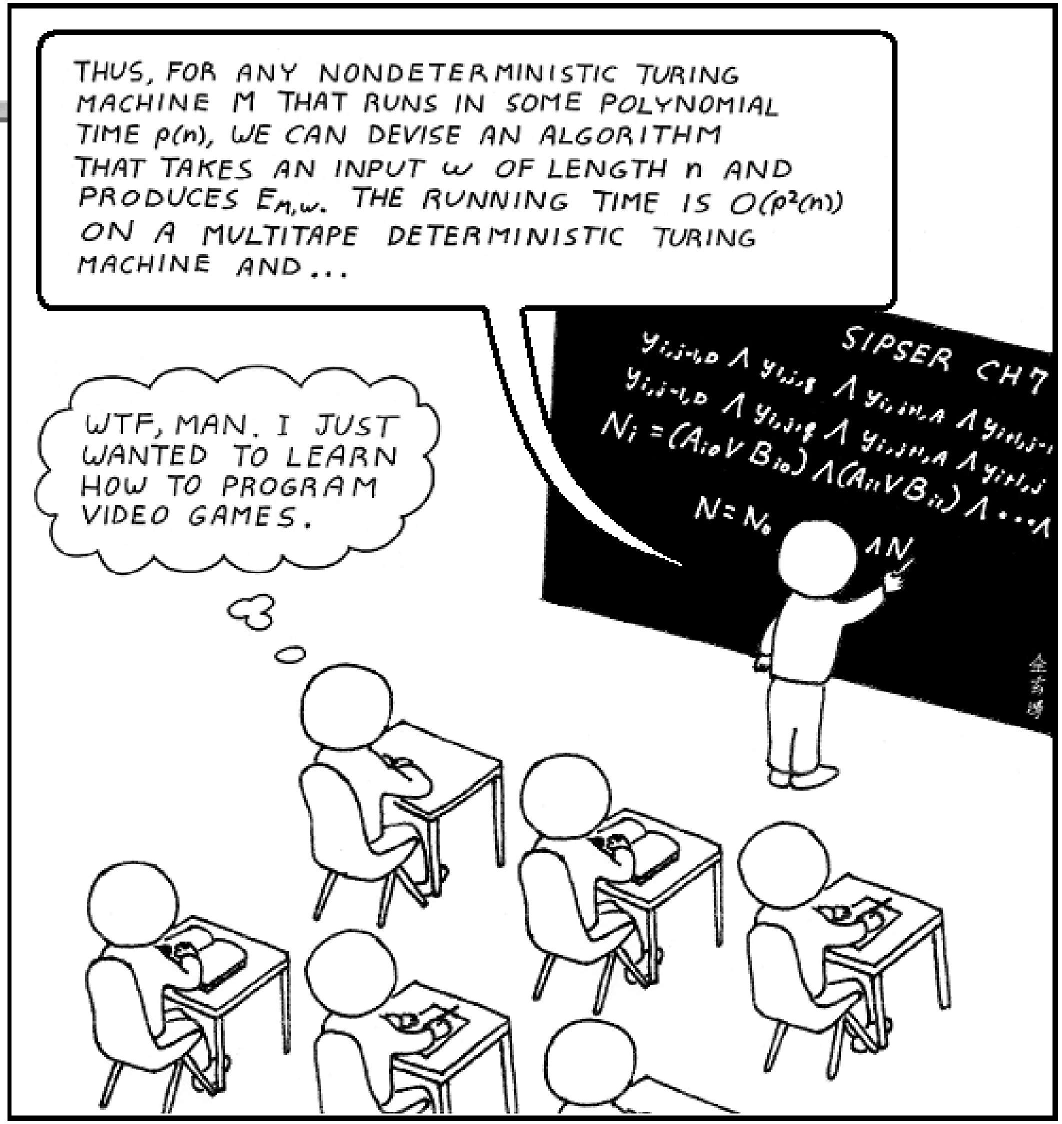
Right of Passage



Chapter 5.1
Graphics

Overview

- Fundamentals
- High-Level Organization
- Rendering Primitives
- Textures
- Lighting
- The Hardware Rendering Pipeline
- Conclusions

Fundamentals

- Frame and Back Buffer
- Visibility and Depth Buffer
- Stencil Buffer
- Triangles
- Vertices
- Coordinate Spaces
- Textures
- Shaders
- Materials

Frame and Back Buffer

- Both hold pixel colors
- Frame buffer is displayed on screen
- Back buffer is just a region of memory
- Image is rendered to the back buffer
  - Half-drawn images are very distracting
- Swapped to the frame buffer
  - May be a swap, or may be a copy
- Back buffer is larger if anti-aliasing
  - Shrink and filter to frame buffer

Visibility and Depth Buffer

- Depth buffer is same size as back buffer
- Holds a depth or "Z" value
  - Often called the "Z buffer"
- Pixels test their depth against existing value
  - If greater, new pixel is further than existing pixel
  - Therefore hidden by existing pixel – rejected
  - Otherwise, is in front, and therefore visible
  - Overwrites value in depth buffer and color in back buffer
- No useful units for the depth value
  - By convention, nearer means lower value
  - Non-linear mapping from world space

Stencil Buffer

- Utility buffers
  - Usually eight bits in size
    - Usually interleaved with 24-bit depth buffer
  - Can write to stencil buffer
  - Can reject pixels based on comparison between existing value and reference
- Many uses for masking and culling

Primitives

Triangles

- Fundamental primitive of pipelines
  - Everything else constructed from them
  - (except lines and point sprites)
- Three points define a plane
- Triangle plane is mapped with data
  - Textures
  - Colors
- “Rasterized” to find pixels to draw

Vertices

- A vertex is a point in space
- Plus other attribute data
  - Colors
  - Surface normal
  - Texture coordinates
  - Whatever data shader programs need
- Triangles use three vertices
  - Vertices shared between adjacent triangles

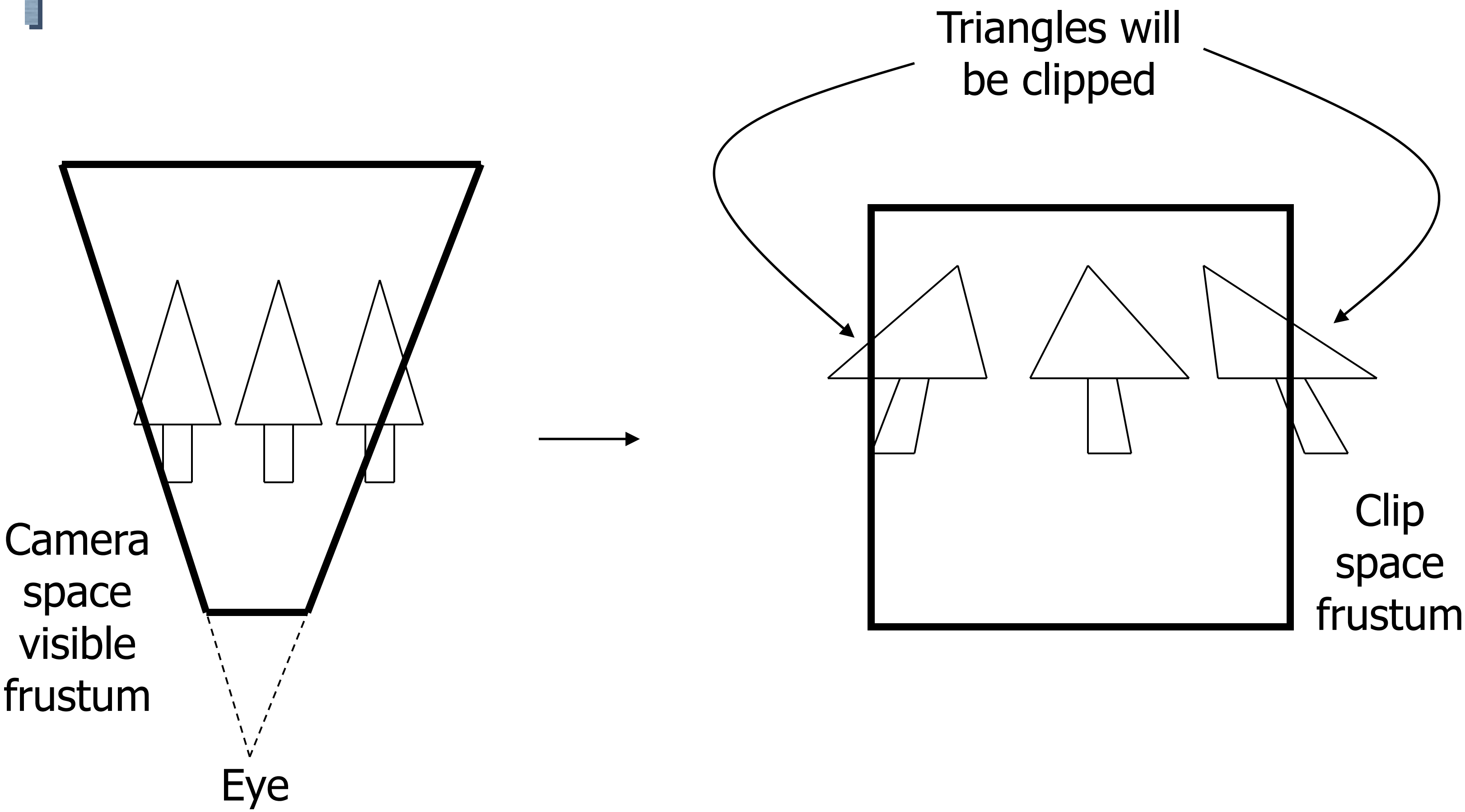
Coordinate Spaces

- World space
  - Arbitrary global game space
- Object space
  - Child of world space
  - Origin at entity's position and orientation
  - Vertex positions and normals stored in this
- Camera space
  - Camera's version of "object" space

Coordinate Spaces (2)

- Clip space
  - Distorted version of camera space
  - Edges of screen make four side planes
  - Near and far planes
    - Needed to control precision of depth buffer
  - Total of six clipping planes
  - Distorted to make a cube in 4D clip space
  - Makes clipping hardware simpler

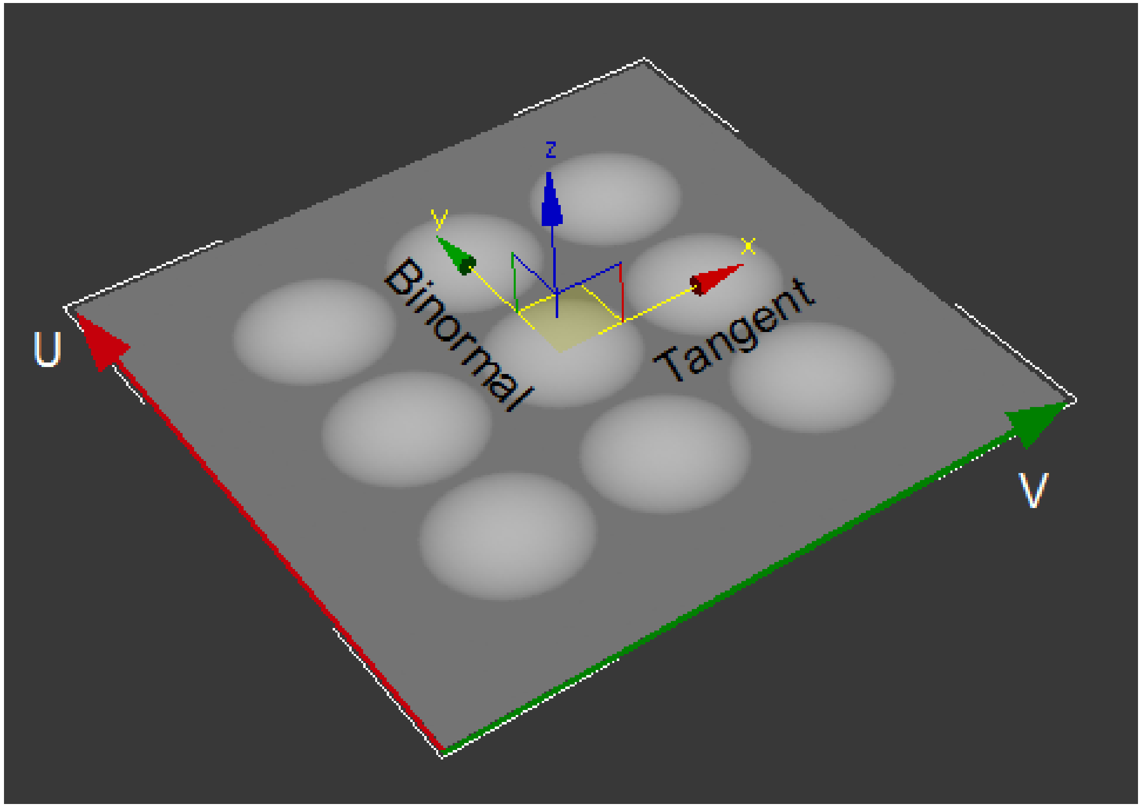
Coordinate Spaces (3)



Coordinate Spaces (4)

- Screen space
  - Clip space vertices projected to screen space
  - Actual pixels, ready for rendering
- Tangent space
  - Defined at each point on surface of mesh
  - Usually smoothly interpolated over surface
  - Normal of surface is one axis
  - "tangent" and "binormal" axes lie along surface
  - Tangent direction is controlled by artist
  - Useful for lighting calculations

More on Tangent Space



Tangent space comes up in shading and 1D texel situations.

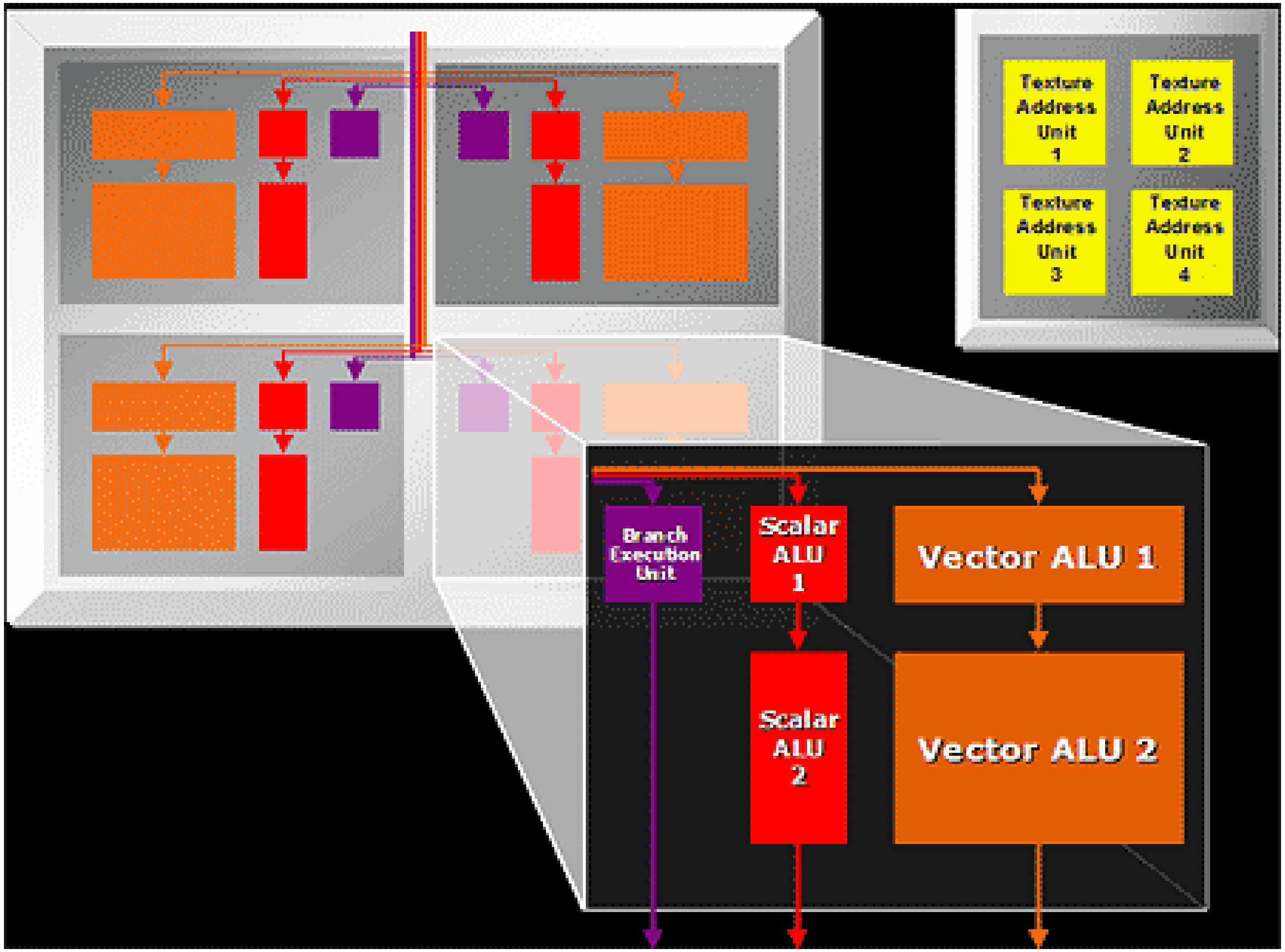
Textures

- Array of texels
  - Same a pixel, but for a texture
  - Nominally R,G,B,A but can mean anything
- 1D, 2D, 3D and “cube map” arrays
  - 2D is by far the most common
  - Basically just a 2D image bitmap
  - Often square and power-of-2 in size
- Cube map - six 2D arrays makes hollow cube
  - Approximates a hollow sphere of texels

Shaders

- A program run at each vertex or pixel
  - Generates pixel colors or vertex positions
- Relatively small programs
  - Usually tens or hundreds of instructions
- Explicit parallelism
  - No direct communication between shaders
  - "Extreme SIMD" programming model
- Hardware capabilities evolving rapidly

Shaders



Texture Address Unit 1
Texture Address Unit 2
Texture Address Unit 3
Texture Address Unit 4
Branch Execution Unit
Scalar ALU 1
Scalar ALU 2
Vector ALU 1
Vector ALU 2

Materials (category)

- Description of how to render a triangle
- Big blob of data and state
  - Vertex and pixel shaders
  - Textures
  - Global variables (lighting)
  - Description of data held in vertices
  - Other pipeline state

High-Level Organization

(Figuring out what to try and draw)

1. Gameplay and Rendering
2. Render Objects (flyweight)
3. Render Instances
4. Meshes
5. Skeletons
6. Volume Partitioning

Gameplay and Rendering

- Rendering speed varies according to scene
  - Some scenes more complex than others
  - Typically 15-60 frames per second
- Gameplay is constant speed
  - Camera view should not change game
  - In multiplayer, each person has a different view, but there is only one shared game
  - 1 update per second (RTS) to thousands (FPS)
- Keep the two as separate as possible!

Render Objects

- Description of renderable object type
  - Mesh data (triangles, vertices)
  - Material data (shaders, textures, etc)
  - Skeleton (+rig) for animation
  - Shared by multiple instances

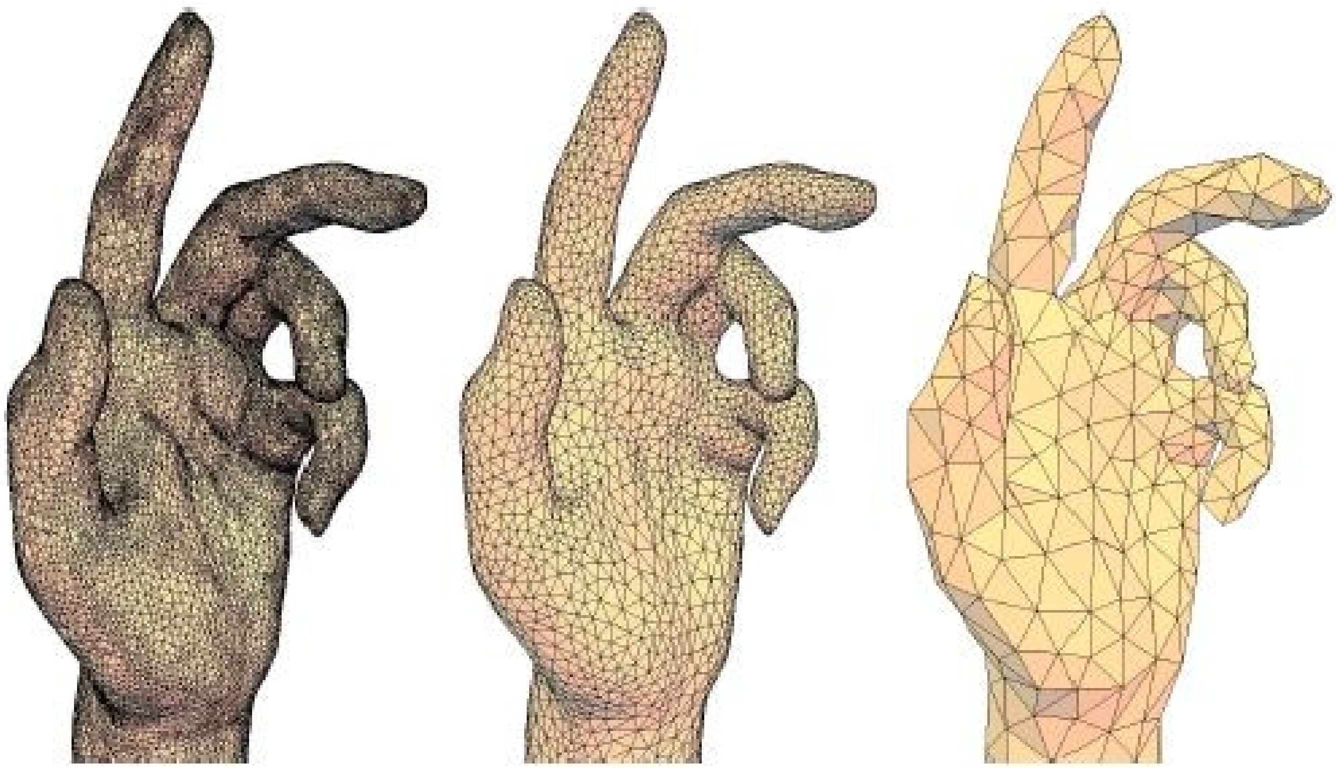
Render Instances

- A single entity in a world
- References a render object
  - Decides what the object looks like
- Position and orientation
- Lighting state
- Animation state

Meshes

- Vertices
- Triangles
- Single material unit
- “Atomic unit of rendering”
  - Not quite atomic, depending on hardware
- Single object may have multiple meshes
  - Each with different shaders, textures, etc

Meshes

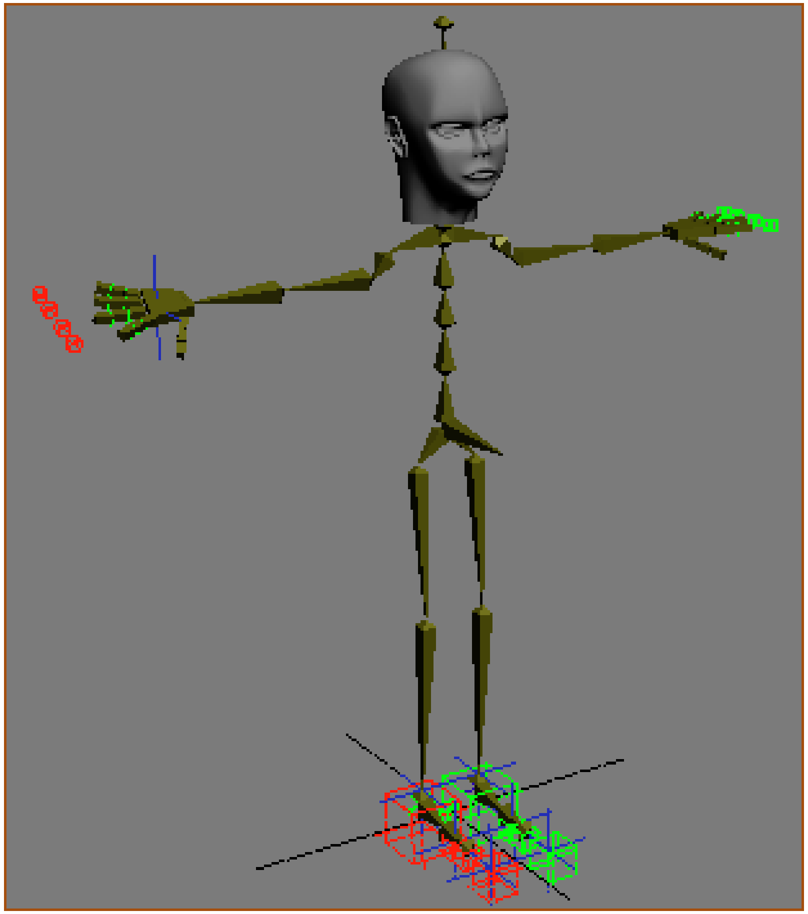


(a) 25,000 vertices. (b) 5,000 vertices. (c) 500 vertices.

Skeletons

- Skeleton is a hierarchy of bones
- Deforms meshes for animation
- Typically one skeleton per object
  - Used to deform multiple meshes
- See "Character Animation" chapter
  - (although deformation is part of rendering)

Skeleton



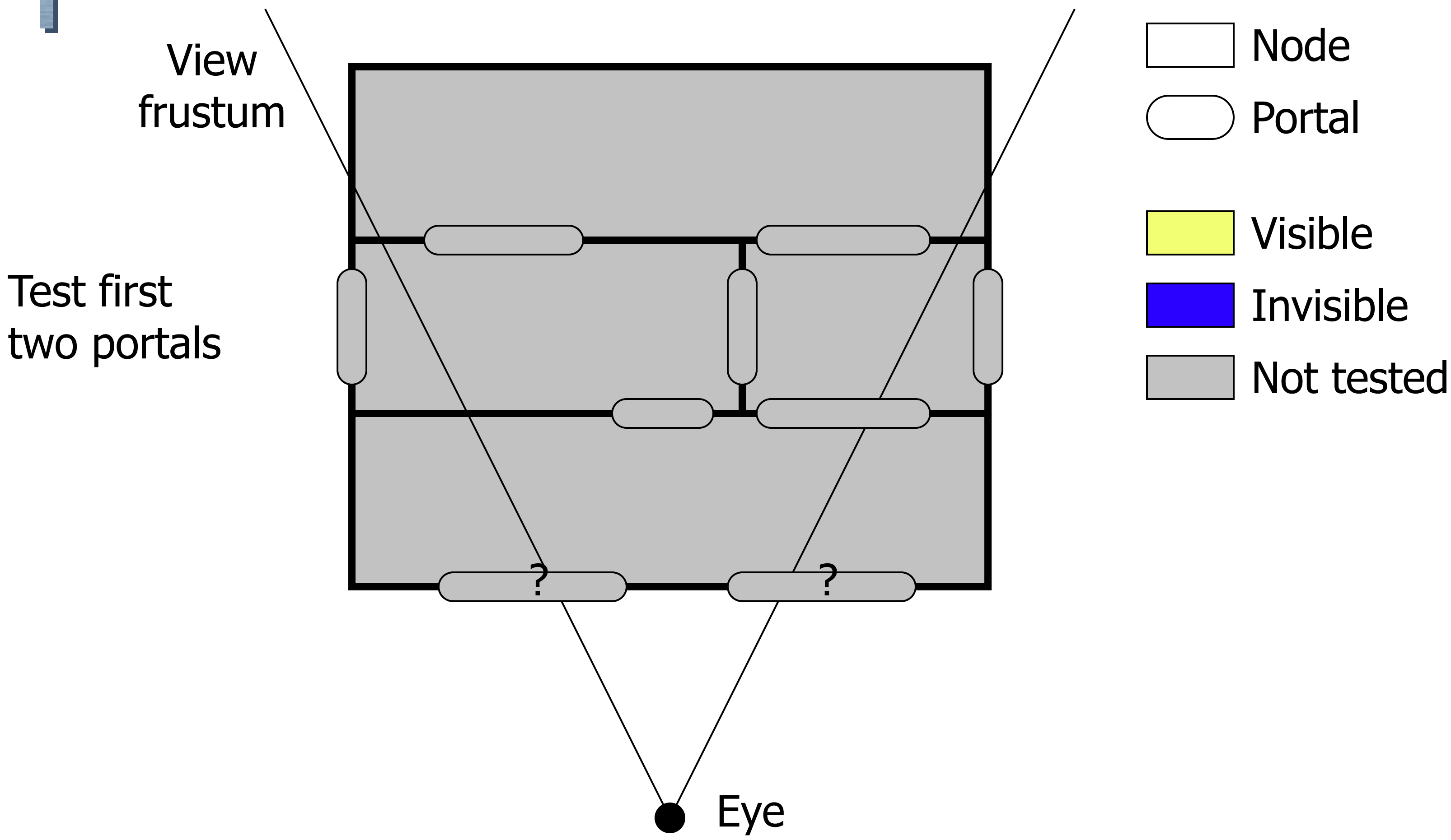
Volume Partitioning

- Cannot draw entire world every frame
  - Lots of objects – far too slow
- Need to decide quickly what is visible
- Partition world into areas
- Decide which areas are visible
- Draw things in each visible area
- Many ways of partitioning the world

Volume Partitioning - Portals

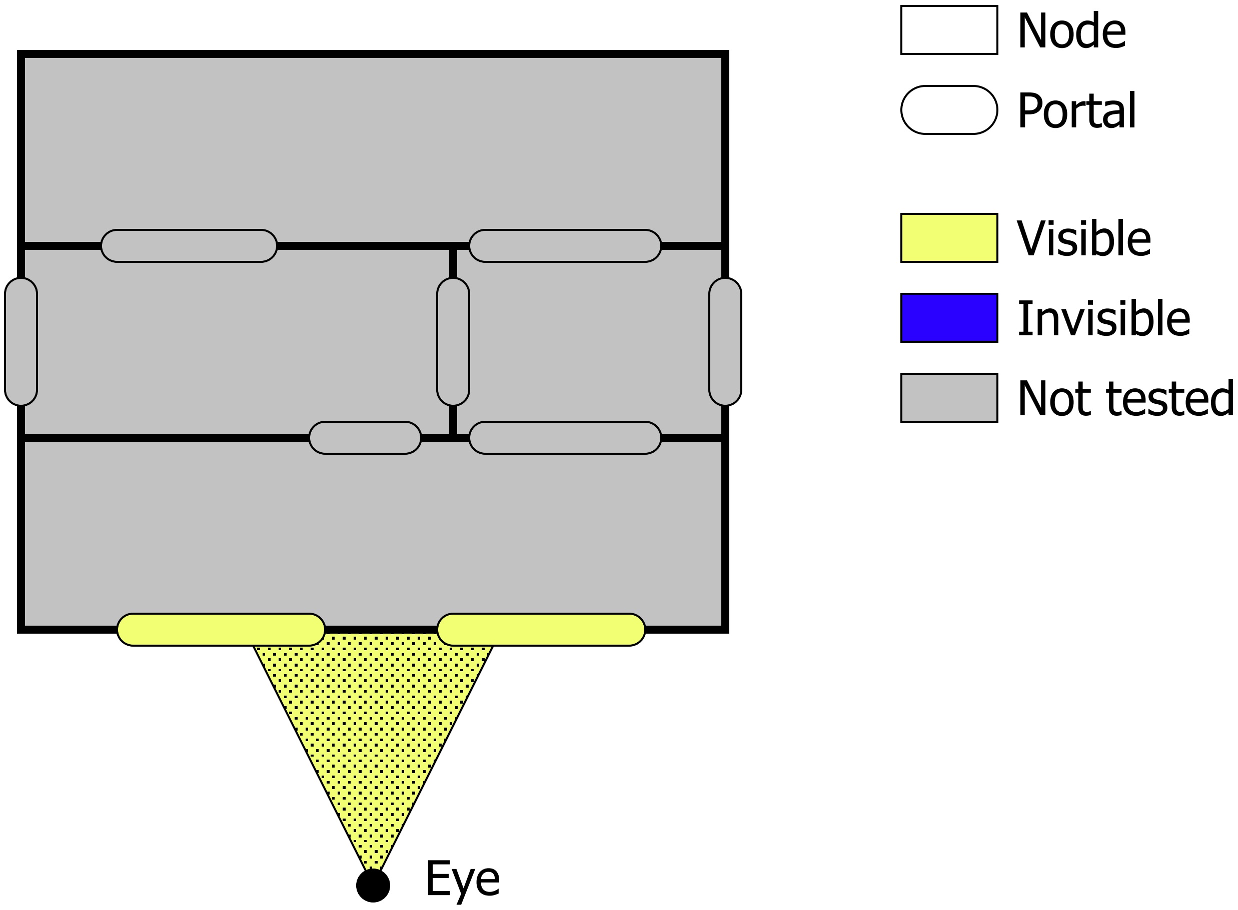
- Nodes joined by portals
  - Usually a polygon, but can be any shape
- See if any portal of node is visible
- If so, draw geometry in node
- See if portals to other nodes are visible
  - Check only against visible portal shape
  - Common to use screen bounding boxes
- Recurse to other nodes

Volume Partitioning – Portals



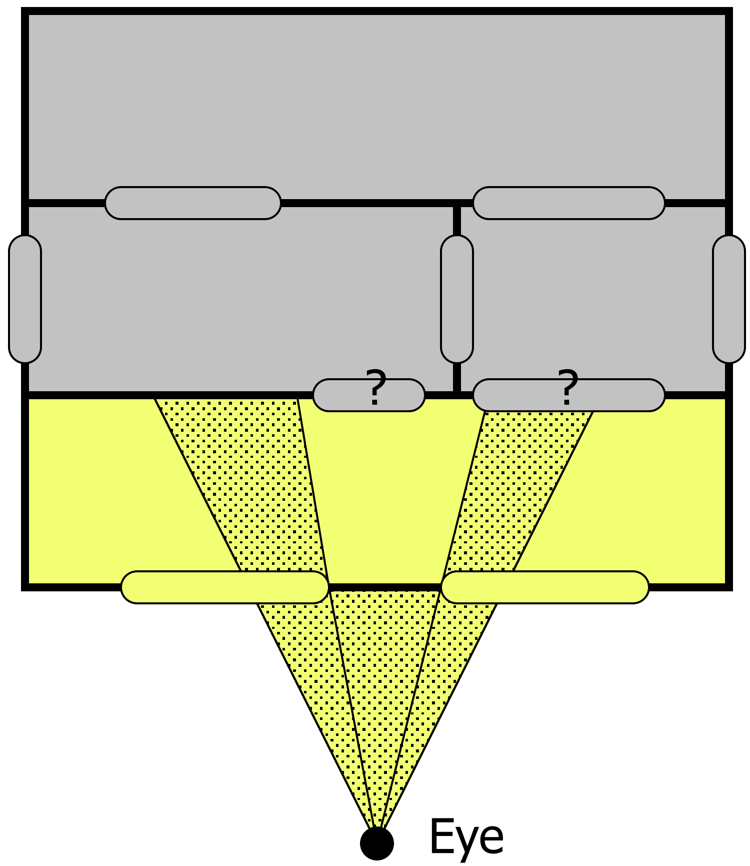
Volume Partitioning – Portals

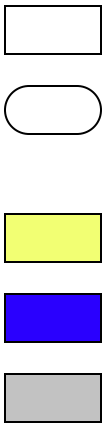
Both visible



Volume Partitioning – Portals

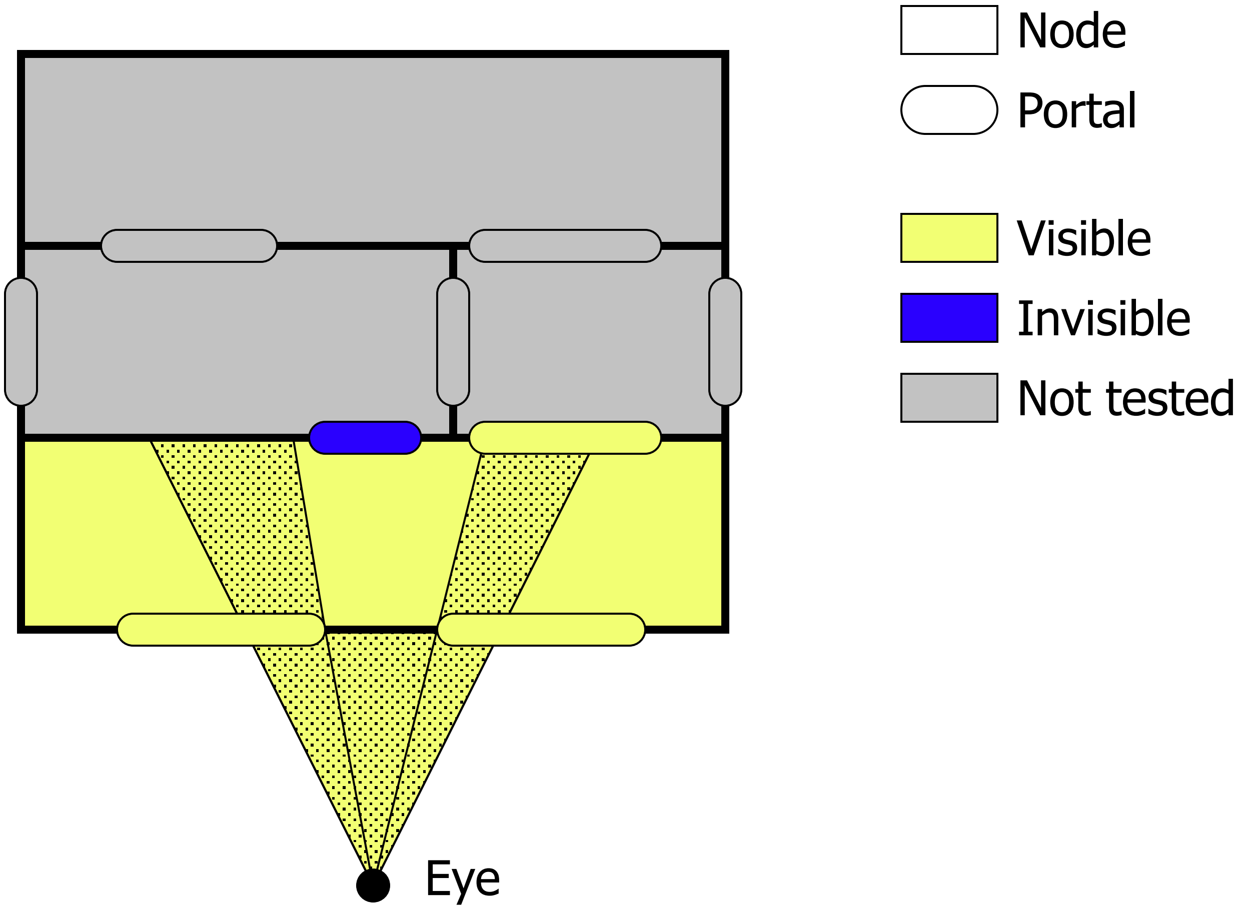
Mark node visible, test all portals going from node





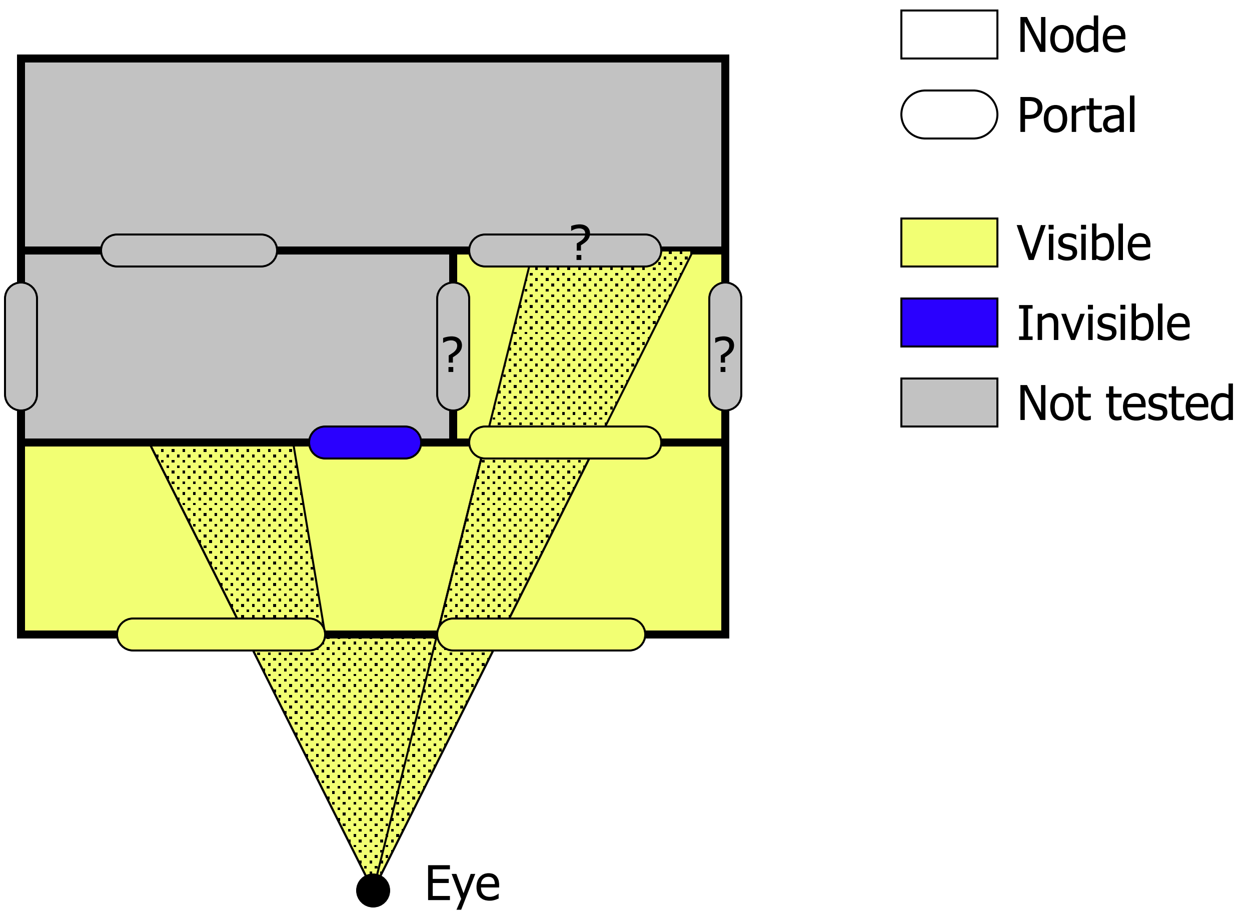
Volume Partitioning – Portals

One portal visible, one invisible



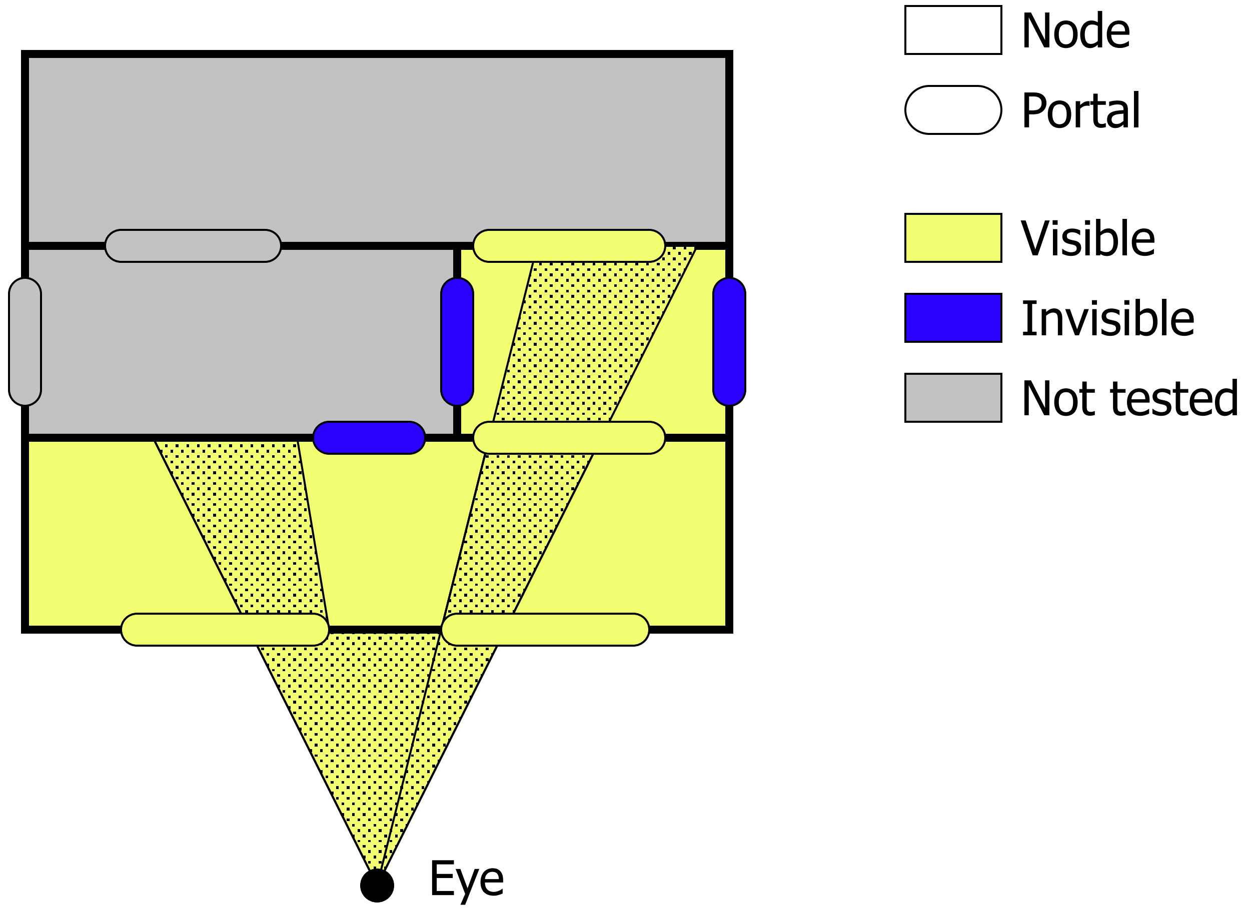
Volume Partitioning – Portals

Mark node as visible, other node not visited at all. Check all portals in visible node



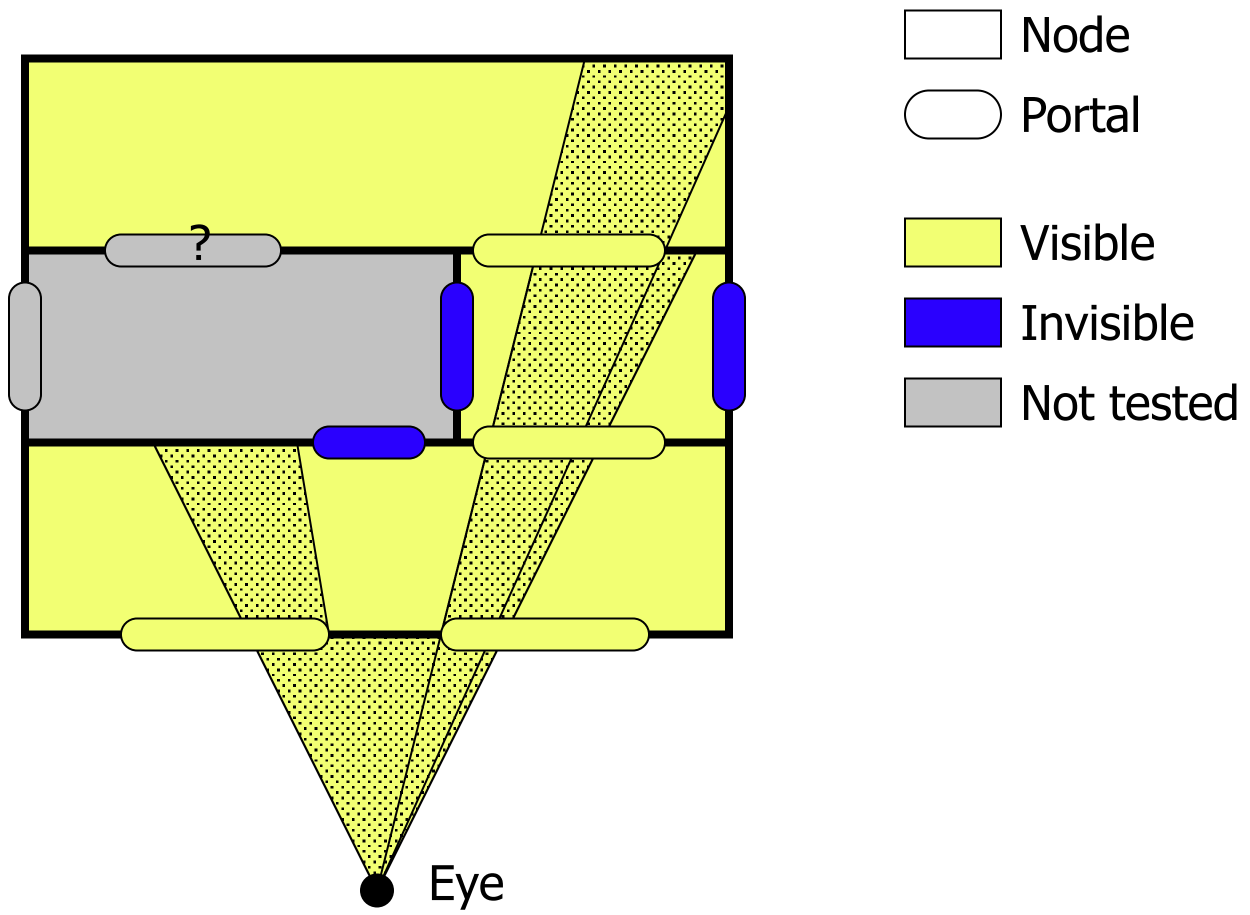
Volume Partitioning – Portals

One visible,
two invisible



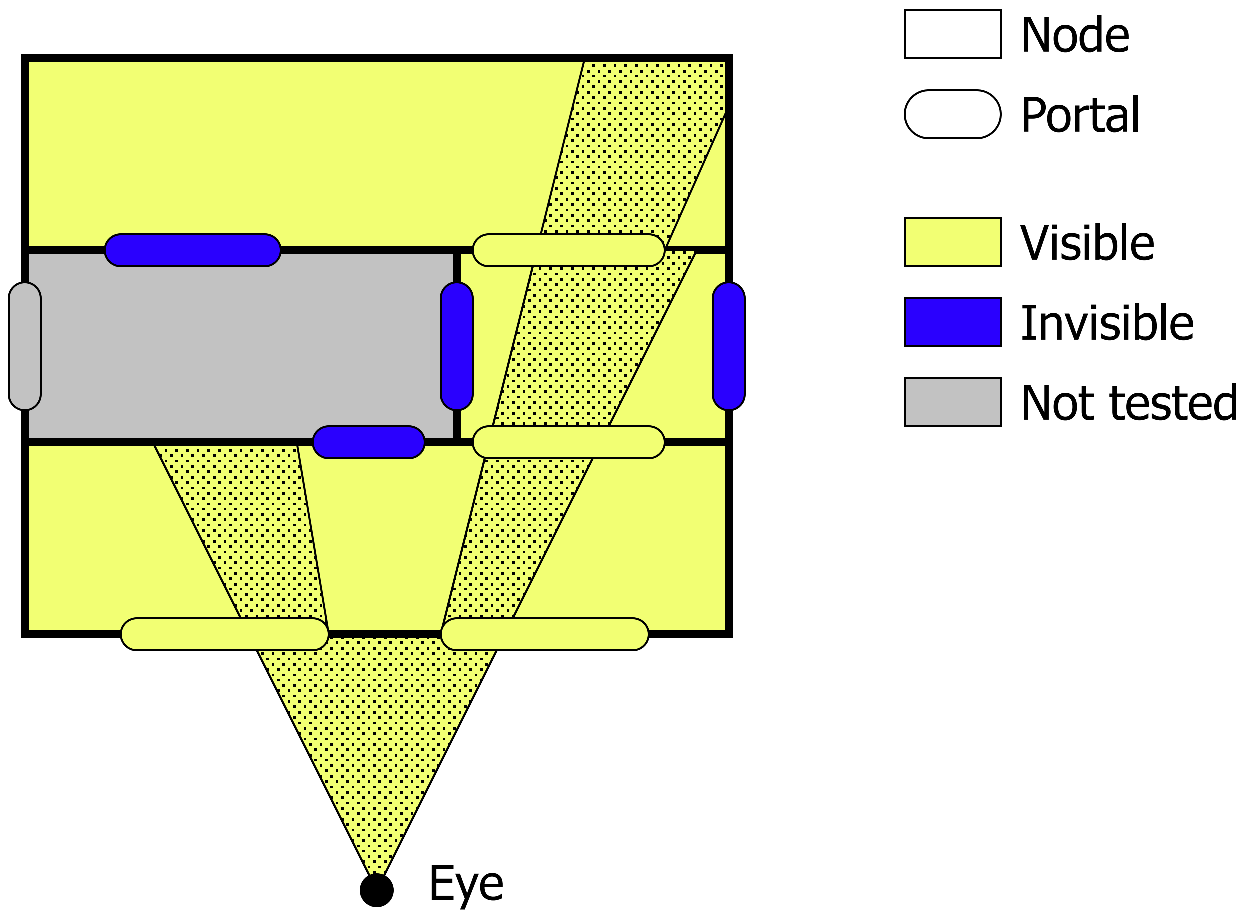
Volume Partitioning – Portals

Mark node as visible, check new node's portals



Volume Partitioning – Portals

One portal invisible.
No more visible nodes or portals to check.
Render scene.



Volume Partitioning – Portals

- Portals are simple and fast
- Low memory footprint
- Automatic generation is difficult
  - Generally need to be placed by hand
- Hard to find which node a point is in
  - Must constantly track movement of objects through portals
- Best at indoor scenes
  - Outside generates too many portals to be efficient

Volume Partitioning – BSP

- Binary space partition tree
- Tree of nodes
- Each node has plane that splits it in two
  - Two child nodes, one on each side of plane
- Some leaves marked as “solid”
- Others filled with renderable geometry

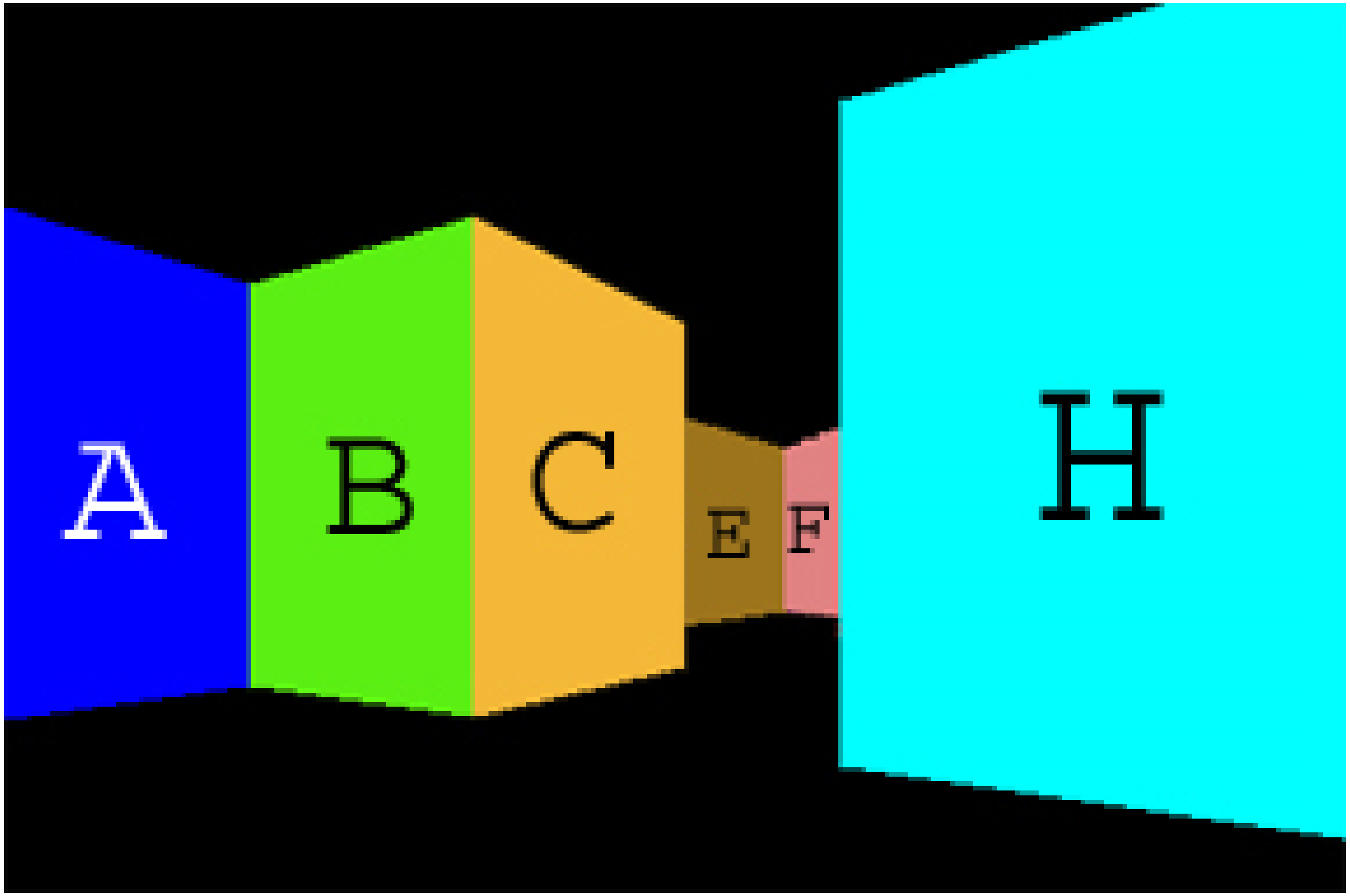
Volume Partitioning – BSP

- Finding which node a point is in is fast
  - Start at top node (current location)
  - Test which side of the plane the point is on
  - Move to that child node
  - Stop when leaf node hit (or all pixels full)
- Visibility determination is similar to portals
  - Portals implied from BSP planes
- Automated BSP generation is common
- Generates far more nodes than portals
  - Higher memory requirements

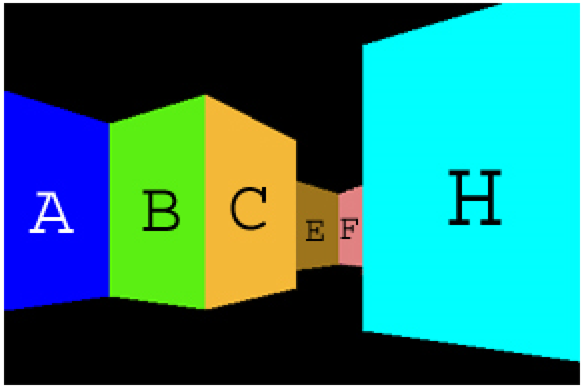
BSP in Doom

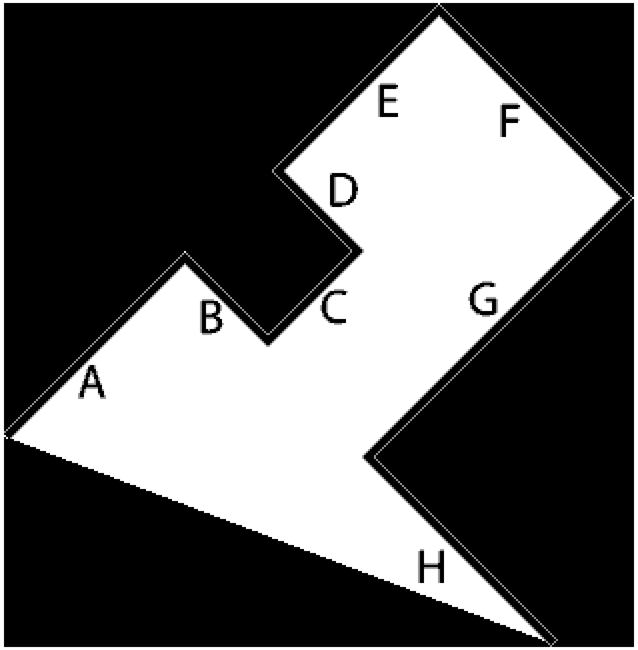
- BSP maps for each level generated ahead of time.
- Start at root node (represent entire level)
- Draw the leaf child nodes of this node recursively.
  - The child node closest to the camera is drawn first.
- When a subsector (leaf) is reached, draw it.
- The process is complete when the whole column of pixels is filled (i.e., there are no more gaps left).
- http://maven.smith.edu/~mcharley/bsp/

BSP (1)



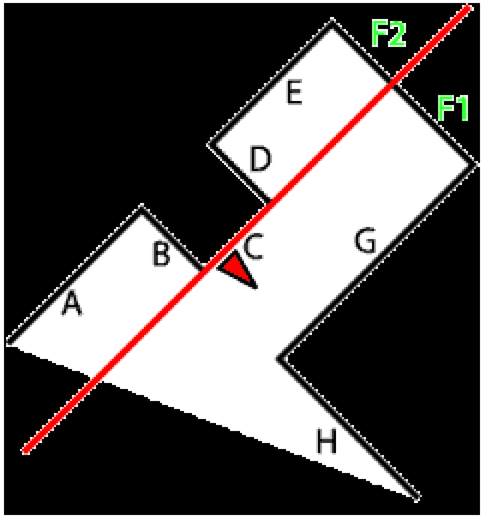
BSP (2)



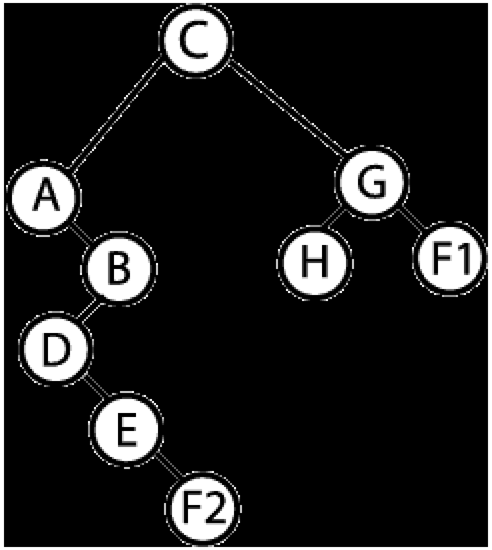


list of all lines: { A,B,C,D,E,F,G,H }.

BSP (3)



Lines that are in front of C: { G, H , F1}
Lines that are in back of C: { A, B, D, E, F2 }



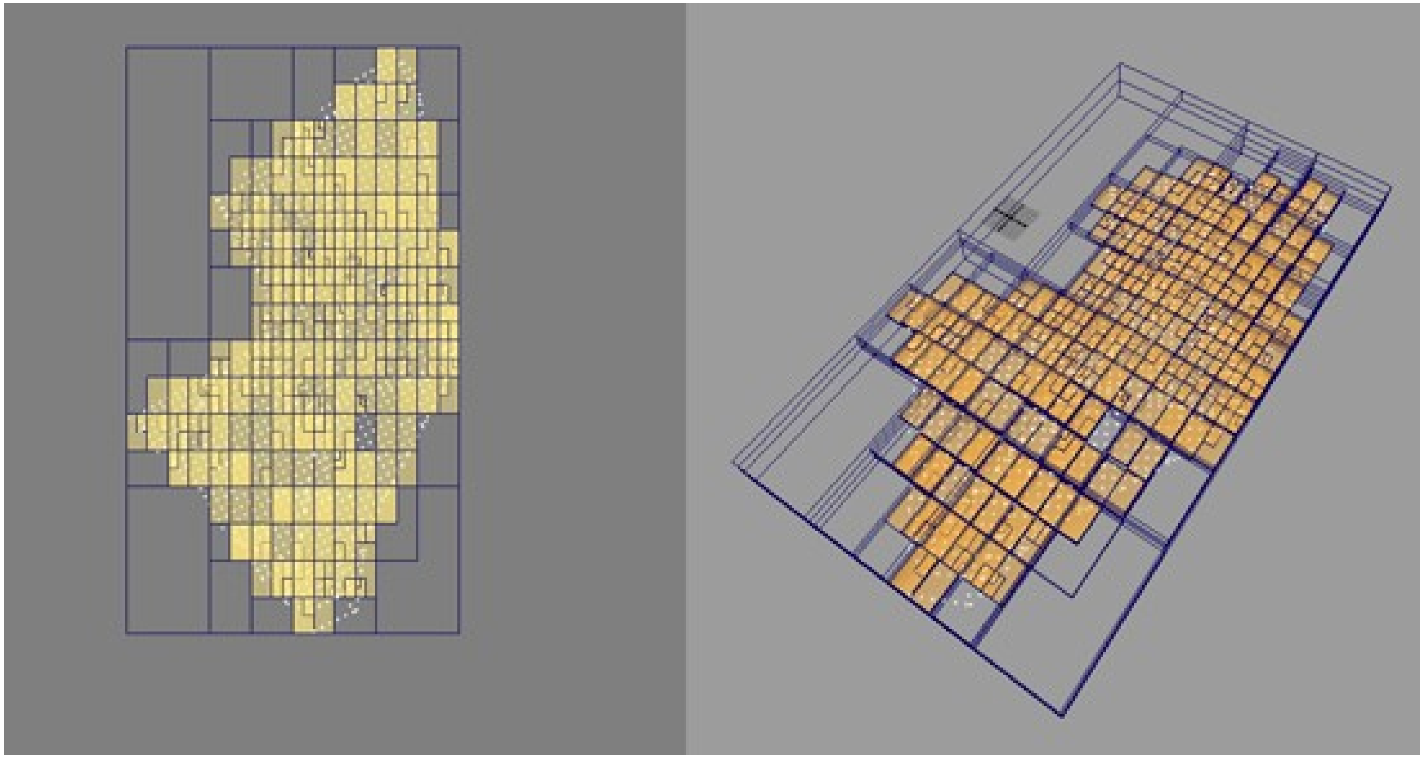
BSP(4)

- Check if camera is in front of or in back of the wall root node of bsp tree (C).
- Viewpoint is in front of wall C. Draw all walls behind C first.
- Rendering order will be F2,E,D,B,A,C,F1,G,H
- Note, drawing G and F1 is dumb!

BSP(5)

- Reverse order of portal (wall) drawing.
- Use Z buffer (or stencil buffer) to skip things that won't be seen.
- Once all pixels are accounted for (something is going to be drawn) stop.
  - Keep track with a utility buffer.
- Parition based on "world space" rather than "wall space" and entire leaf nodes can be eliminated because to get to them you have to pass through one or more "solid" nodes.

BSP



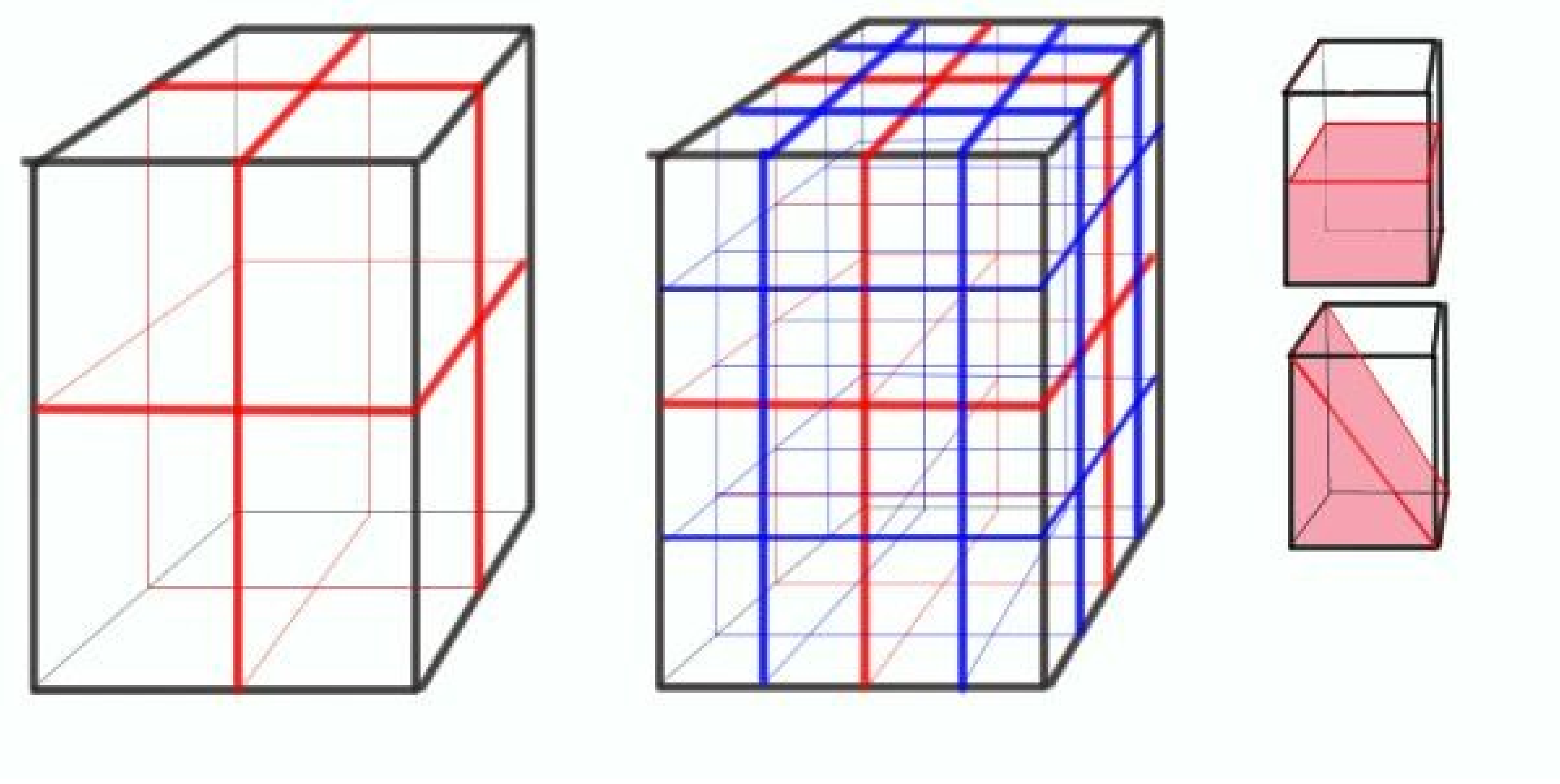
Volume Partitioning: Quadtree

- Quadtree (2D) and octree (3D)
  - Quadtrees described here
  - Extension to 3D octree is obvious
- Each node is square
  - Usually power-of-two in size
- Has four child nodes or leaves
- Each is a quarter of size of parent

Volume Partitioning: Quadtree

- Fast to find which node point is in
- Mostly used for simple frustum culling
- Not very good at indoor visibility
  - Quadtree edges usually not aligned with real geometry
- Very low memory requirements
- Good at dynamic moving objects
  - Insertion and removal is very fast

Quadtrees



Volume Partitioning - PVS

- Potentially visible set
- Based on any existing node system
- For each node, stores list of which nodes are potentially visible
- Use list for node that camera is currently in
  - Ignore any nodes not on that list – not visible
- Static lists
  - Precalculated at level authoring time
  - Ignores current frustum
  - Cannot deal with moving occluders

Volume Partitioning - PVS

- Very fast
  - No recursion, no calculations
- Still need frustum culling
- Difficult to calculate
  - Intersection of volumes and portals
  - Lots of tests – very slow
- Most useful when combined with other partitioning schemes

Volume Partitioning

- Different methods for different things
- Quadtree/octree for outdoor views
  - Does frustum culling well
  - Hard to cull much more for outdoor views
- Portals or BSP for indoor scenes
- BSP or quadtree for collision detection
  - Portals not suitable

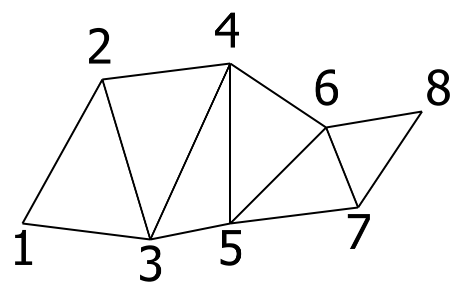
Stop Here ?

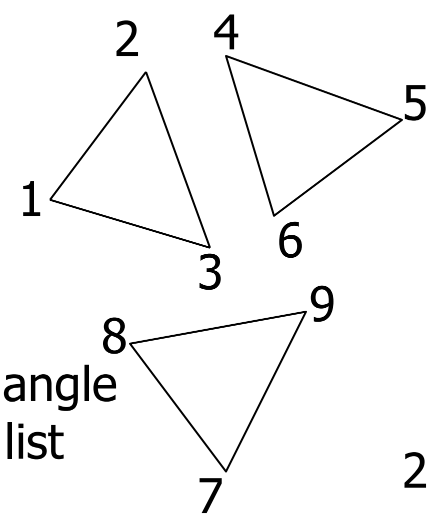
Rendering Primitives

- Strips, Lists, Fans
- Indexed Primitives
- The Vertex Cache
- Quads and Point Sprites

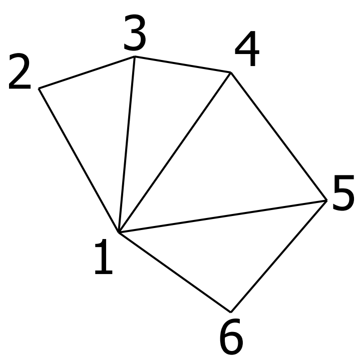
Strips, Lists, Fans

Triangle strip

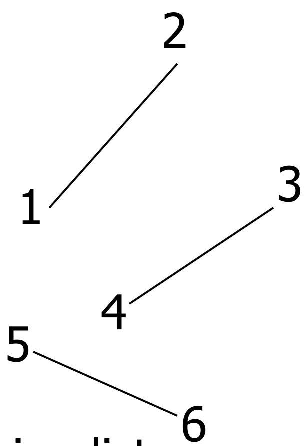




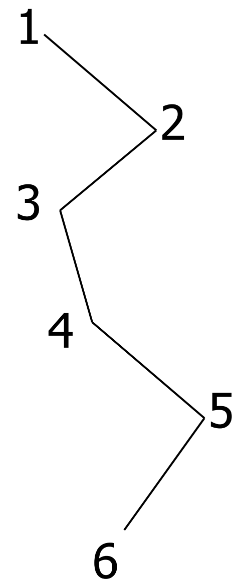
Triangle list



Triangle fan



Line list



Line strip

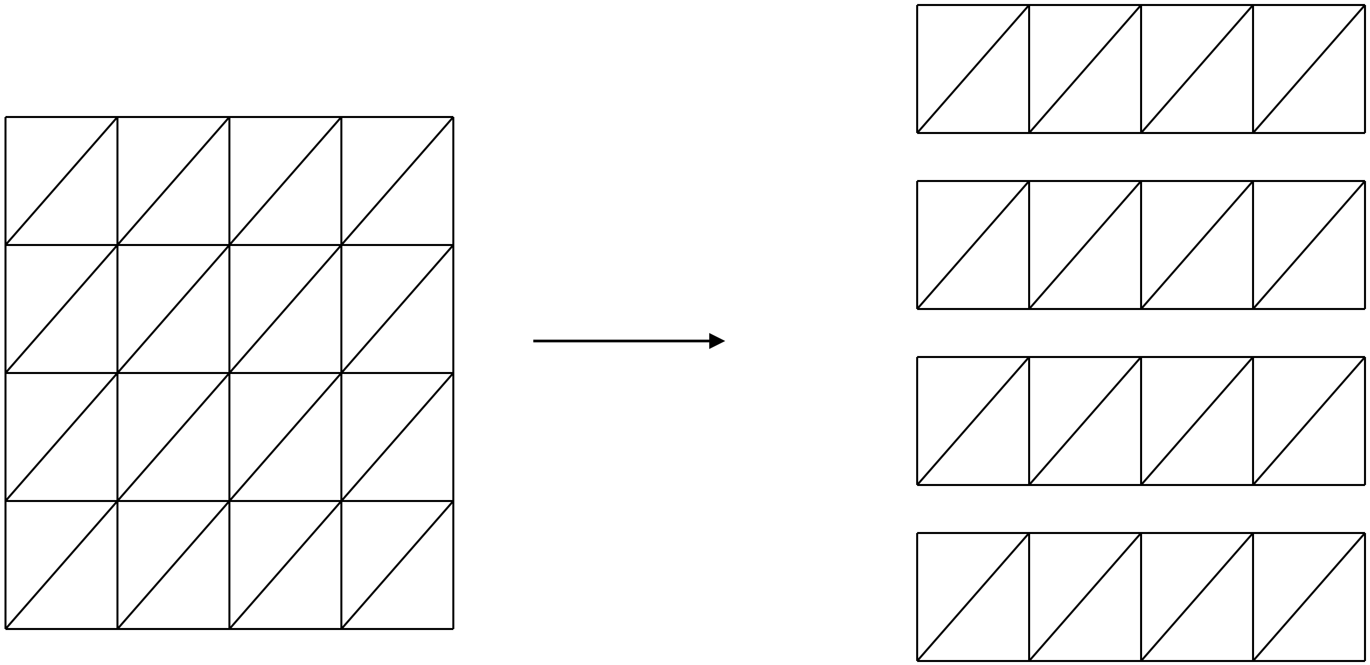
Strips, Lists, Fans (2)

- List has no sharing
  - Vertex count = triangle count * 3
- Strips and fans share adjacent vertices
  - Vertex count = triangle count + 2
  - Lower memory
  - Topology restrictions
  - Have to break into multiple rendering calls

Strips, Lists, Fans (3)

- Most meshes: tri count = 2x vert count
- Using lists duplicates vertices a lot!
  - Total of 6x number of rendering verts
- Strips or fans still duplicate vertices
  - Each strip/fan needs its own set of vertices
  - More than doubles vertex count
    - Typically 2.5x with good strips
  - Hard to find optimal strips and fans
  - Have to submit each as separate rendering call

Strips, Lists, Fans (4)



32 triangles, 25 vertices

4 strips, 40 vertices

25 to 40 vertices is 60% extra data!

Indexed Primitives

- Vertices stored in separate array
  - No duplication of vertices
  - Called a “vertex buffer” or “vertex array”
- Triangles hold indices, not vertices
- Index is just an integer
  - Typically 16 bits
  - Duplicating indices is cheap
  - Indexes into vertex array

The Vertex Cache

- Vertices processed by vertex shader
- Results used by multiple triangles
- Avoid re-running shader for each tri
- Storing results in video memory is slow
- So store results in small cache
  - Requires indexed primitives
- Cache typically 16-32 vertices in size
  - This gets around 95% efficiency

The Vertex Cache (2)

- Size and type of cache usually unknown
  - LRU or FIFO replacement policy
  - Also odd variants of FIFO policy
  - Variable cache size according to vertex type
- Reorder triangles to be cache-friendly
  - **Not** the same as finding optimal strips!
  - Render nearby triangles together
  - “Fairly good” is easy to achieve
  - Ideal ordering still a subject for research

Quads and Point Sprites

- Quads exist in some APIs
  - Rendered as two triangles
  - Think of them as a tiny triangle fan
  - Not significantly more efficient
- Point sprites are single vertex + a screen size
  - Screen-aligned square
  - Not just rendered as two triangles
  - Annoying hardware-specific restrictions
  - Rarely worth the effort

Textures

- Texture Formats
- Texture Mapping
- Texture Filtering
- Rendering to Textures

Texture Formats

- Textures made of texels
- Texels have R,G,B,A components
  - Often do mean red, green, blue colors
  - Really just a labelling convention
  - Shader decides what the numbers “mean”
- Not all formats have all components
- Different formats have different bit widths for components
  - Trade off storage space and speed for fidelity

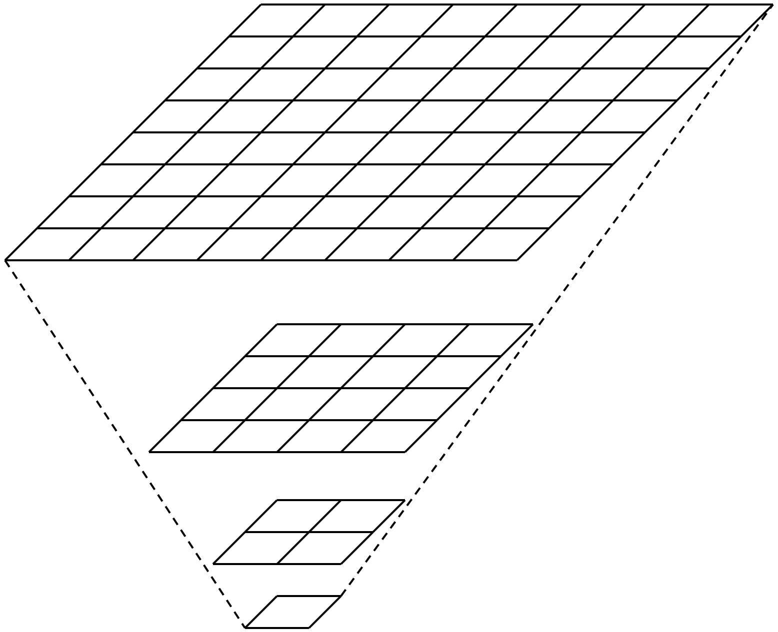
Texture Formats (2)

- Common formats:
  - A8R8G8B8: 8 bits per comp, 32 bits total
  - R5G6B5: 5 or 6 bits per comp, 16 bits total
  - A32f: single 32-bit floating-point comp
  - A16R16G16B16f: four 16-bit floats
  - DXT1: compressed 4x4 RGB block: 64 bits

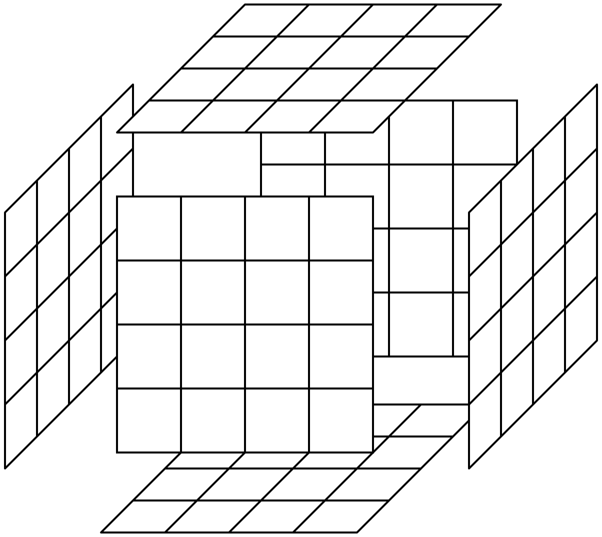
Texture Formats (3)

- Texels arranged in variety of ways
  - 1D linear array of texels
  - 2D rectangle/square of texels
  - 3D solid cube of texels
  - Six 2D squares of texels in hollow cube
- All the above can have mipmap chains
  - Mipmap is half the size in each dimension
  - Mipmap chain – all mipmaps to size 1

Texture Formats (4)



8x8 2D texture with mipmap chain



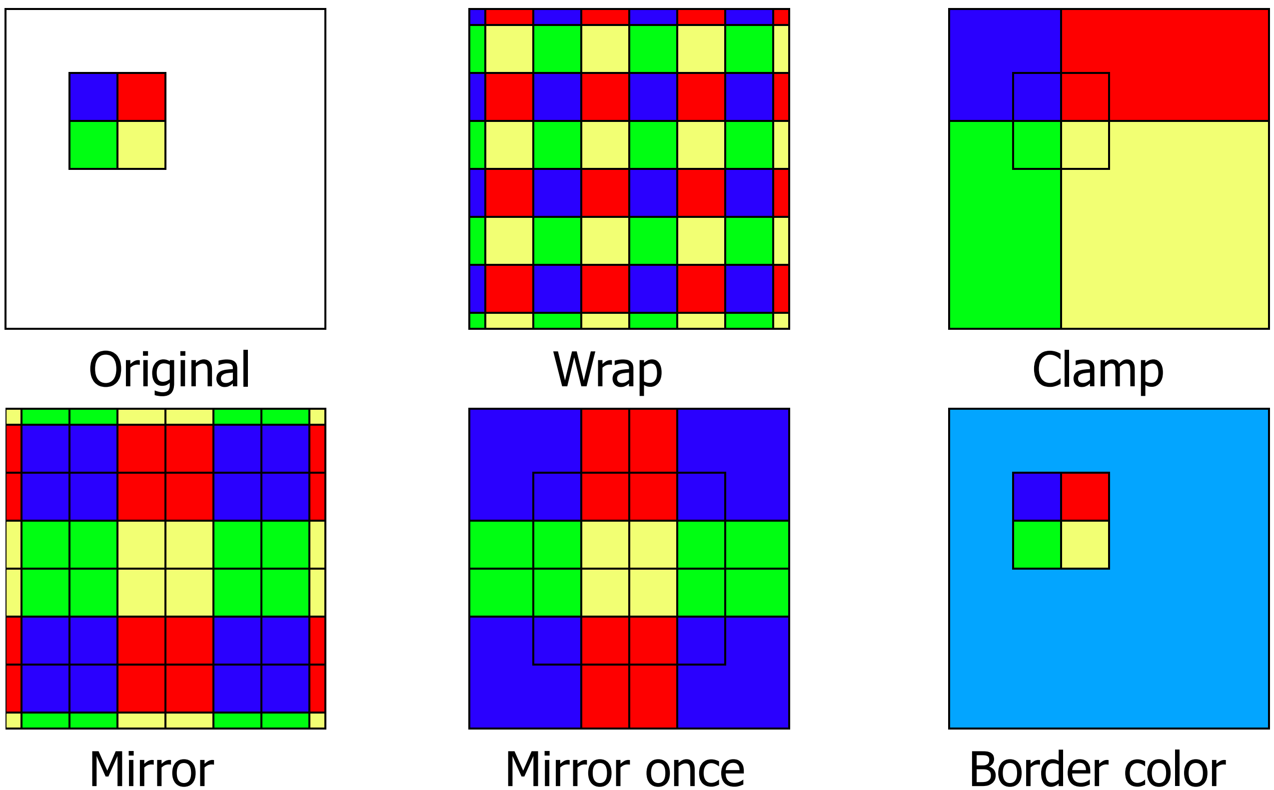
4x4 cube map (shown with sides expanded)

Texture Mapping

- Texture coordinates called U, V, W
- Only need U for 1D; U,V for 2D
- U,V,W typically stored in vertices
- Or can be computed by shaders
- Ranges from 0 to 1 across texture
  - However many texels texture contains
- Except for cube map – range is -1 to +1

Texture Mapping (2)

- Wrap mode controls values outside 0-1



Black edges shown for illustration only

Texture Filtering

- Point sampling enlarges without filtering
  - When magnified, texels very obvious
  - When minified, texture is “sparkly”
  - Useful for precise UI and font rendering
- Bilinear filtering blends edges of texels
  - Texel only specifies color at centre
  - Magnification looks better
  - Minification still sparkles a lot

Texture Filtering (2)

- Mipmap chains help minification
  - Pre-filters a texture to half-size
  - Multiple mipmaps, each smaller than last
  - Rendering selects appropriate level to use
- Transitions between levels are obvious
  - Change is visible as a moving line
- Use trilinear filtering
  - Blends between mipmaps smoothly

Texture Filtering (3)

- Trilinear can over-blur textures
  - When triangles are edge-on to camera
  - Especially roads and walls
- Anisotropic filtering solves this
  - Takes multiple samples in one direction
  - Averages them together
  - Quite expensive in current hardware

Rendering to Textures

- Textures usually made in art package
  - Loaded from disk
- But any 2D image can be a texture
  - Can set texture as the target for rendering
  - Render scene 1 to texture
  - Then set backbuffer as target again
  - Render scene 2 using texture
- Cube map needs six renders, one per face

Lighting

- Components
- Lighting Environment
- Multiple Lights
- Diffuse Material Lighting
- Normal Maps
- Pre-computed Radiance Transfer
- Specular Material Lighting
- Environment Maps

Components

- Lighting is in three stages:
  - What light shines on the surface?
  - How does the material interact with light?
  - What part of the result is visible to eye?
  - Real-time rendering merges last two
- Occurs in vertex and/or pixel shader
  - Many algorithms can be in either

Lighting Environment

- Answers first question:
  - What light shines on the surface?
- Standard model is infinitely small lights
  - Position
  - Intensity
  - Color
- Physical model uses inverse square rule
  - brightness = light brightness / distance^2

Lighting Environment (2)

- But this gives huge range of brightnesses
- Monitors have limited range
- In practice it looks terrible
- Most people use inverse distance
  - brightness = light brightness / distance
- Add min distance to stop over-brightening
  - Except where you *want* over-brightening!
- Add max distance to cull lights
  - Reject very dim lights for performance

Multiple Lights

- Environments have tens or hundreds
  - Too slow to consider every one every pixel
  - Approximate less significant ones
- Ambient light
  - Single color added to all lighting
  - Washes contrasts out of scene
  - Acceptable for overcast daylight scenes

Multiple Lights (2)

- Hemisphere lighting
  - Sky is light blue
  - Ground is dark green or brown
  - Dot-product normal with “up vector”
  - Blend between the two colors
  - Good for brighter outdoor daylight scenes

Multiple Lights (3)

- Cube map of irradiance
  - Stores incoming light from each direction
  - Look up value that normal points at
  - Can represent any lighting environment
- Spherical harmonic irradiance
  - Store irradiance cube map in frequency space
  - 10 color values gives at most 6% error
  - Calculation instead of cube-map lookup
  - Mainly for diffuse lighting

Diffuse Material Lighting

- Light is absorbed and re-emitted
- Re-emitted in all directions equally
- So it does not matter where the eye is
  - Same amount of light hits the pupil
- "Lambert" diffuse model is common
- Brightness is dot-product between surface normal and incident light vector

Normal Maps

- Surface normal vector stored in vertices
- Changes slowly
  - Surfaces look smooth
- Real surfaces are rough
  - Lots of variation in surface normal
  - Would require lots more vertices
- Normal maps store normal in a texture
- Look up normal at each pixel
- Perform lighting calculation in pixel shader

Pre-computed Radiance Transfer

- Surface usually represented by:
  - Normal
  - Color
  - Roughness
- But all we need is how it responds to light from a certain direction
- Above data is just an approximation
- Why not store response data directly?

Pre-computed Radiance Transfer

- Can include effects of:
  - Local self-shadowing
  - Local scattering of light
  - Internal structure (e.g. skin layers)
- But data size is huge
  - Color response for every direction
  - Different for each part of surface
  - Cube-map per texel would be crazy!

Pre-computed Radiance Transfer

- Store cube-maps as spherical harmonics
  - One SH per texel
  - Further compression by other methods
- But:
  - Difficult to do animated meshes
  - Still lots of memory
  - Lots of computation
  - Poor at specular materials

Specular Material Lighting

- Light bounces off surface
- How much light bounced into the eye?
  - Other light did not hit eye – so not visible!
- Common model is “Blinn” lighting
- Surface made of “microfacets”
- They have random orientation
  - With some type of distribution

Specular Material Lighting (2)

- Light comes from incident light vector
  - …reflects off microfacet
  - …into eye
- Eye and light vectors fixed for scene
- So we know microfacet normal required
- Called “half vector”
  - half vector = (incident + eye)/2
- How many have that normal?

Specular Material Lighting (3)

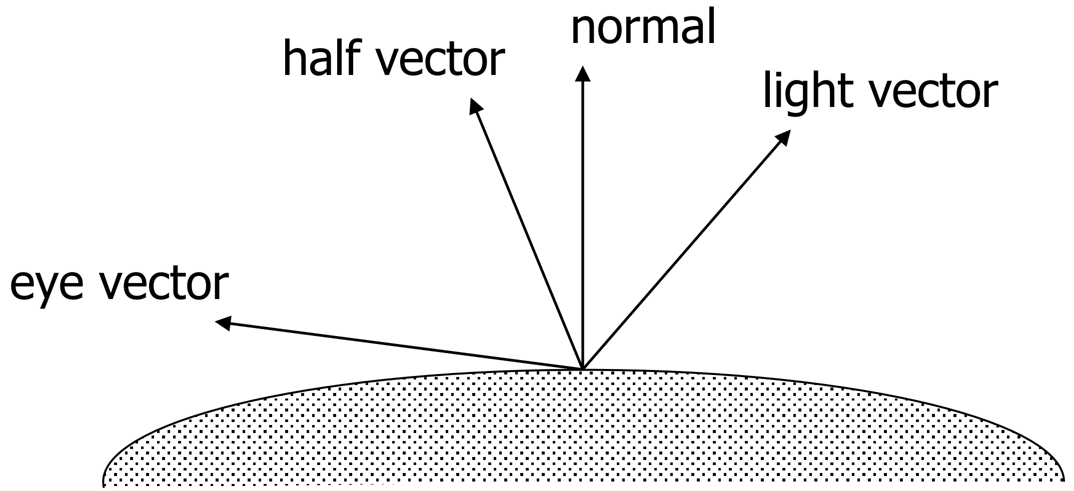
- Microfacets distributed around surface normal
  - According to “smoothness” value
- Dot-product of half-vector and normal
  - Then raise to power of “smoothness”
- Gives bright spot
  - Where normal=half vector
  - Tails off quicker when material is smoother

Specular Material Lighting (4)

half=(light+eye)/2

alignment=max (0, dot (half,normal))

brightness=alignment^{smoothness}



Environment Maps

- Blinn used for slightly rough materials
- Only models bright lights
  - Light from normal objects is ignored
- Smooth surfaces can reflect everything
  - No microfacets for smooth surfaces
  - Only care about one source of light
  - The one that reflects to hit the eye

Environment Maps - 2

- Put environment picture in cube map
- Reflect eye vector in surface normal
- Look up result in cube map
- Can take normal from normal map
  - Bumpy chrome

Environment Maps - 3

- Environment map can be static
  - Generic sky + hills + ground
  - Often hard to notice that it's not correct
  - Very cheap, very effective
- Or render every frame with real scene
  - Render to cube map sides
  - Selection of scene centre can be tricky
  - Expensive to render scene six times

Hardware Rendering Pipe

- Input Assembly
- Vertex Shading
- Primitive Assembly, Cull, Clip
- Project, Rasterize
- Pixel Shading
- Z, Stencil, Framebuffer Blend
- Shader Characteristics
- Shader Languages

Hardware Rendering Pipe

- Current outline of rendering pipeline
- Can only be very general
- Hardware moves at rapid pace
- Hardware varies significantly in details
- Functional view only
  - Not representative of performance
  - Many stages move in actual hardware

Input Assembly

- State changes handled
  - Textures, shaders, blend modes
- Streams of input data read
  - Vertex buffers
  - Index buffers
  - Constant data
- Combined into primitives

Vertex Shading

- Vertex data fed to vertex shader
  - Also misc. states and constant data
- Program run until completion
- One vertex in, one vertex out
  - Shader cannot see multiple vertices
  - Shader cannot see triangle structure
- Output stored in vertex cache
- Output position must be in clip space

Primitive Assembly, Cull, Clip

- Vertices read from cache
- Combined to form triangles
- Cull triangles
  - Frustum cull
  - Back face (clockwise ordering of vertices)
- Clipping performed on non-culled tris
- Produces tris that do not go off-screen

Project, Rasterize

- Vertices projected to screen space
  - Actual pixel coordinates
- Triangle is rasterized
  - Finds the pixels it actually affects
  - Finds the depth values for those pixels
- Finds the interpolated attribute data
  - Texture coordinates
  - Anything else held in vertices
- Feeds results to pixel shader

Pixel Shading

- Program run once for each pixel
- Given interpolated vertex data
- Can read textures
- Outputs resulting pixel color
- May optionally output new depth value
- May kill pixel
  - Prevents it being rendered

Z, Stencil, Framebuffer Blend

- Z and stencil tests performed
- Pixel may be killed by tests
- If not, new Z and stencil values written
- If no framebuffer blend
  - Write new pixel color to backbuffer
  - Otherwise, blend existing value with new

Shader Characteristics

- Shaders rely on massive parallelism
- Breaking parallelism breaks speed
  - Can be thousands of times slower
- Shaders may be executed in any order
- So restrictions placed on what shader can do
  - Write to exactly one place
  - No persistent data
  - No communication with other shaders

Shader Languages

- Many different shader capabilities
- Early languages looked like assembly
  - Different assembly for each shader version
- Now have C-like compilers
  - Hides a lot of implementation details
  - Works with multiple versions of hardware
- Still same fundamental restrictions
  - Don't break parallelism!
- Expected to keep evolving rapidly

Conclusions

- Traverse scene nodes
  - Reject or ignore invisible nodes
  - Draw objects in visible nodes
- Vertices transformed to screen space
  - Using vertex shader programs
  - Deform mesh according to animation
- Make triangles from them
- Rasterize into pixels

Conclusions (2)

- Lighting done by combination
  - Some part vertex shader
  - Some part pixel shader
  - Results in new color for each pixel
- Reject pixels that are invisible
- Write or blend to backbuffer