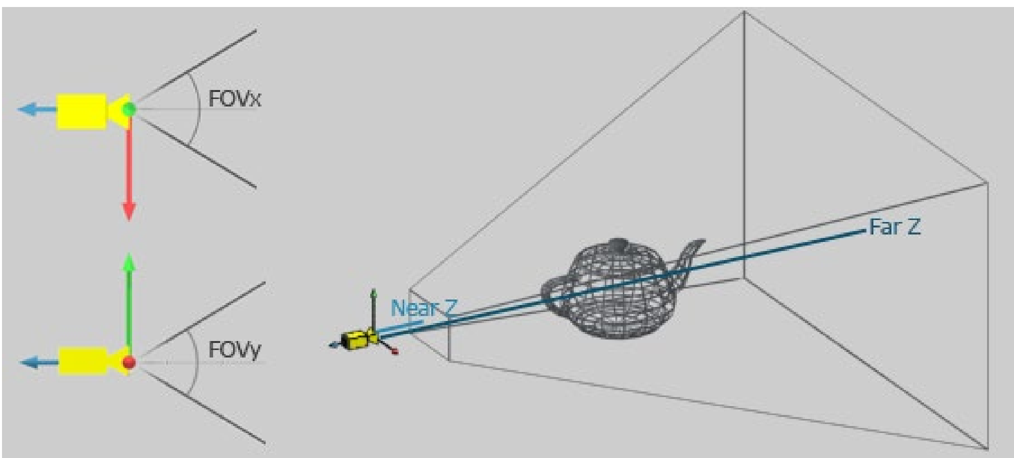
- Objects in world space must be transformed relative to the camera
  - Camera position, orientation is just another matrix
  - Can be pre-multiplied with the model-world transform and applied to all points




Images from: http://www.codinglabs.net/article_world_view_projection_matrix.aspx

# Projection

- From view space, points are *projected* onto the view plane in front of the camera (near Z)
  - Conveniently, projection can be done as another matrix!




$$\begin{bmatrix} \tan^{-1}(\frac{FOVx}{2}) & 0 & 0 & 0 \\ 0 & \tan^{-1}(\frac{FOVy}{2}) & 0 & 0 \\ 0 & 0 & -\frac{Z_{far}+Z_{near}}{Z_{far}-Z_{near}} & -\frac{2\,(Z_{near}\,Z_{far})}{Z_{far}-Z_{near}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Images from: http://www.codinglabs.net/article_world_view_projection_matrix.aspx

# Graphics Algorithms

- Finding more efficient, scalable ways to do realistic rendering
  - In real-time, for games
- Fun problem-solving domain

# Culling

- Only draw polygons that the player can see
  - Too far away (trivial distance culling)
  - Outside view frustrum
  - Blocked by another object

- Naïve approach: project all vertices, only display the ones that fall within the view plane
  - Inefficient, scales poorly

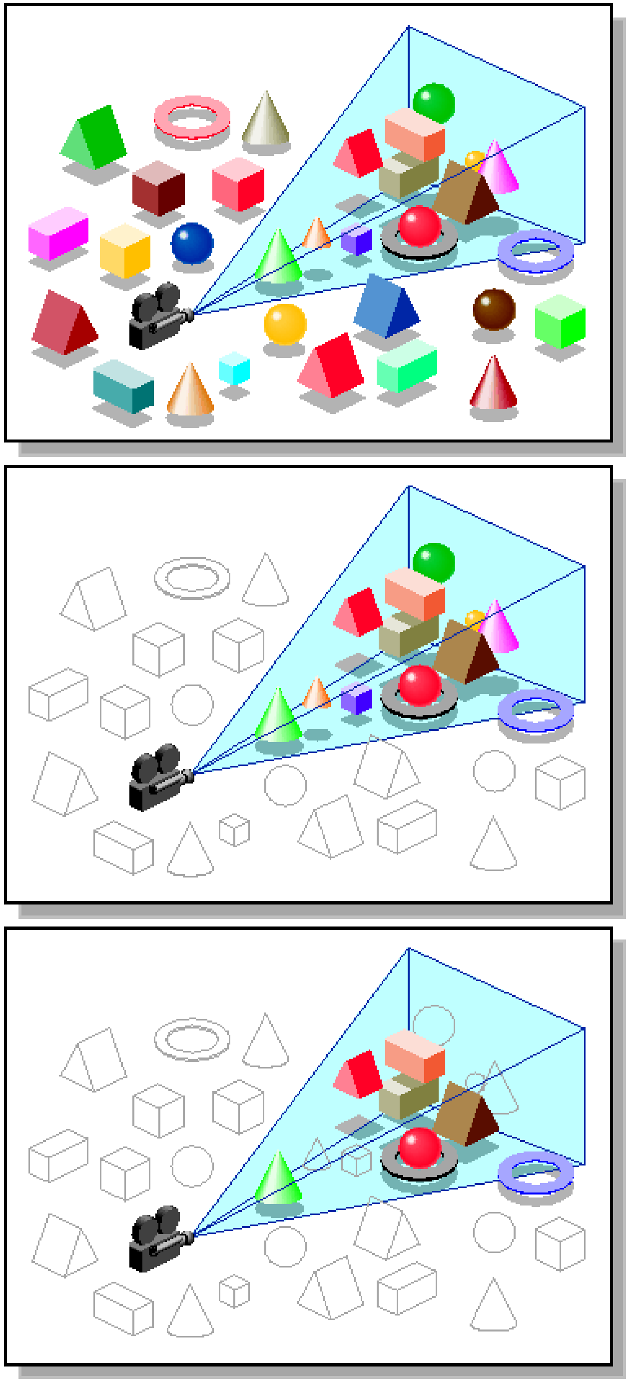


Image from https://techpubs.jurassic.nl/manuals/0640/developer/Optimizer_PG/sgi_html/ch05.html

# Culling

- Spatial partitioning
  - Octrees
    - Recursively divide space into eight cubes
    - If a node is outside the frustrum, so are all its children
    - Useful in visibility, line-of-sight, collision, awareness, etc…

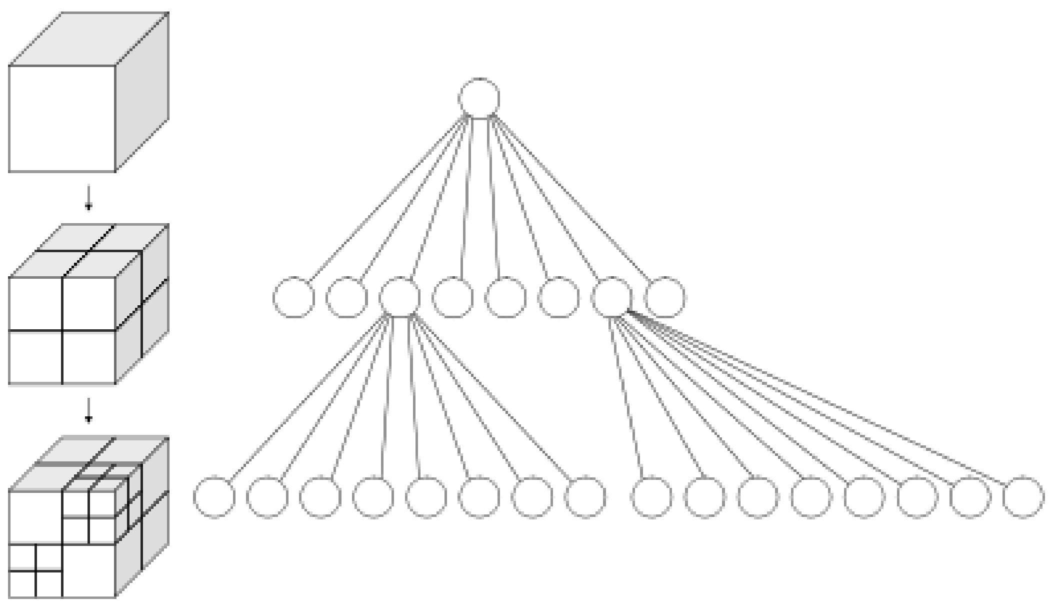


Image from Wikipedia: http://en.wikipedia.org/wiki/Octree

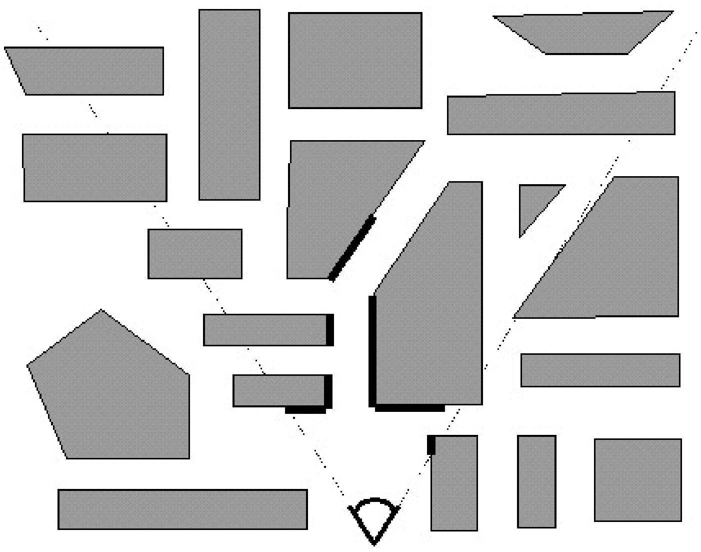
# Occlusion

- Z-buffer
  - Draw everything
  - Also write the distance from camera (depth) for each pixel
    - Need a screen-sized buffer to hold the distance values
  - Only draw if the new pixel is closer than the old

- Inefficient in space and time

# Occlusion

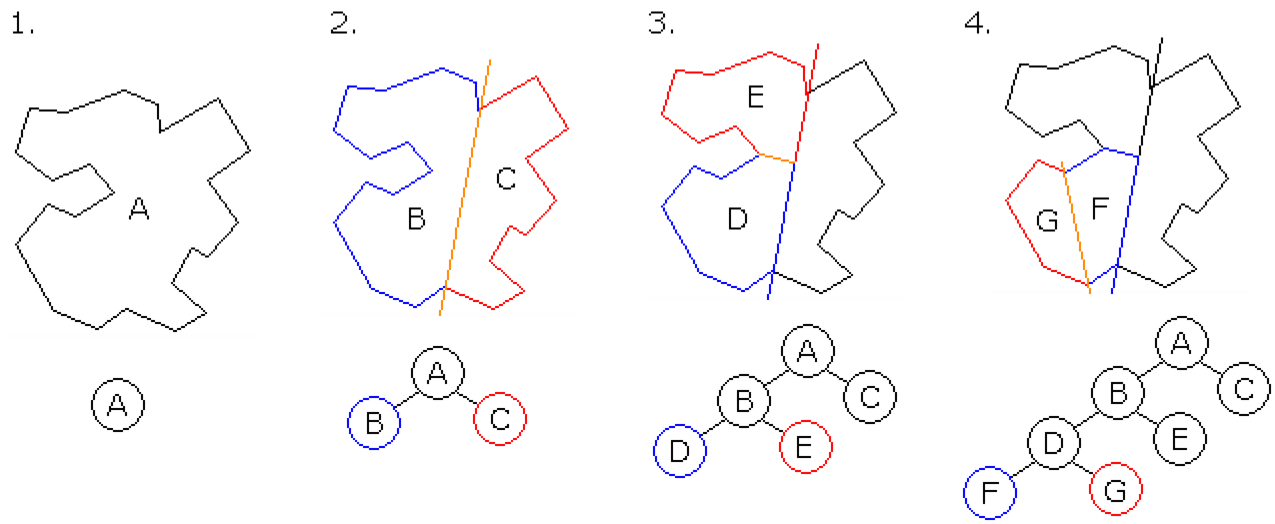
- Painter's algorithm:
  - Draw back-to-front from viewpoint, "painting over"
  - Need to sort polygons back-to-front
- Binary Space Partitioning (BSP trees)
  - Provides fast, reliable back-to-front ordering from any position in the scene (linear time)
  - Product of much research starting in 1969
  - Popularized by John Carmack in DOOM and Quake



Summary of Michael Abrash' article at: http://www.bluesnews.com/abrash/chap64.shtml

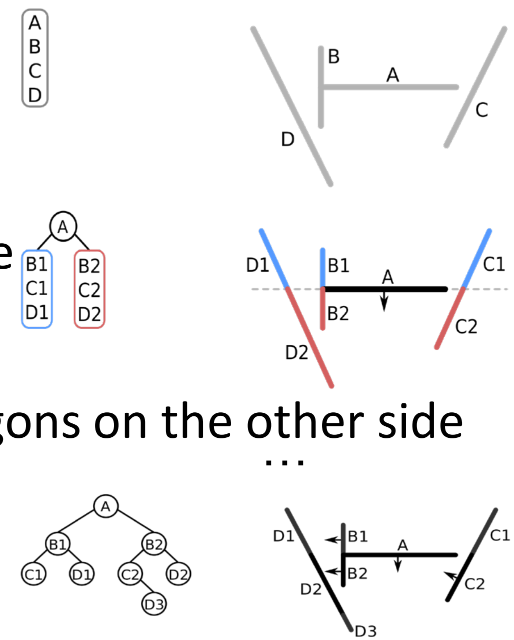
# Binary Space Partitioning

- Recursively subdivide space into two subspaces, storing them in a binary tree
  - If any node is not visible, neither are its children



# Binary Space Partitioning

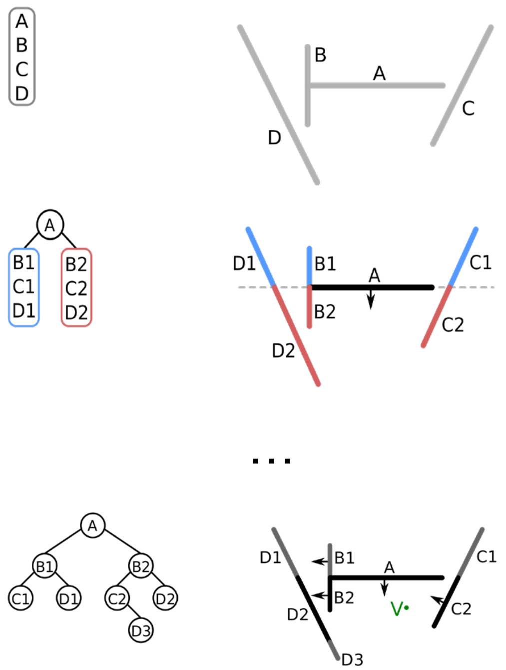
- More specifically, divide each subspace by a *hyperplane*
  - Plane corresponds to walls in the game
  - Plane divides all polygons in the scene
    - e.g. D -> D1 and D2
    - Left child is all polygons behind the plane
    - Right child is all polygons in front of the plane
  - Critical property:
    - From either side of a plane (e.g. A), the polygons on the other side can **never** occlude the polygons on this side
    - Can safely draw them first



# Binary Space Partitioning

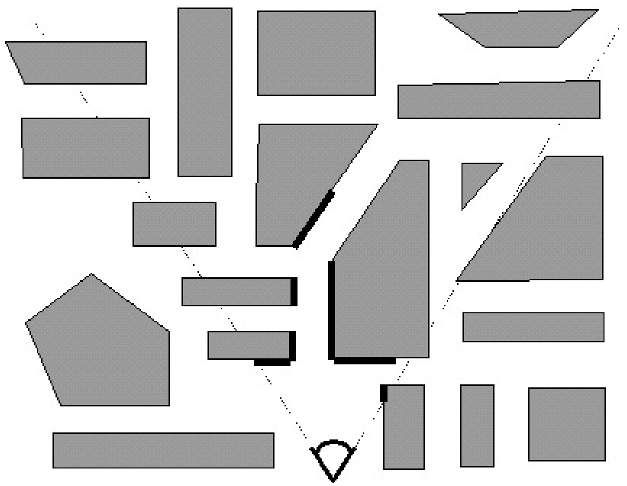
- Traversal algorithm
  - Render child node on the other side
  - Render this node
  - Render child node on this side

- Example result from position V:
  - D1, B1, C1, A, D2, B2, C2, D3



# Eliminating Overdraw

- Still a lot of polygons in the view frustum
  - Tons of overdraw
- Draw front-to-back instead, keep track of filled pixels
  - Combine BSP sorting with z-buffer
  - Only store "filled or not" (1 bit) instead of depth
  - Trade-off depends on the cost of rasterization, shading



Summary of Michael Abrash' article at: http://www.bluesnews.com/abrash/chap64.shtml

# Eliminating Overdraw

- *Potentially Visible Set (PVS):* Pre-calculate from every node which other nodes can be seen and store as lists. Size concern (several MB) compressed as a bit array with RLE (zero-byte) down to 20kb.
  - With PVS, most nodes are culled up-front in traversal, making the best, average and worst cases much more alike
  - Costly pre-processing, only good for static geometry
  - Combined with z-filling to enable efficient culling of dynamic objects

# Shadows

- Intuitively, enhance the lighting calculation
  - Already calculating intensity contribution from each light
  - Check if that light is blocked by an object by raytracing
  - Can be baked into texture maps for static lights, objects
  - Too expensive for real-time

# Shadows

- Shadow map
  - Pre-render the scene from the light PoV into a depth buffer (stores closest distance for each pixel)
  - For each dynamic vertex, project to the light PoV and compare against stored depth
  - If equal, that vertex is lit, otherwise in shadow
  - Limited by resolution of shadow map

# Shadows

- Shadow volumes
  - Get the silhouette of each object in the scene
    - Edges connecting back-facing to front-facing faces
  - Project the silhouette away from the light to create a volume that is in shadow
  - For each vertex, see if it is in a shadow volume or not

# Shadows

- Shadow volumes
  - Real-time acceleration:
    - A point is in shadow if a ray from the camera to that point crosses an *even number* of (convex) shadow volume faces
    - Render entire scene with no lights to get ambient color and depth
      - Depth values are stored in the *stencil buffer*
    - Render all front-facing shadow volume faces into stencil buffer
      - +1 where a shadow face is in front of the visible pixel depth
    - Render all back-facing shadow volume faces into stencil buffer
      - -1 where a shadow face is in front of the visible pixel depth
    - Re-render scene with lighting only where stencil buffer = 0
      - AKA the pixels that are not in shadow
    - Stencil buffer is hardware accelerated for fast update/compare

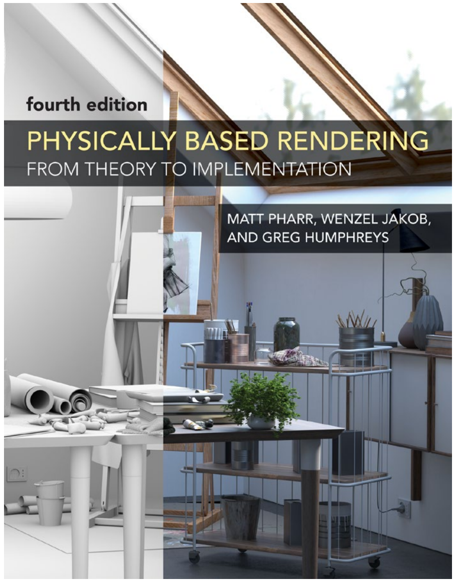
# Current Techniques

- Raytracing available staring with nVidia RTX cards
  - Still want to support lower cost devices, mobile

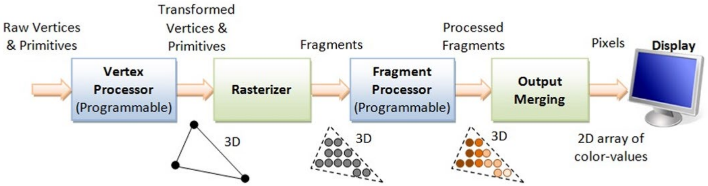
- https://gfxcourses.stanford.edu/cs248/winter22content/media/realtimetechniques/11_modernrast.pdf
  - Soft Shadows
    - Ray tracing vs. PCF
  - Ambient occlusion
  - Reflections
  - Interreflections, subsurface scattering

# Further Reading

- Physically Based Rendering
  - [https://pbrt.org/](https://pbrt.org/)
  - Third edition free online as of 2018
  - Fourth edition released March 2023




- Shaders!
  - Programmable GPU computing units
    - Vertex shaders run on each vertex
    - Fragment shaders run on each rasterized fragment




Images from: https://www.ntu.edu.sg/index/ehchua/programming/opengl/CG_BasicsTheory.html