Real-time 3D Rendering Primer

Wessam Bahnassi – EA Montreal

Acknowledgements & Copyrights

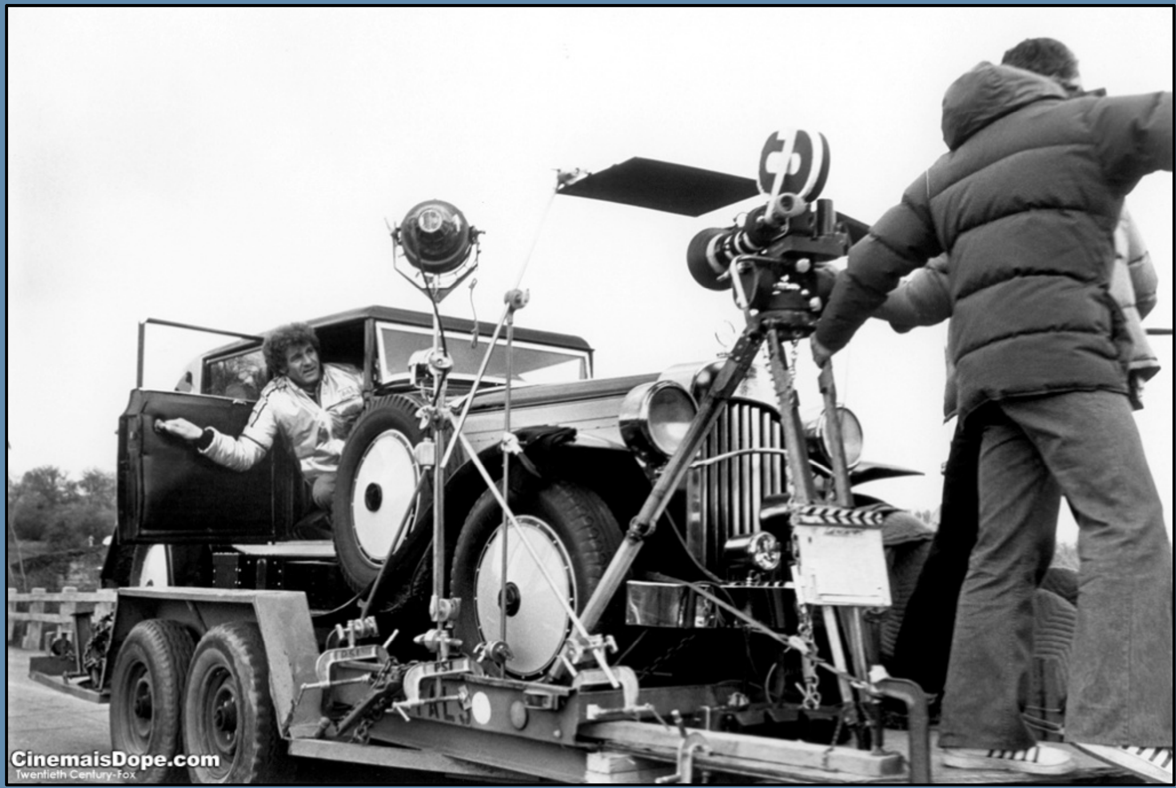
Table of Contents

1. Short Computer Graphics Intro

- Scene Components
- Real-time Rendering Pipeline
- Positioning in 3D
- Transformations and Spaces

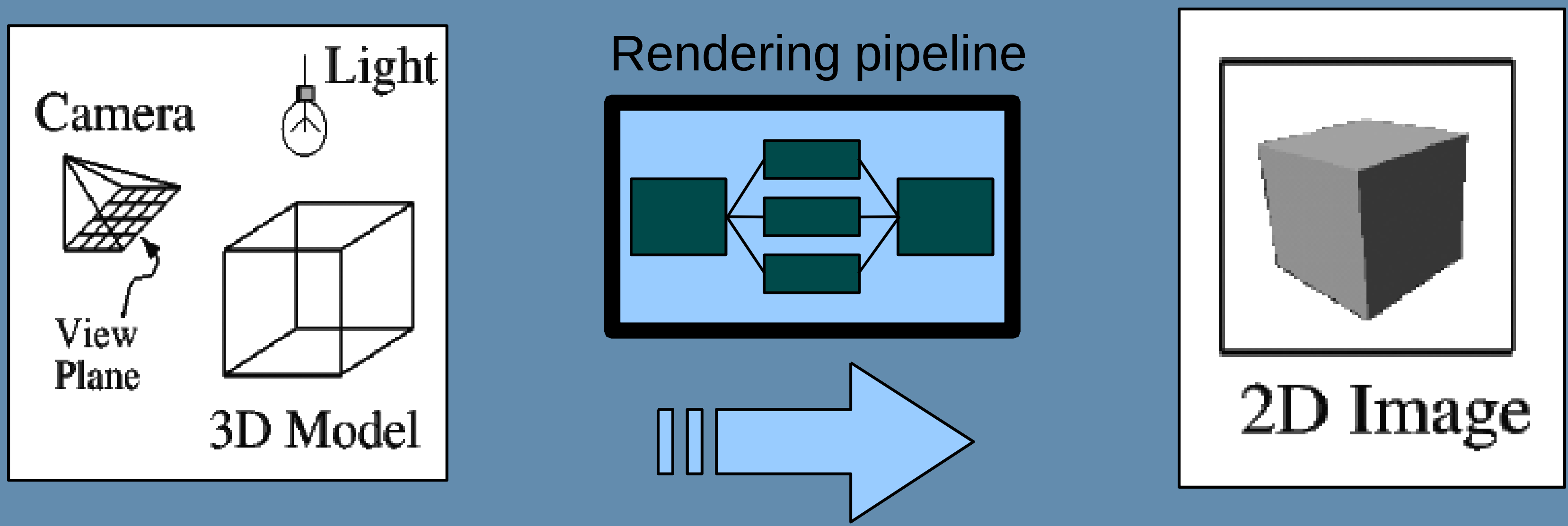
Scene Components

- Sharing a lot with cinematography.
- A scene is made of various components:
  - Things to film (the setting)
  - Camera
  - Lights



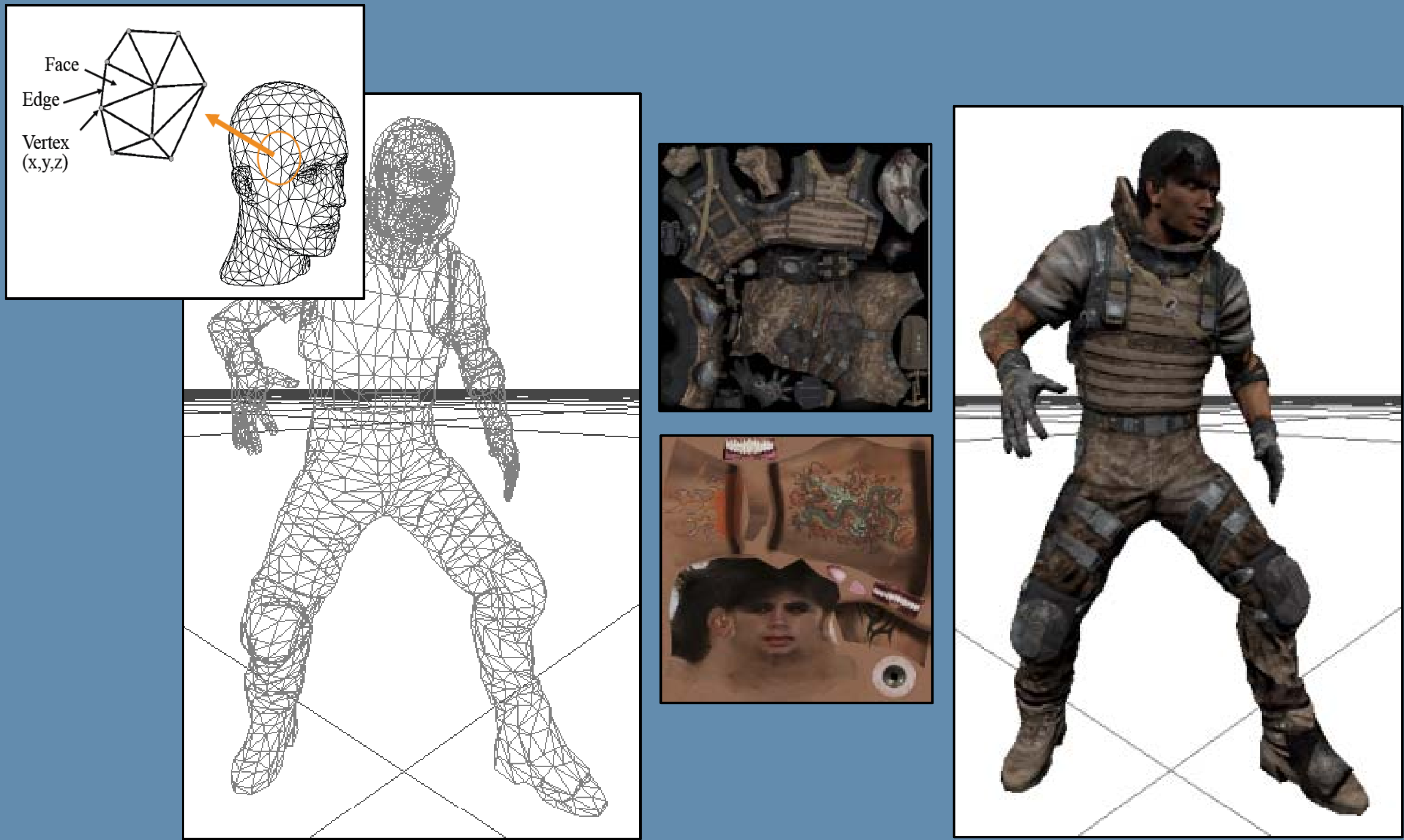
Scene Components

- Similar scene components
- Scene information goes through a “pipeline” that transforms it to a 2D image displayed on the screen



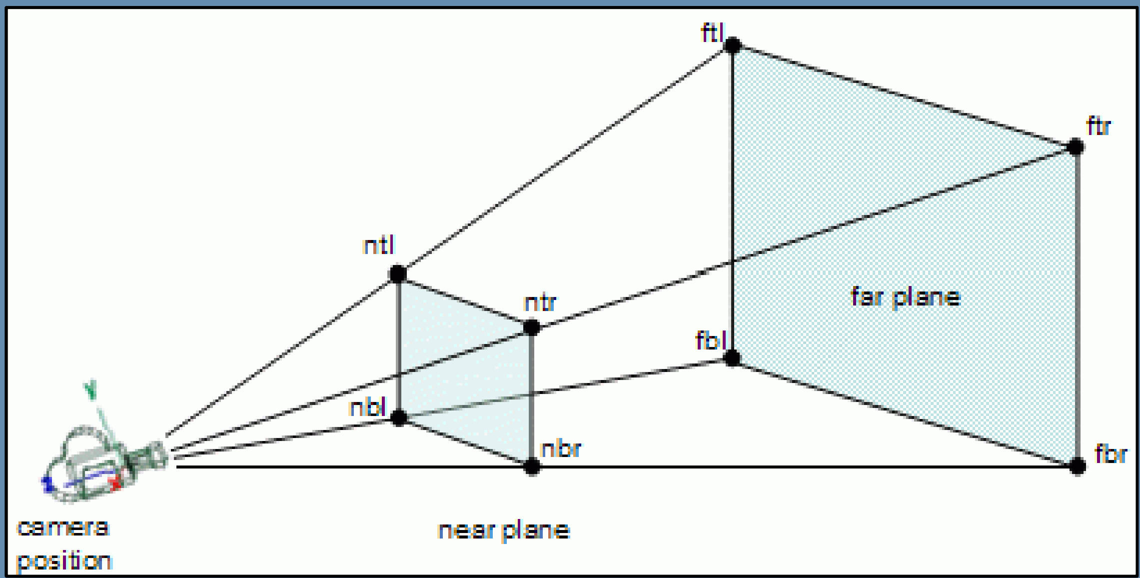
Scene Components: 3D Models

- Geometry ultimately drawn as triangles, accompanied with additional data to increase detail.



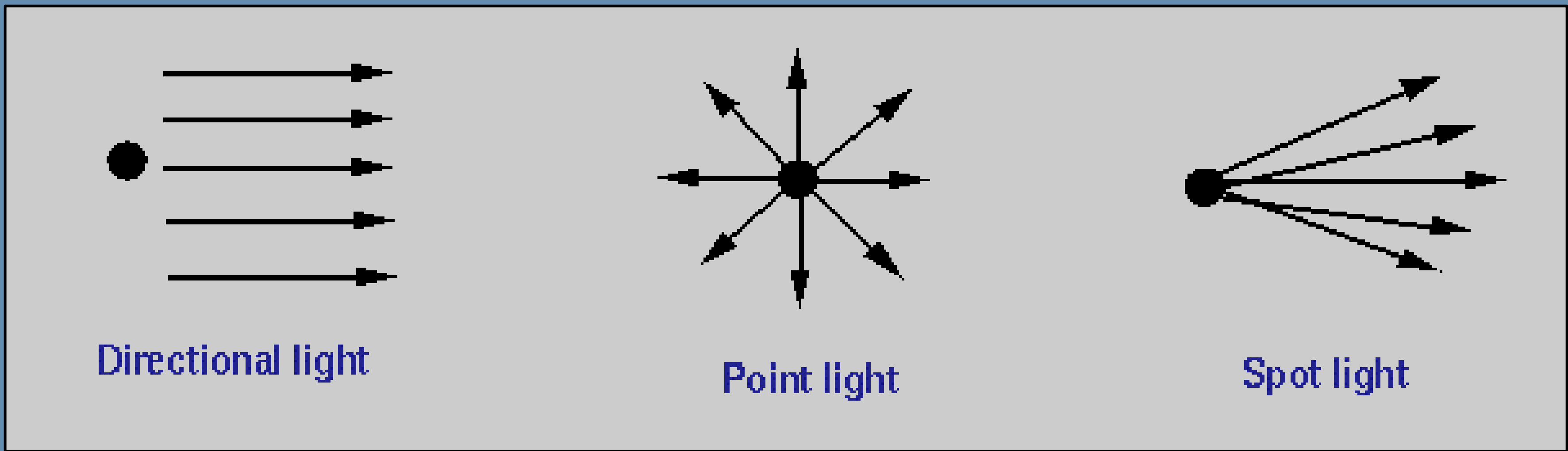
Scene Components: Camera

- An imaginary entity that carries view properties and 2D projection parameters, including:
  - Position
  - View direction
  - Lens properties
  - Projection type
  - Clip planes

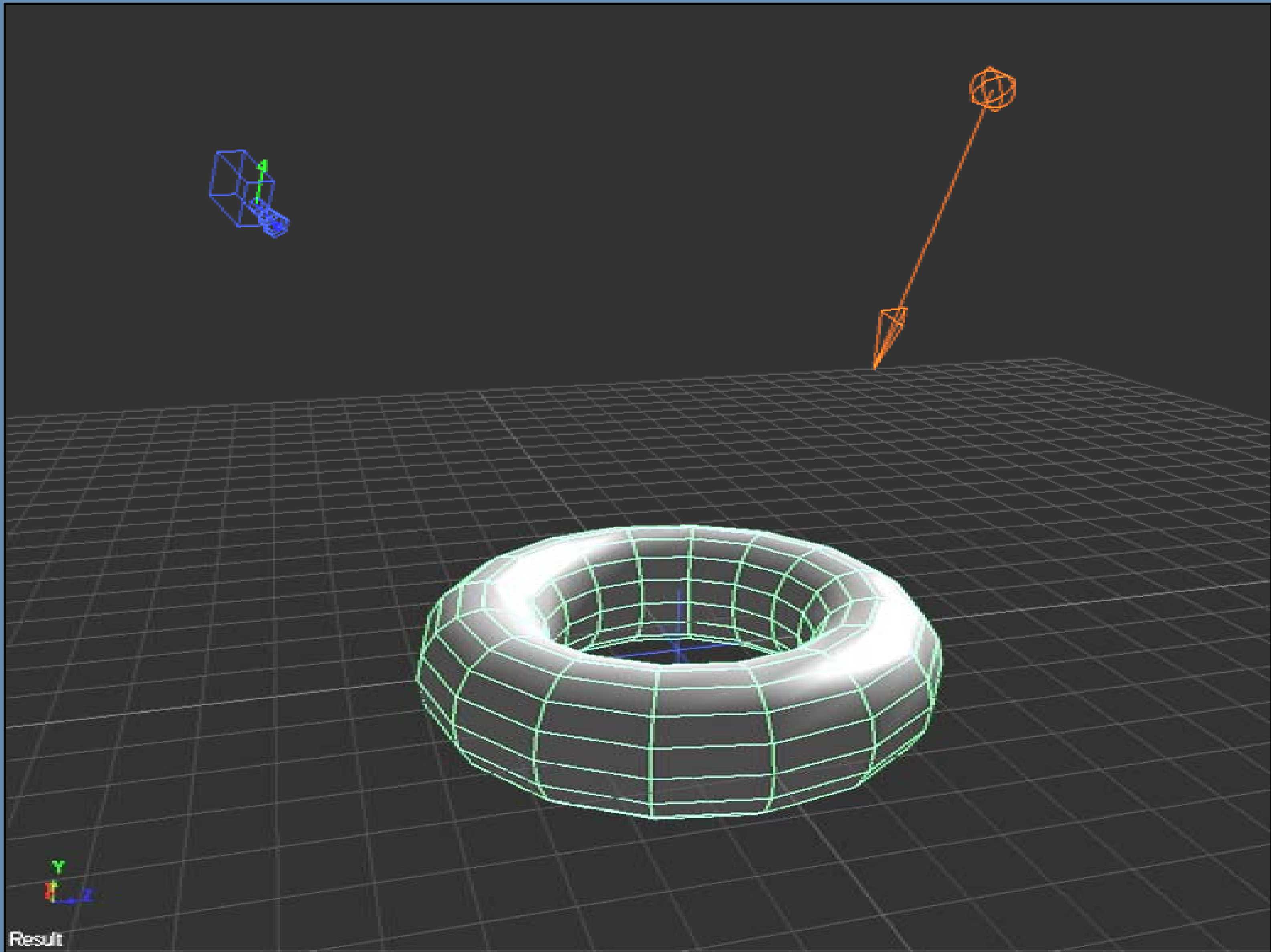


Scene Components: Lights

- Another imaginary entity that carries lighting method and properties.
- More detail to come in the rendering algorithms section.

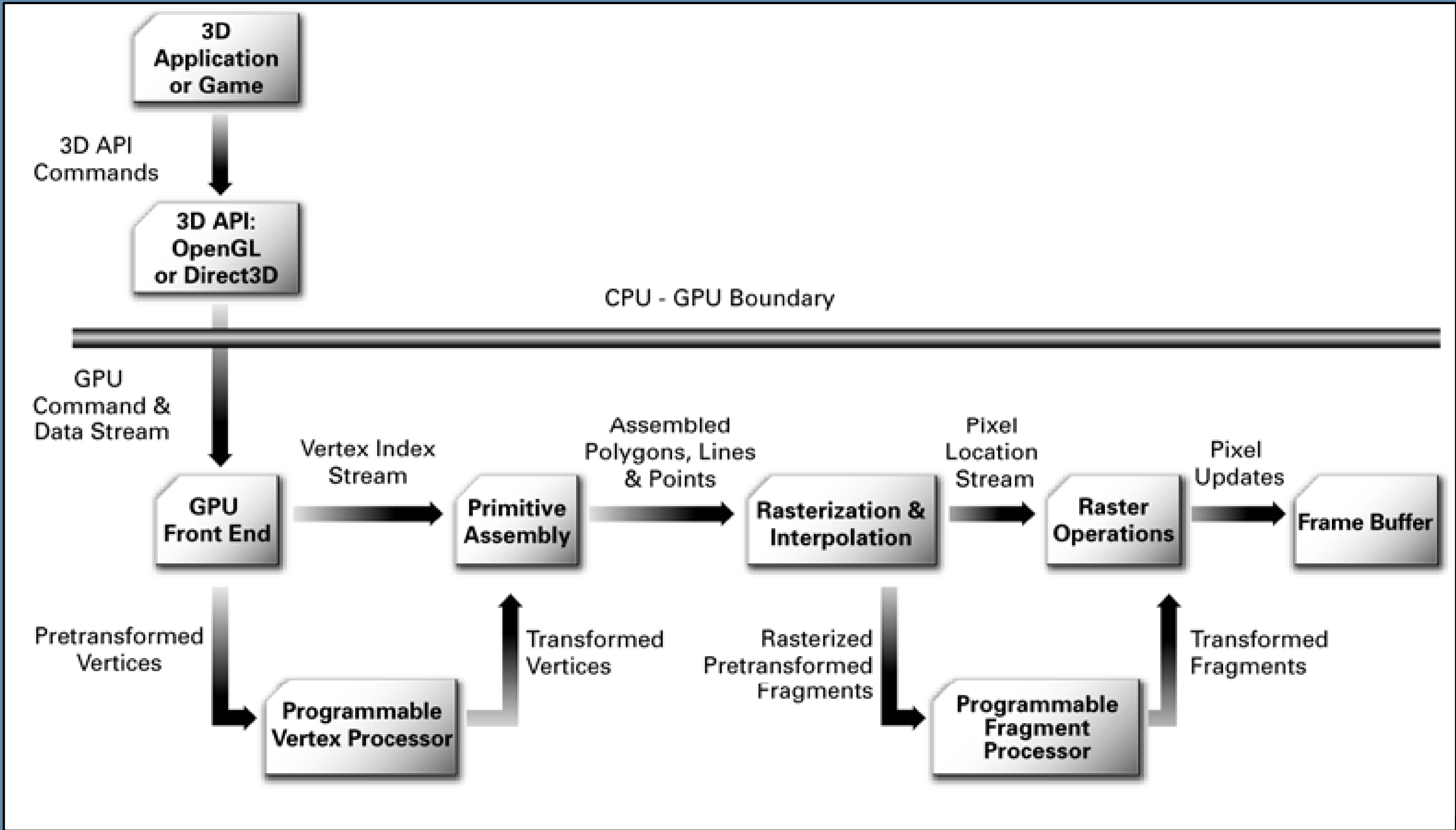


Scene Components



Real-time Rendering Pipeline

- Takes information through a series of steps to generate the final image



Real-time Rendering Pipeline

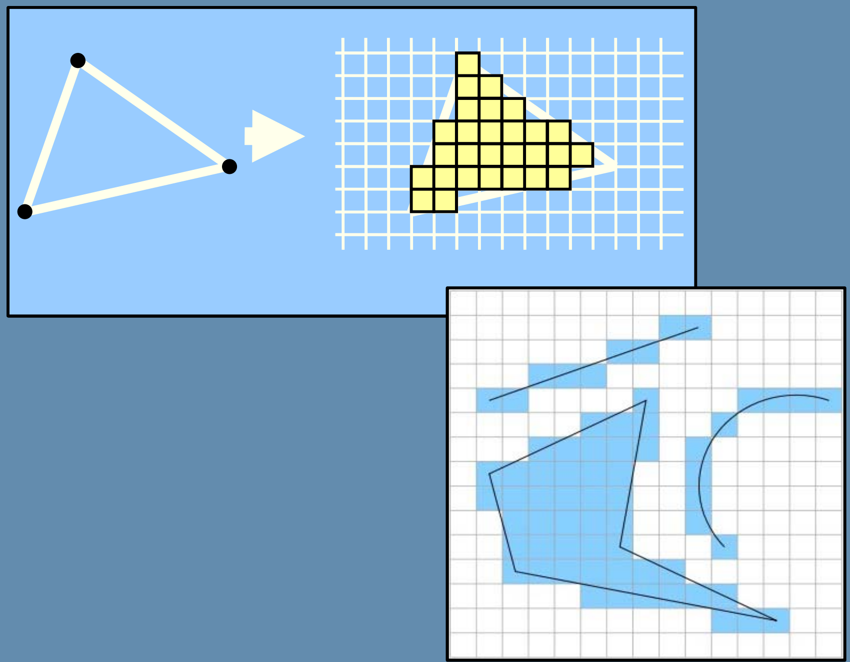
- GPU is a big state machine.
- Set drawing states, then issue draw commands.
- Sample rendering code:

```
D3DMATERIAL RedMat;
RedMat.Diffuse = RGBA(1,0,0,1);
RedMat.Specular = RGBA(1,1,1,1);
RedMat.SpecularPower = 24.0;

D3DDevice->SetMaterial(&RedMat);
D3DDevice->SetTexture(ClothTexture);

D3DDevice->DrawPrimitive(… Object1 Draw Info …);
D3DDevice->DrawPrimitive(… Object2 Draw Info …);

D3DDevice->SetTexture(ConcreteTexture);

D3DDevice->DrawPrimitive(… Object3 Draw Info …);
```

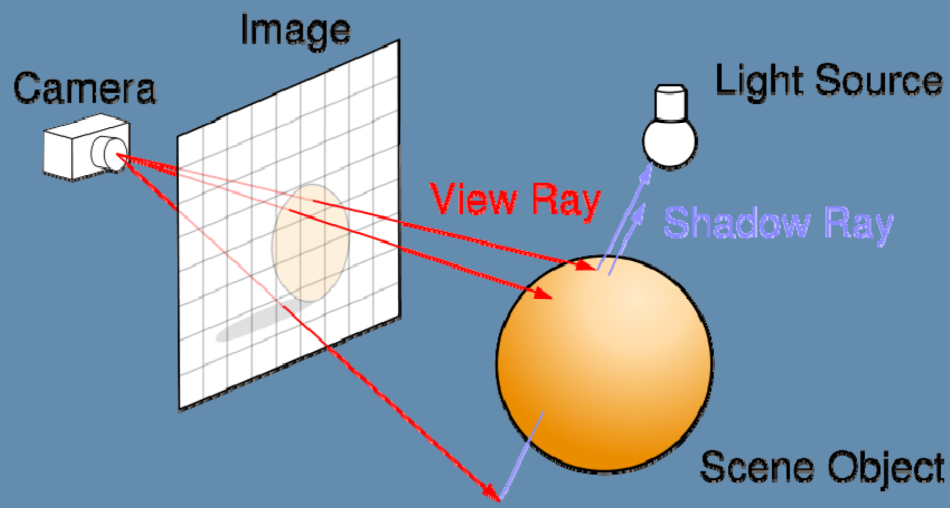
Real-time Rendering Pipeline: Polygon Presentation

- Polygons need to be presented on a screen with a finite number of pixels:
  - Rasterization
  - Ray tracing (rarely used in real-time applications and games)

Rasterization

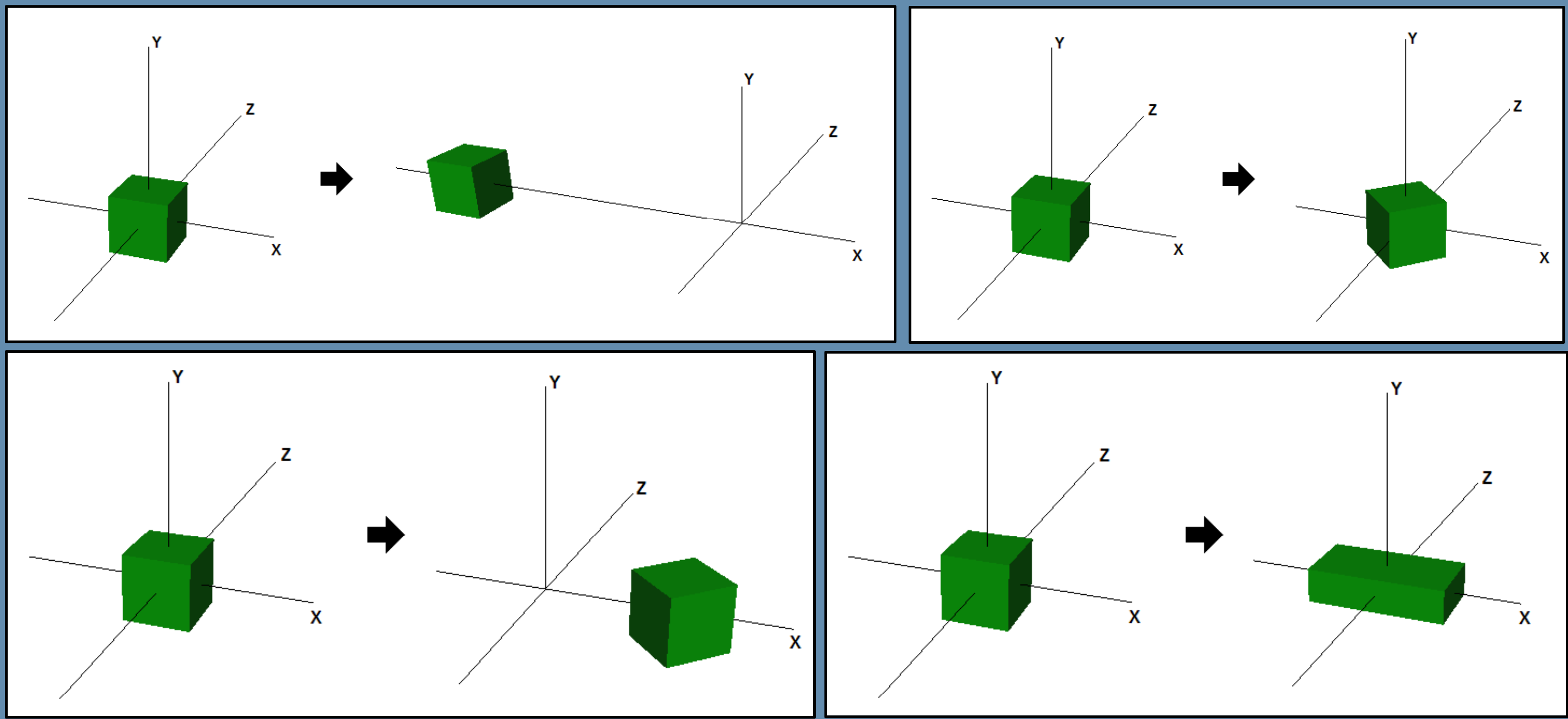


Ray tracing



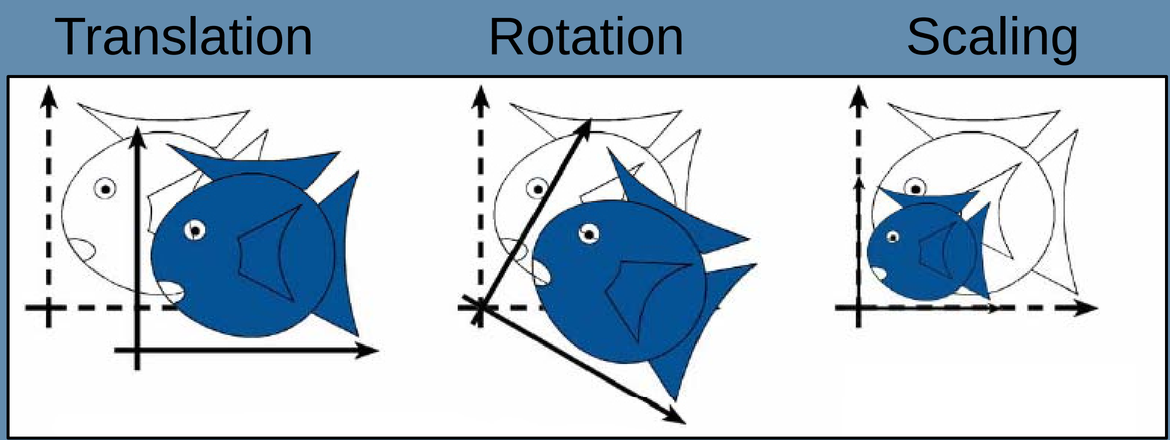
Positioning in 3D

- We need to position and orientate models in 3D space while preserving their structure.
- We can control structure change via scaling, or more advanced calculations (will be discussed further in the algorithms section).
- Need a framework to represent such transformations.



Transformations and Spaces (1)

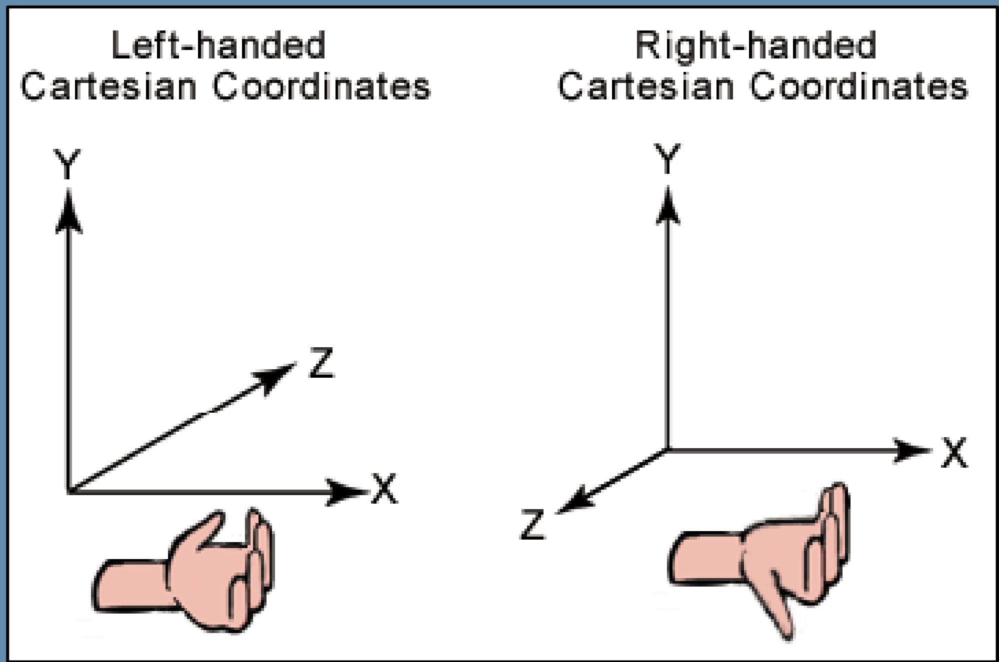
- 4x4 transformation matrices offer a very useful framework to move and orientate 3D models in the world.
- Affine transforms include translation (move), rotation, and scaling.
- They can be expressed in row-major (D3DX) order or column-major (OpenGL).



$$\begin{bmatrix} 1 & 0 & 0 & tX \\ 0 & 1 & 0 & tY \\ 0 & 0 & 1 & tZ \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} sX & 0 & 0 & 0 \\ 0 & sY & 0 & 0 \\ 0 & 0 & sZ & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transformations and Spaces (2)

- 3D scenes must establish a global coordinate system convention:
  - Usually either left-handed or right-handed
- 3D models, lights, and cameras all rely on this system to locate themselves in the world, thus it is called: *world coordinates*.

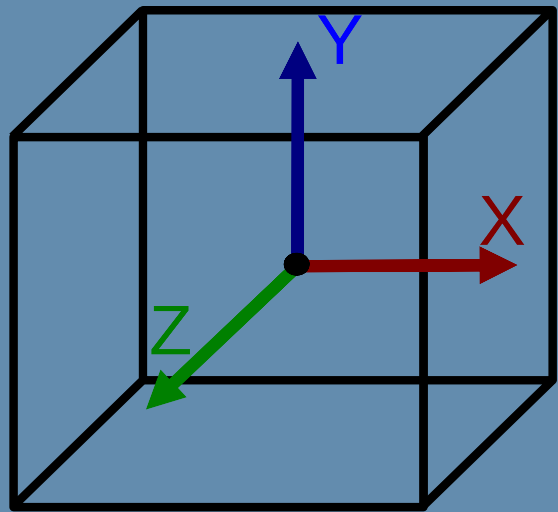


Transformations and Spaces (3)

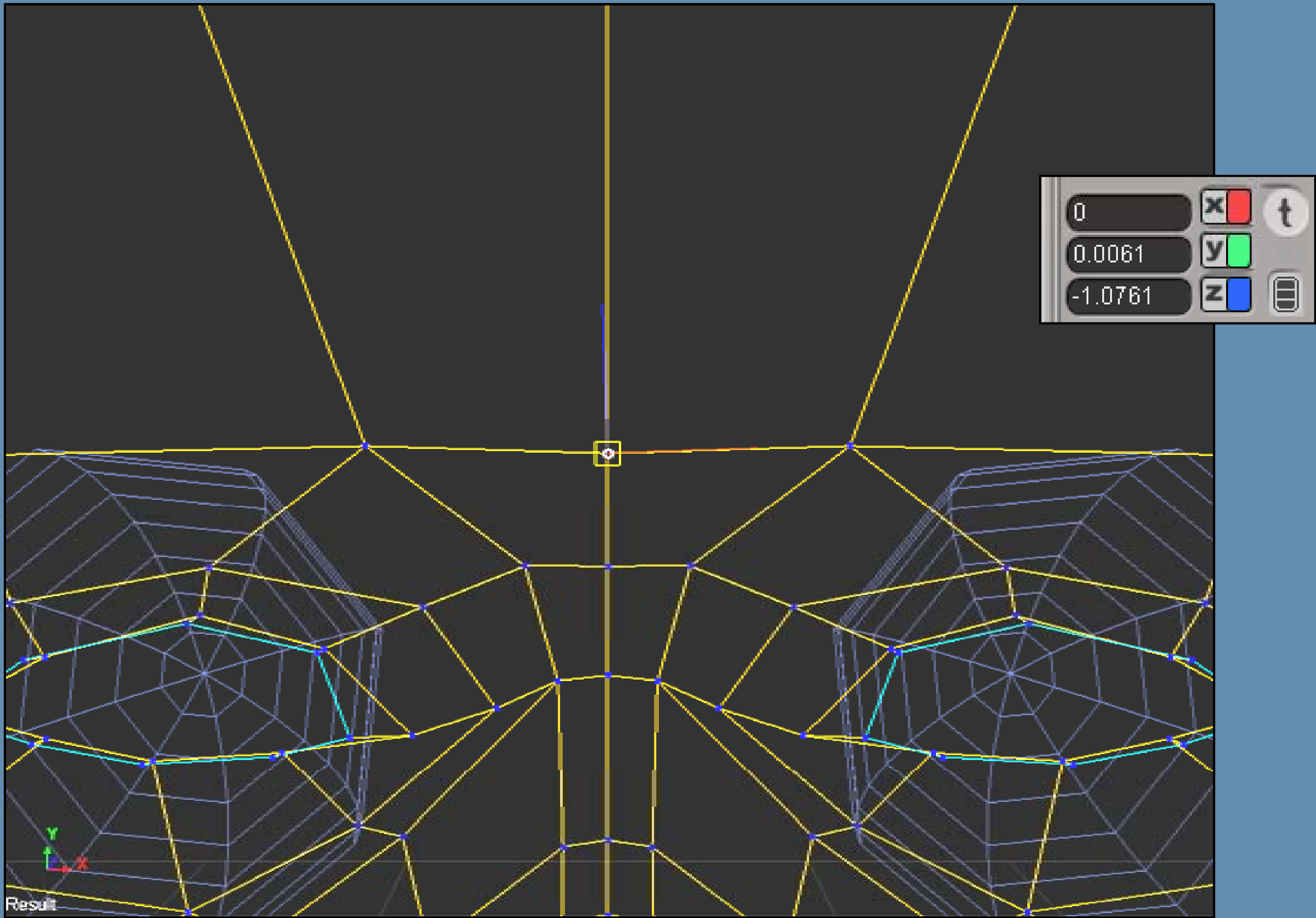
- A matrix can be seen as representing a transform between coordinate systems (spaces).
- Common space transformations are:
  - Local-To-World
  - World-To-Camera
  - Camera-To-Clip (projection)
- Combine transforms via matrix multiplication (order-dependent!):
  - Local-To-World * World-To-Camera = Local-To-Camera

Transformations and Spaces: Local Space

- 3D models are defined in local space.
- Vertex positions are relative to an imaginary *pivot*.
- Usually the center of the object.

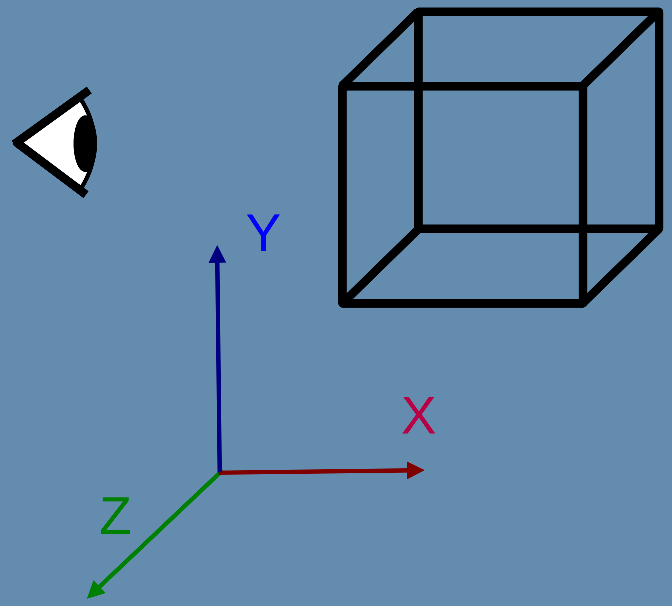


Transformations and Spaces: Local Space

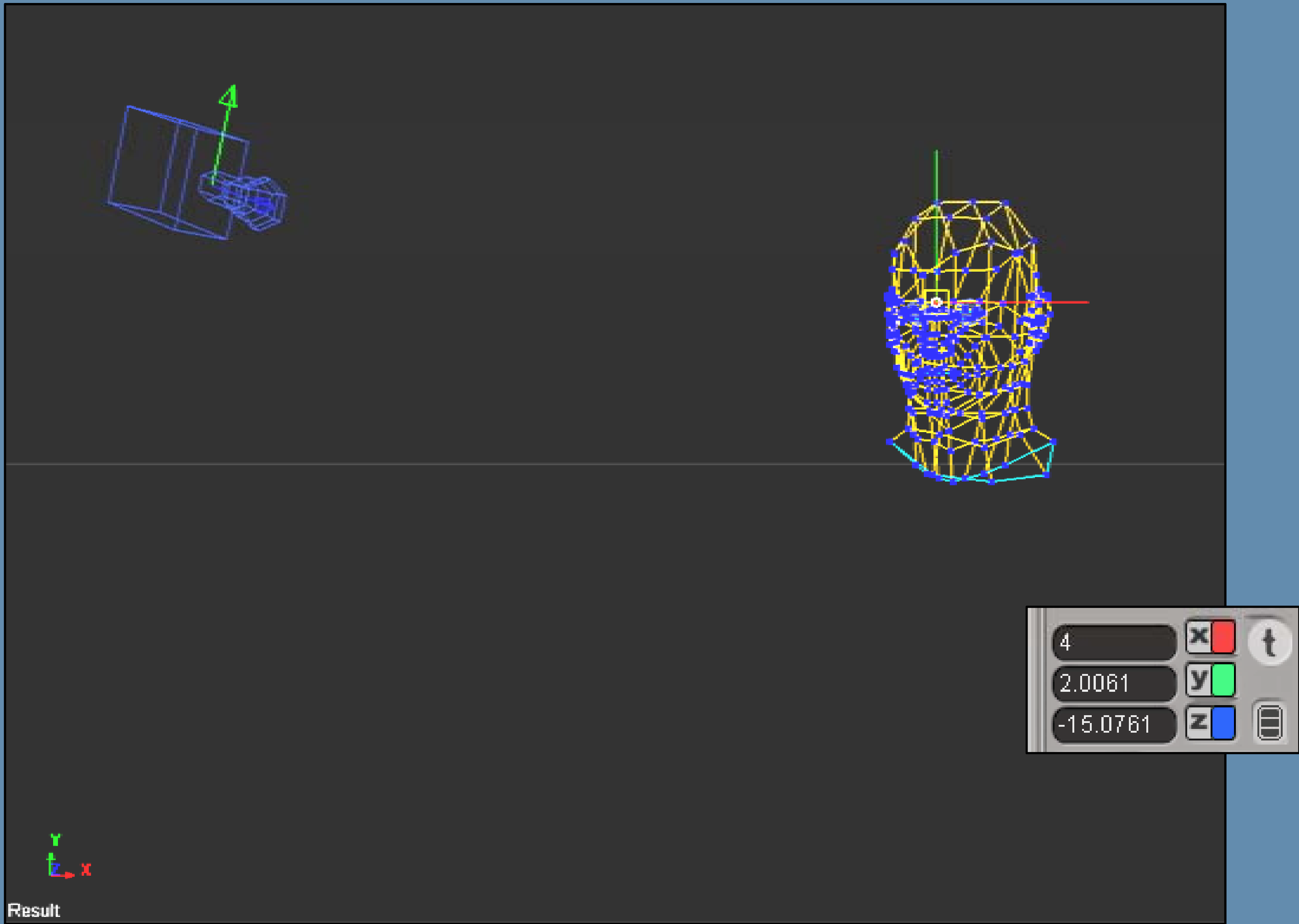


Transformations and Spaces: World Space

- World space is the reference for all other objects.
- All object positions/orientations are in world coordinates (including cameras and lights).

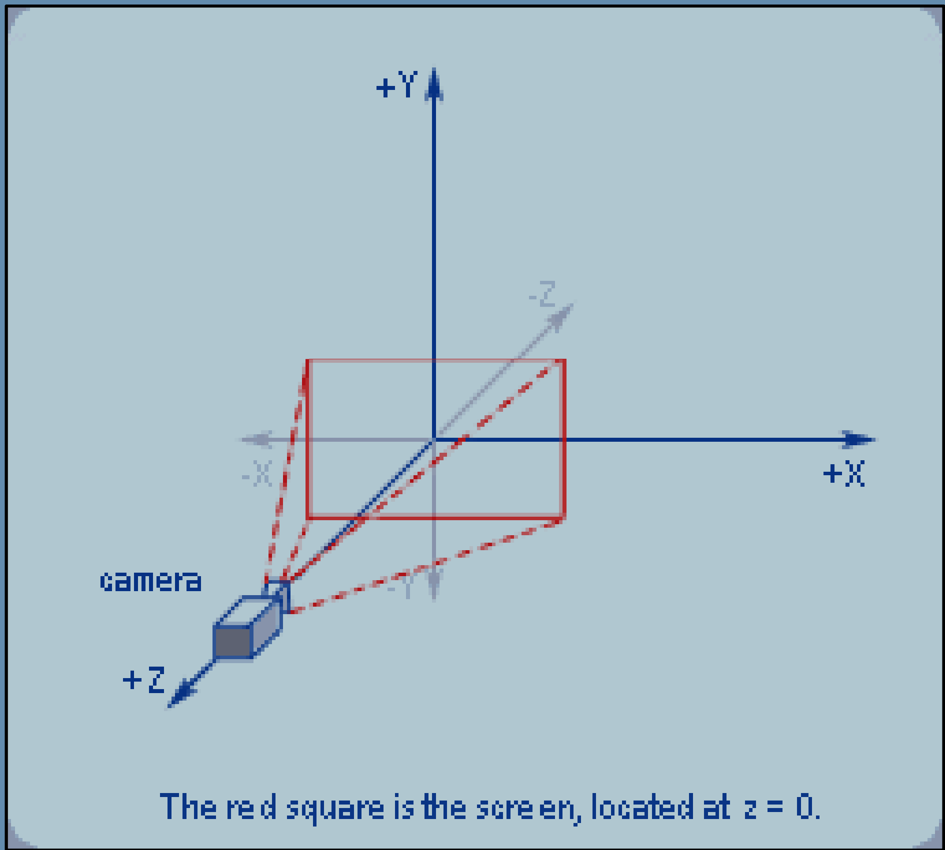


Transformations and Spaces: World Space

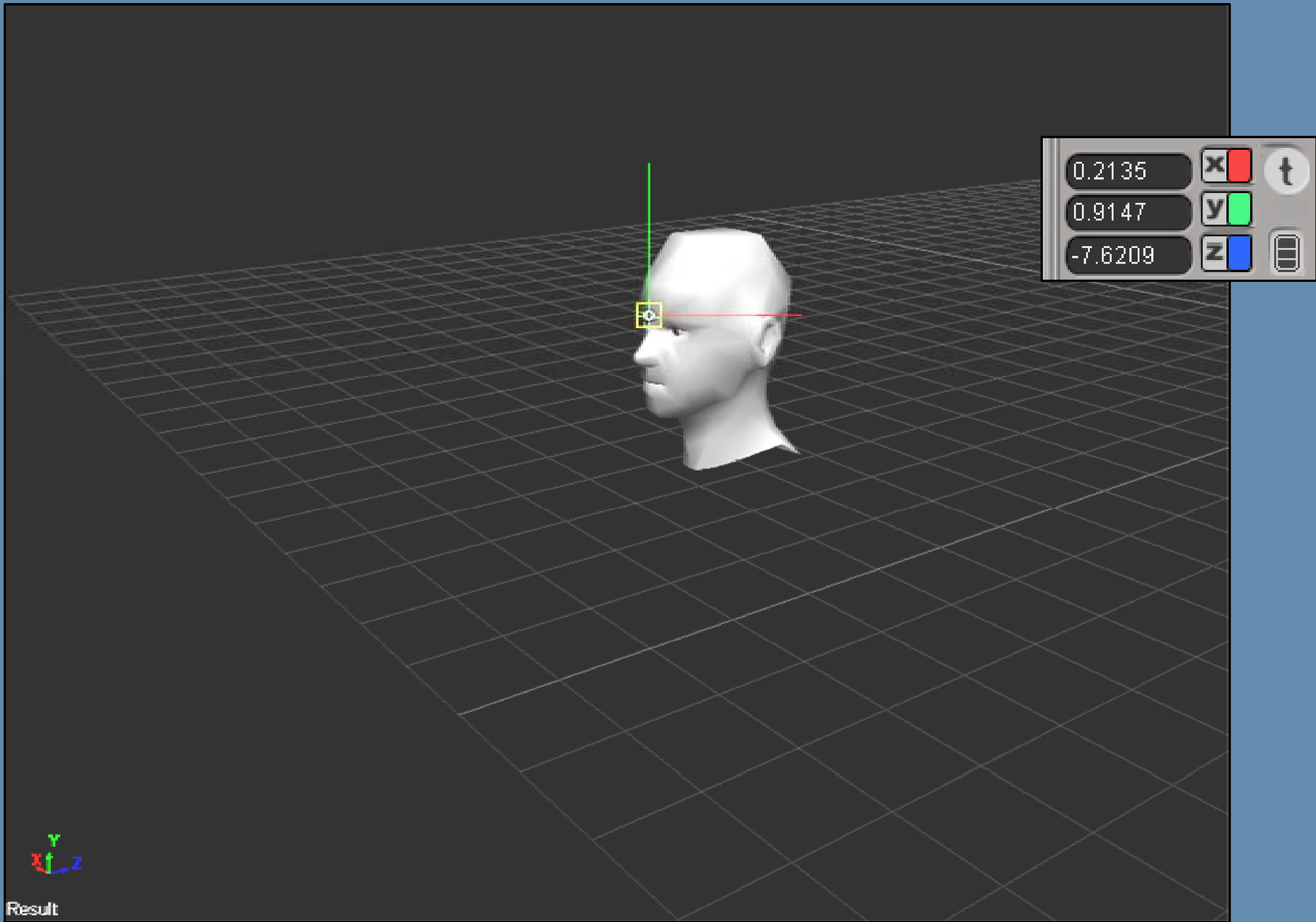


Transformations and Spaces: View Space

- Camera space is centered at the camera's optical center and looks down the z-axis (either positive or negative).
- Last 3D step before 2D projection.

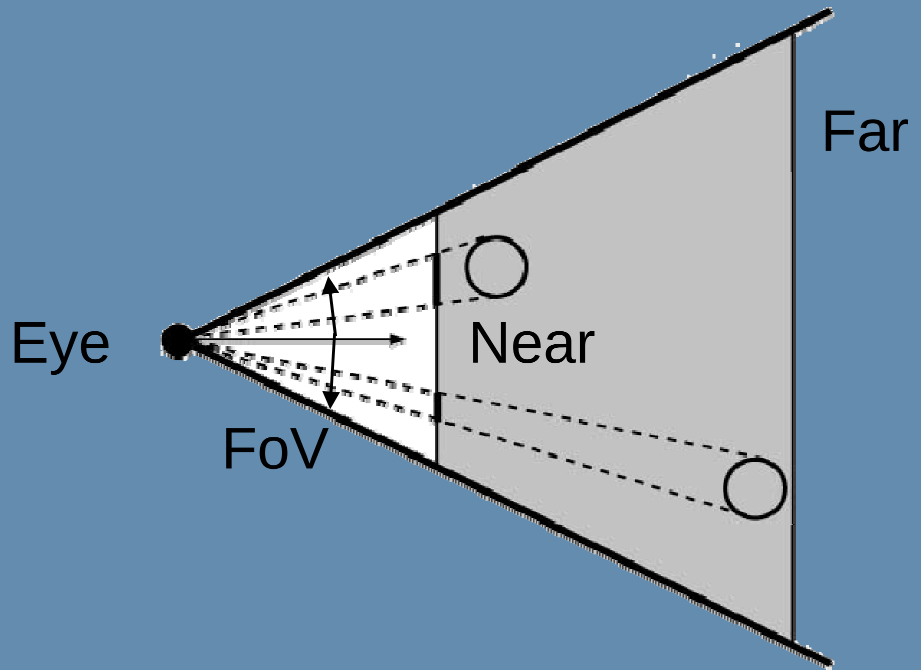


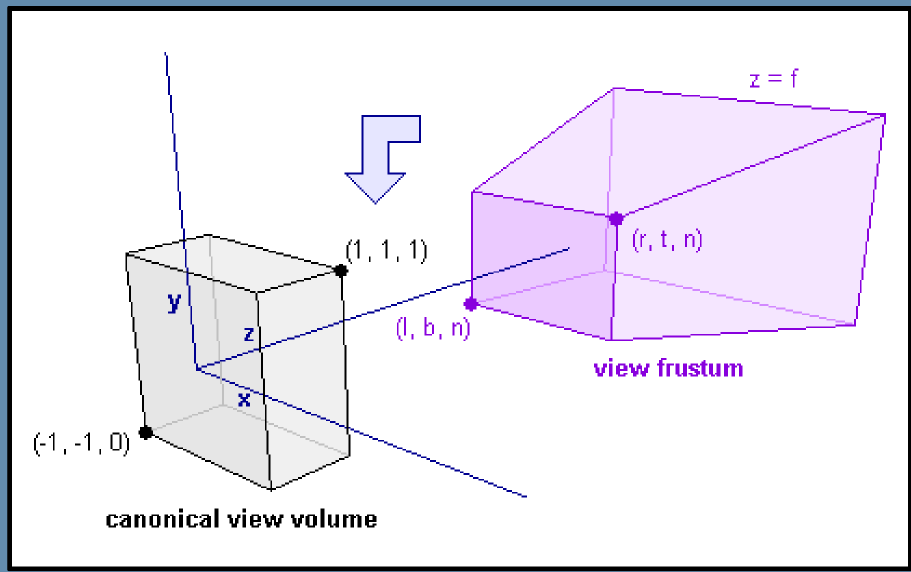
Transformations and Spaces: View Space



Transformations and Spaces: Projection

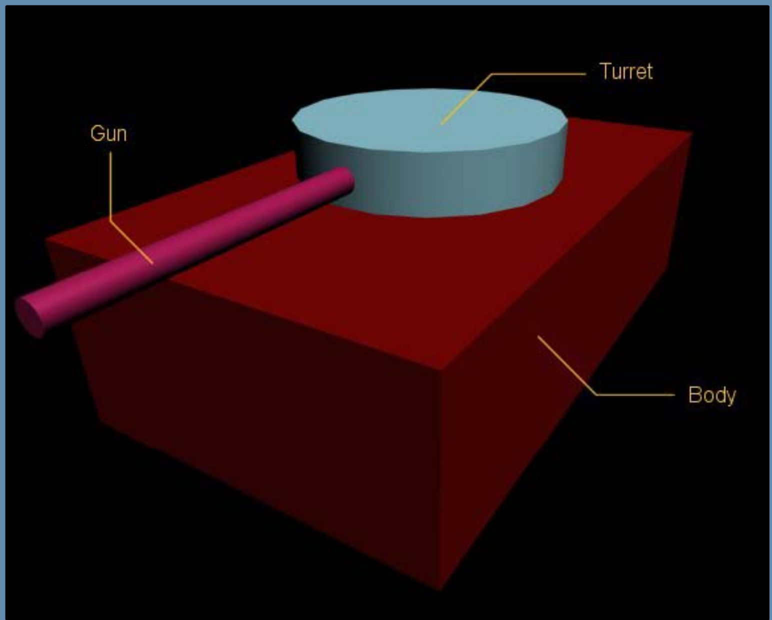
- Simulates physical camera lens properties.
- Transforms 3D coordinates to 2D coordinates with Z remap.
- Visible coordinates range is [-1,1] for X and Y, and [0,1] for Z ([0,-1] in OpenGL).
- One final transform is needed to map [-1,1] to screen coordinates (e.g, [0,640]).

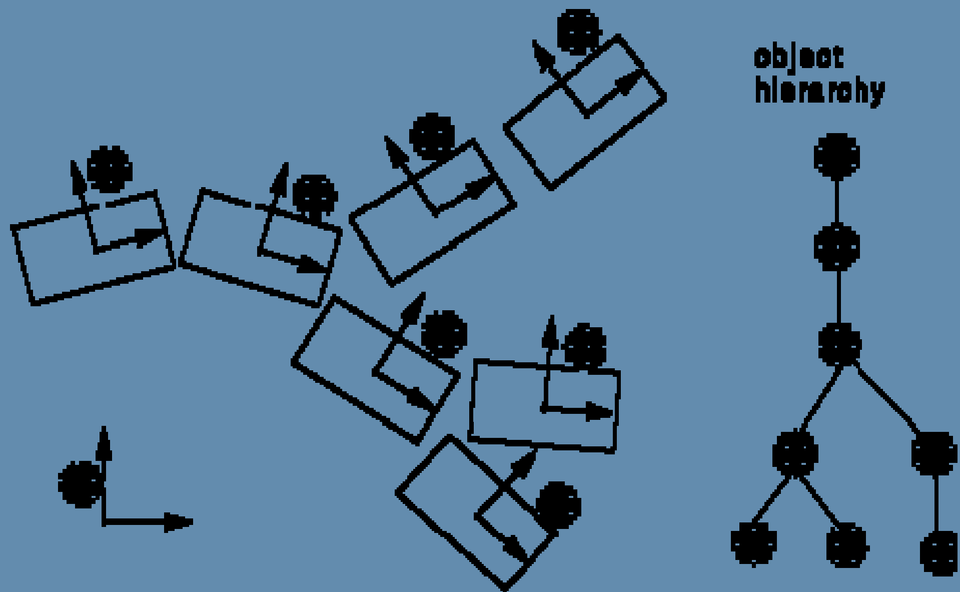




Transformations and Spaces: Matrix Concatenation

- Transforms can be concatenated to form one transform that represents all of them via matrix multiplication.
- Chain them in trees to represent skeleton hierarchies and relationships.





Transformations and Spaces: Matrix Recognition

- A common affine transformation matrix is laid out as below.
- It is possible to extract individual scale/rotation/translation information from such a matrix.

$$\begin{array}{rc} & \begin{array}{cccc} & \text{Orientation} & & \text{Translation} \end{array} \\ \begin{array}{r} \text{x-axis} \\ \text{y-axis} \\ \text{z-axis} \\ \end{array} & \begin{bmatrix} \boldsymbol{scaleX} & \mathbf{0} & \mathbf{0} & \boldsymbol{tX} \\ \mathbf{0} & \boldsymbol{scaleY} & \mathbf{0} & \boldsymbol{tY} \\ \mathbf{0} & \mathbf{0} & \boldsymbol{scaleZ} & \boldsymbol{tZ} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{bmatrix} \end{array}$$

Real-time 3D Computer Graphics Algorithms

- Modeling and Geometry Manipulation
- Rendering Techniques
- Global Effects
- Image Space

Modeling and Geometry Manipulation

- Billboards
- High-Order Surfaces
- Morphing
- Skinning

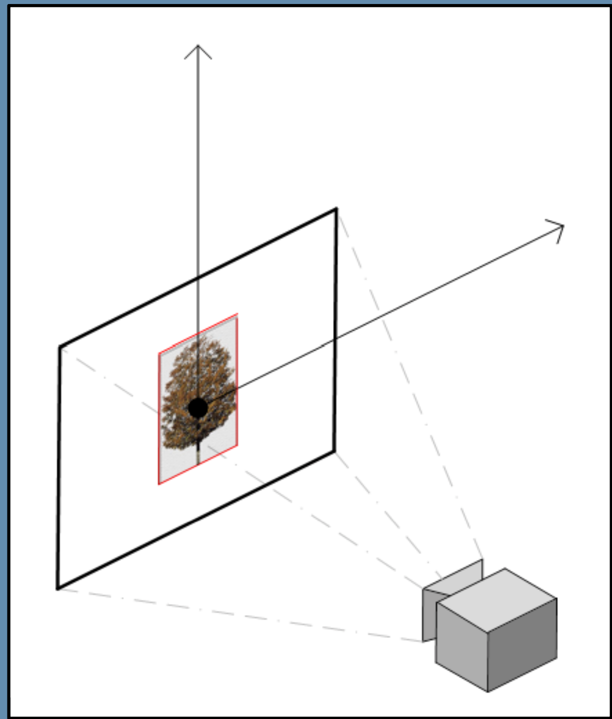
Billboards

- Simple textured quads.
- Always facing the camera.
- Used a lot in rendering trees, and particles in general.

Billboards (cont'd)

- Expand the billboard's position point to a quad in view space's up and right axes:

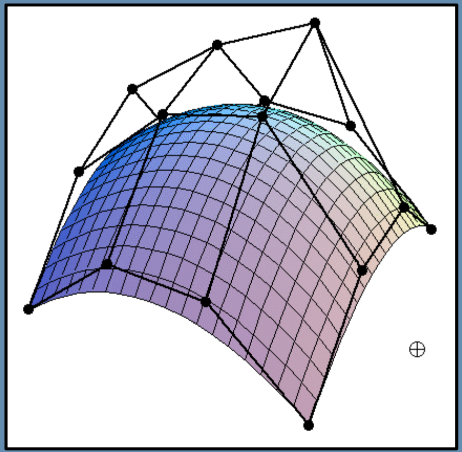
```
billboard.vertices.bottomleft =
    billboard.center +
    camera.up*(-billboard.height/2) +
    camera.right*(-billboard.width/2);

billboard.vertices.bottomright =
    billboard.center +
    camera.up*(-billboard.height/2) +
    camera.right*(+billboard.width/2);

billboard.vertices.topright =
    billboard.center +
    camera.up*(+billboard.height/2) +
    camera.right*(+billboard.width/2);

billboard.vertices.topleft =
    billboard.center +
    camera.up*(+billboard.height/2) +
    camera.right*(-billboard.width/2);
```



High-Order Surfaces: Bezier Patches

- 3D geometry represented by a parametric surface: Bezier cubic patches.
- Control points guide the surface (convex hull). Surface only passes through end points.
- Continuous(*), infinite resolution, compact representation.

* Continuity on boundaries requires special care.

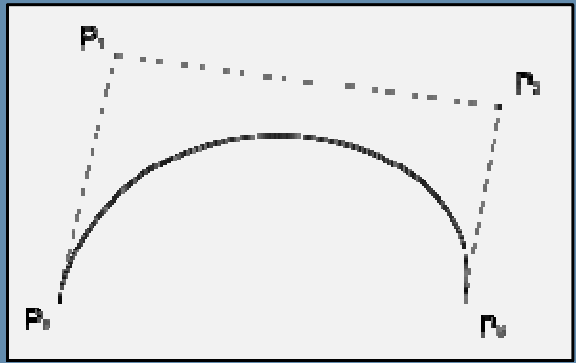


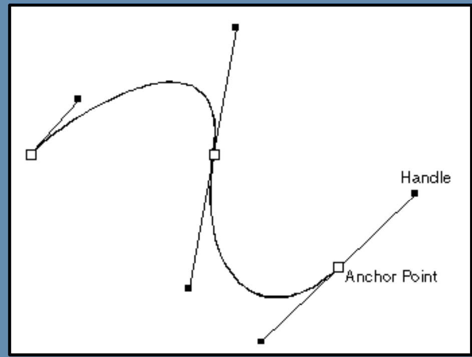
High-Order Surfaces: Bezier Patches (cont'd)

- A quick look on Bezier curve evaluation:

$$Q(t) = \sum_{i=0}^{3} B_i P_i$$

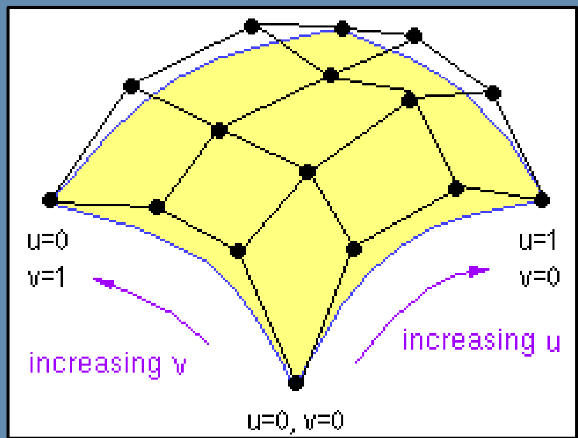
$$B_0 = (1-t)^3$$
$$B_1 = 3t(1-t)^2$$
$$B_2 = 3t^2(1-t)$$
$$B_3 = t^3$$





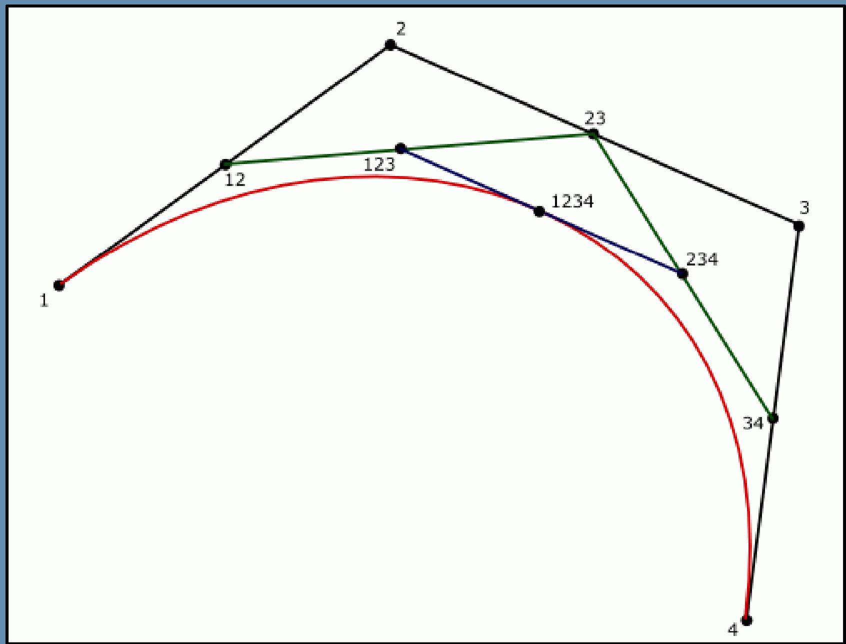
- Cubic Bezier patches can be evaluated with a slightly extended formula:

$$Q(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} P_{i,j} B_i(u) B_j(v)$$



High-Order Surfaces: Bezier Patches (cont'd)

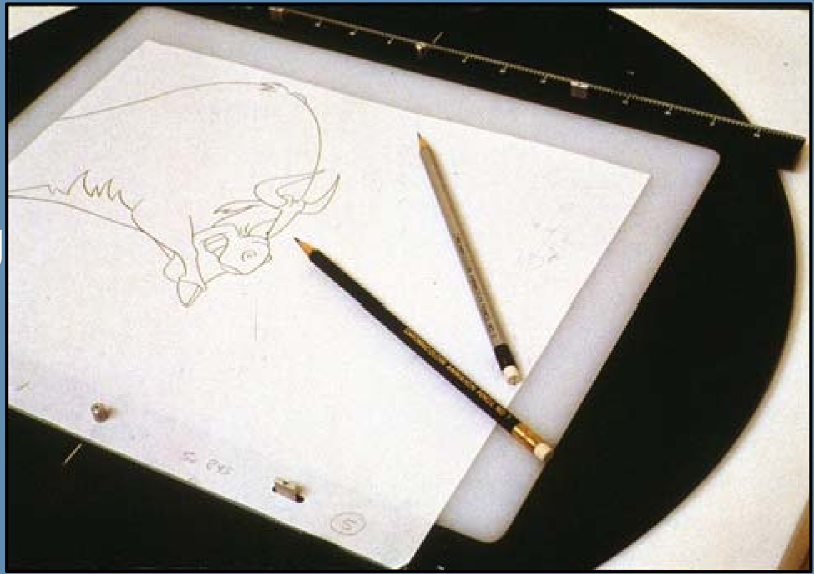
- Bezier curves/patches can be evaluated recursively (Paul de Casteljau).
- A curve can be broken into two other curves of the same degree.

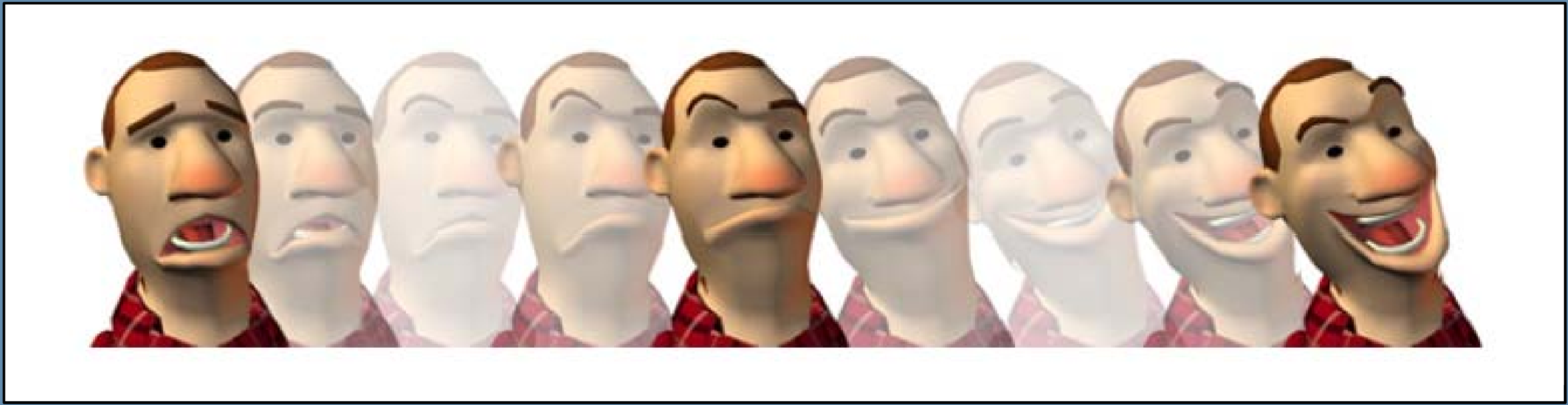


- Can be fast if recursion end criteria is properly determined.
- A game can subdivide until a certain amount of polygons have been generated.

Geometry Morphing (blending/tweening)

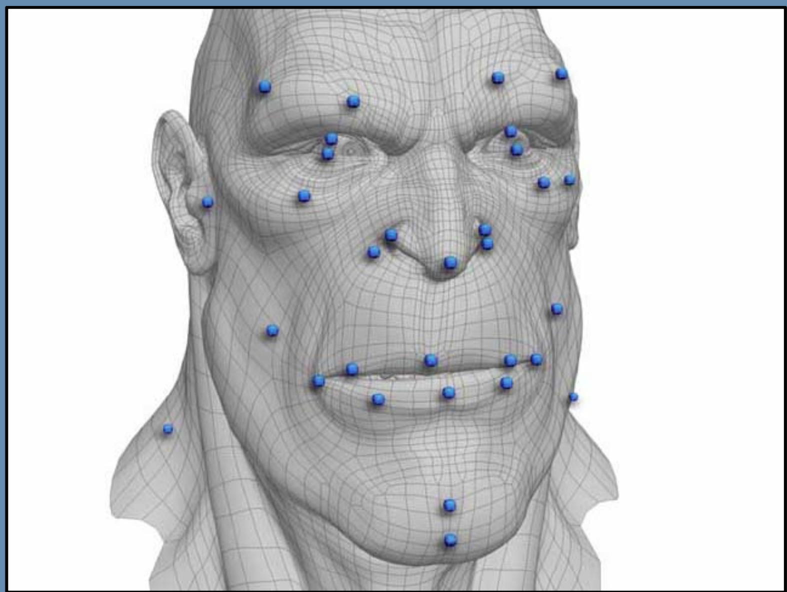
- Similar to the concept of key frames in traditional 2D animation.
- Key geometry frames, sharing the same topology, vertex count and vertex ordering
- Intermediate frames are generated by interpolating between two key frames.
- Flexible deformations.
- Can take a lot of memory, especially for long animations.

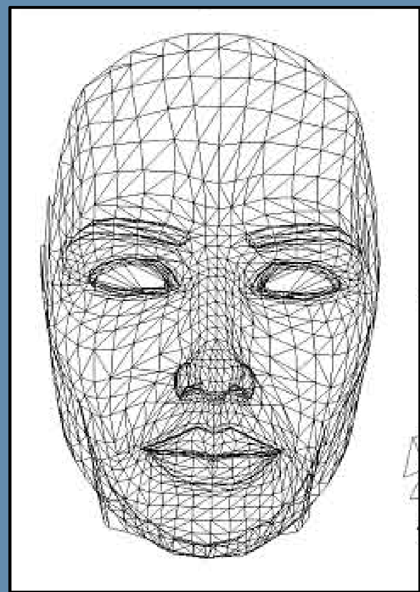




Geometry Morphing (blending/tweening), cont'd.

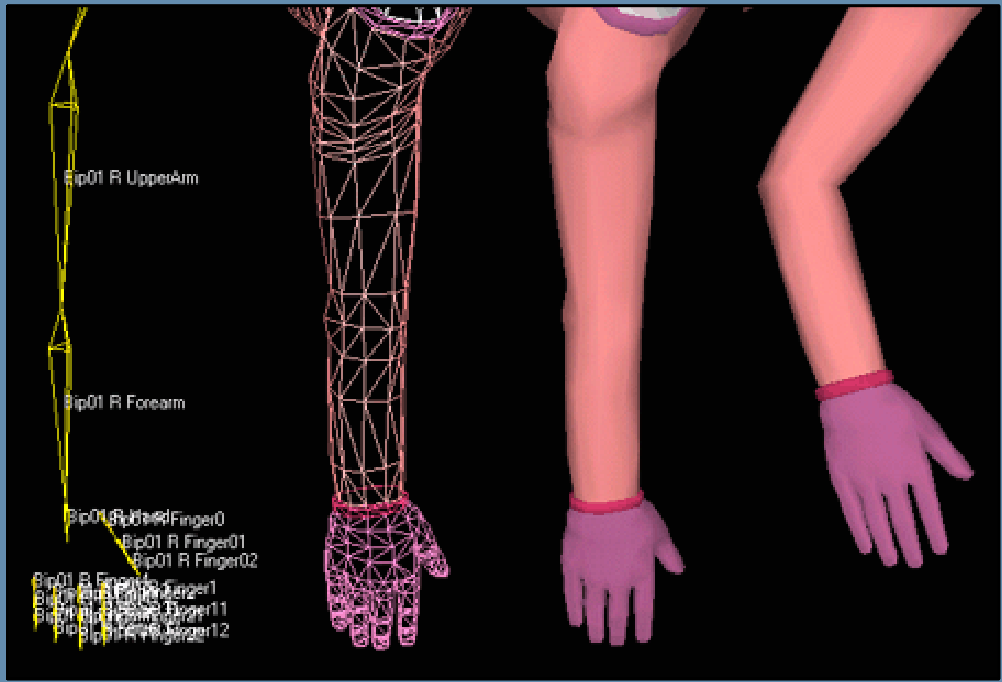
- Done in two methods:
- **Blended**: The final pose is a blend between two keyframes:
  - ```
    for (int i=0; i<Mesh.Vertices.Count; i++)
        Mesh.Vertices[i] = Lerp(keyShape1.Vertices[i], keyShape2.Vertices[i], percentage);
    ```
- **Additive**: The final pose is an accumulation of an open number of relative keyframes (used a lot in facial animation):
  - ```
    for (int i=0; i<Mesh.Vertices.Count; i++)
        Mesh.Vertices[i] = Base.Vertices[i] + Smile.Vertices[i]    * SmilePercentage +
                                              Blink.Vertices[i]    * BlinkPercentage +
                                              Surprise.Vertices[i] * SurprisePercentage;
    ```





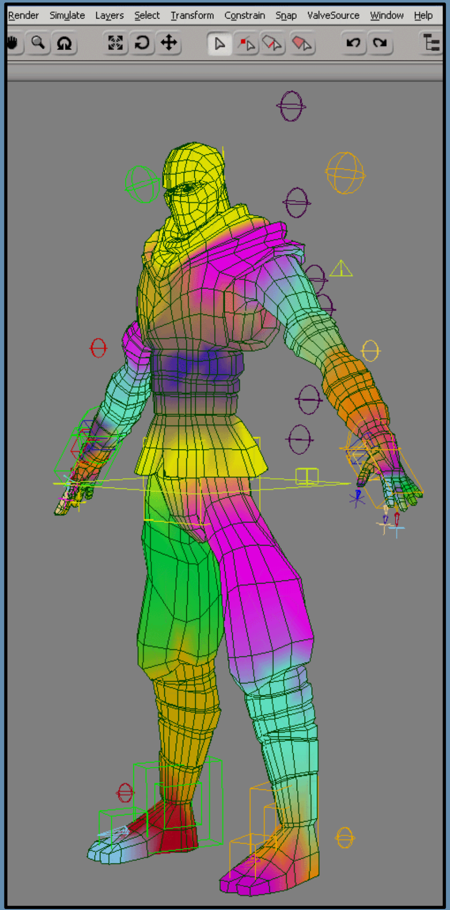
Skinning: Skeletal Animation

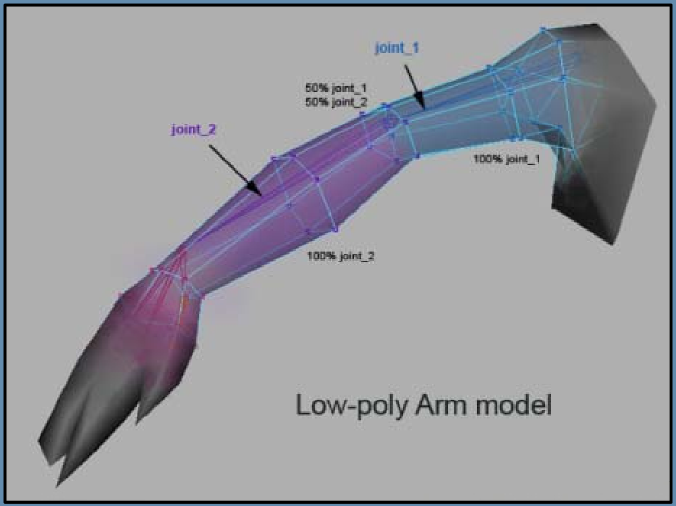
- Geometry deformation based on skeletal animation.
- Geometry is “skinned” over a skeleton and attaches to its bones.
- Vertices expressed relative to their owner bones, or in *bone-space*.
- Transforming a vertex to world space now involves an additional *bone-to-world* matrix (almost always updated every frame from animation).

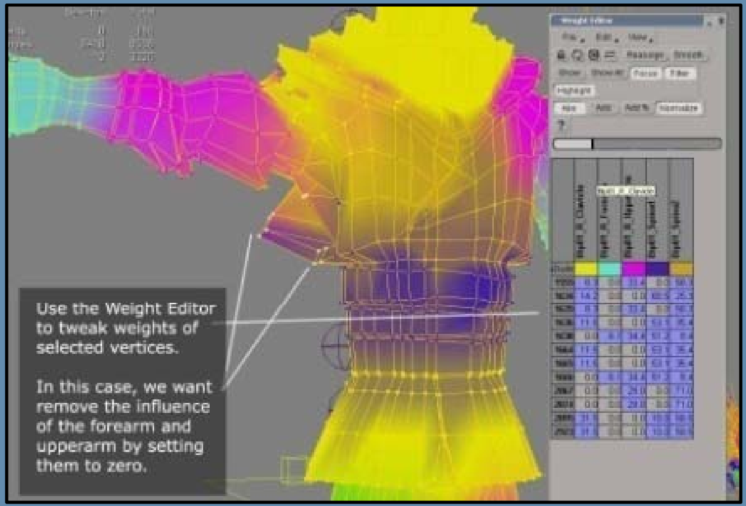


Skinning: Matrix Palette

- A skinned vertex can be attached to more than one *influencing bone* at the same time using different *weights*, which in turn should all sum up to 1.0.
- For practical reasons, the maximum number of influences is usually assumed to be 3 or 4 (mostly 3).
- The limited number of GPU constant registers may prevent fitting all bone matrices to draw the character in a single batch.
- Normals, tangents and binormals must be skinned as well.







Skinning: Sample Code

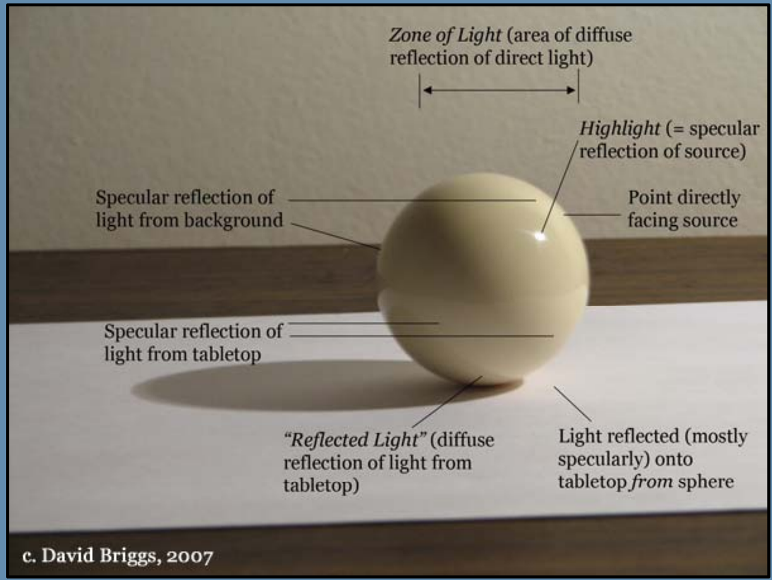
```
#define MAX_INFLUENCES 4
struct SkinnedVertex
{
    float3 boneSpacePos;
    float3 worldSpacePos;
    int    controllingBones[MAX_INFLUENCES];
    float  boneWeights[MAX_INFLUENCES];
}


for each (Vertex v in mesh.Vertices)
{
    v.worldSpacePos = float3(0,0,0);
    for (b=0 to MAX_INFLUENCES)
    {
        Bone bone = mesh.Bones[v.controllingBones[b]];
        v.worldSpacePos += transform(v.boneSpacePos,bone.localToWorldMatrix) *
                           v.boneWeights[b];
    }
}
```

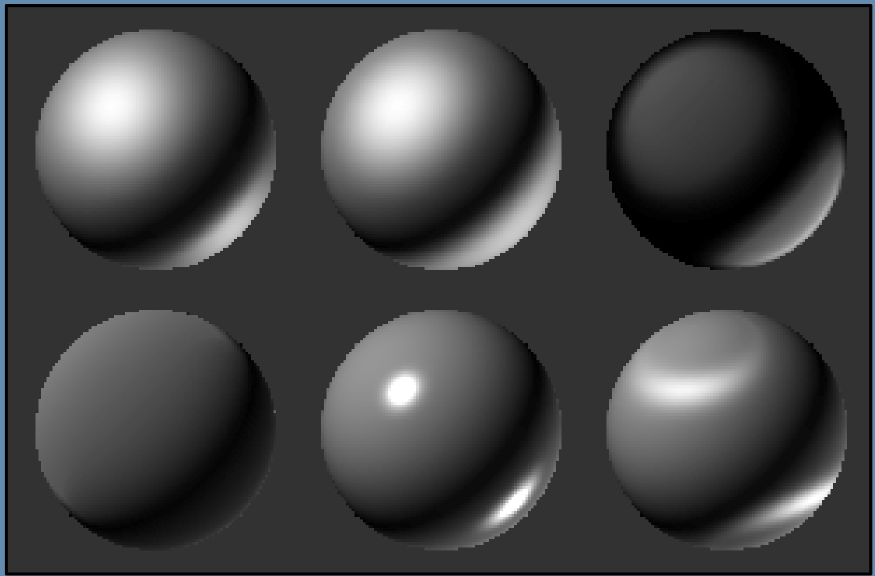
Rendering Techniques

- Materials and Lighting
- Texture Mapping
- Fog
- Translucency and Transparency
- HDR Rendering

Materials & Lighting

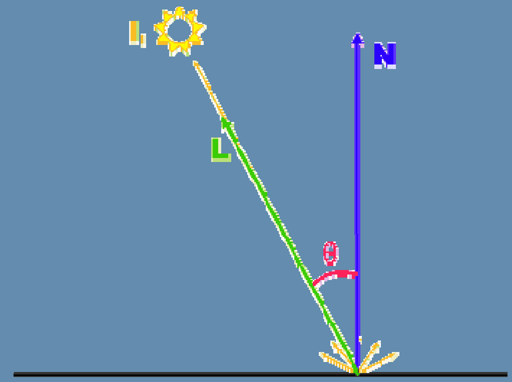
- Materials are identified based on their surface properties (e.g. smoothness/roughness) and the way they interact with light (how we perceive them).
- Real-time rendering uses simplified formulas that empirically match a certain material's properties (simplification of BRDFs).
- Some key models: Lambert, Phong (or Blinn), Strauss, Cook-Torrance, Oren-Nayar, Anisotropic, …

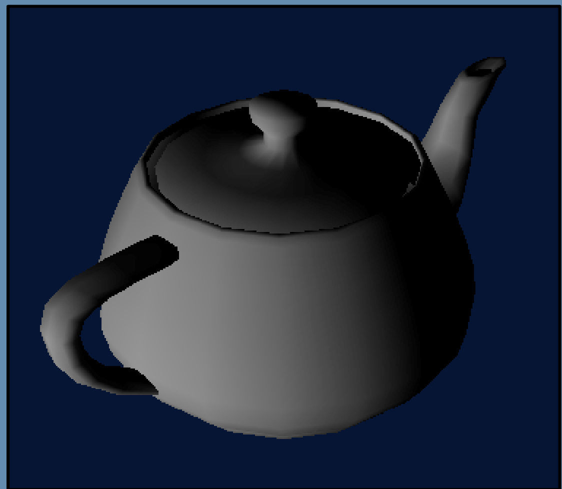


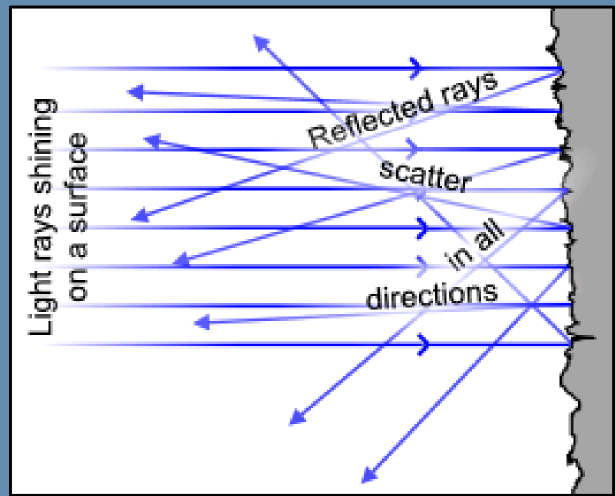


Materials & Lighting: Lambert Shading

- Simulates micro-roughness on surfaces ⇒ light diffusion.
- Simplified to the angular relationship between surface normal and incoming light direction.
- `shade = cos(θ)`
  or by getting rid of the trigonometry stuff:
  `shade = N.L`
- Usually referred to as *The Diffuse Component*.

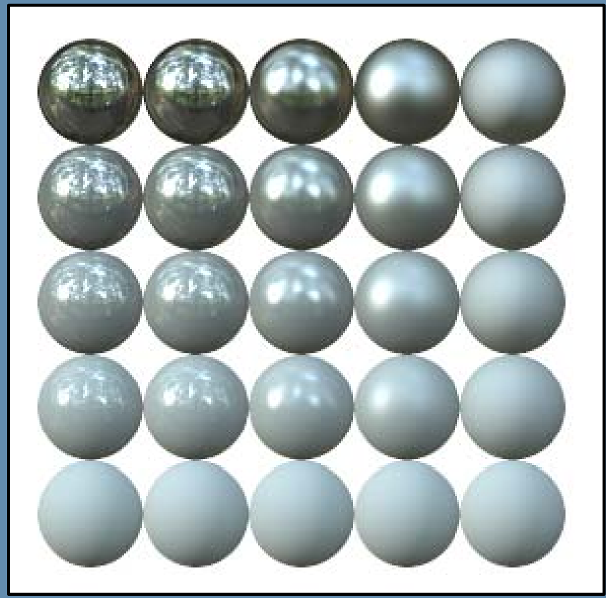


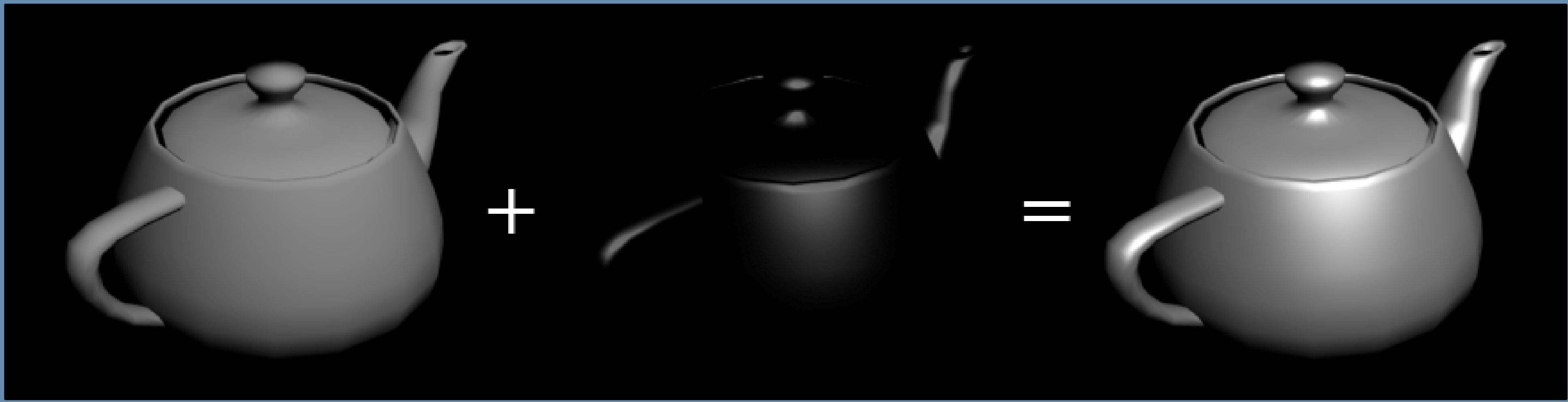




Materials & Lighting: Phong/Blinn Shading

- Building on Lambert, adds a highlight component.
- Mimics reflection of the light source.
- Function to view direction.



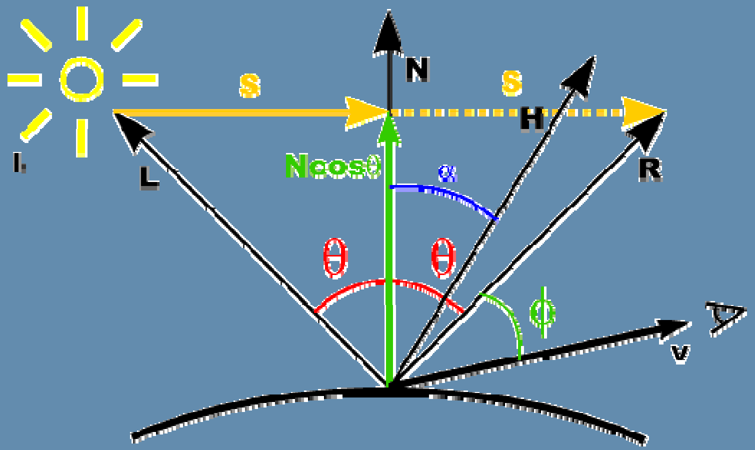


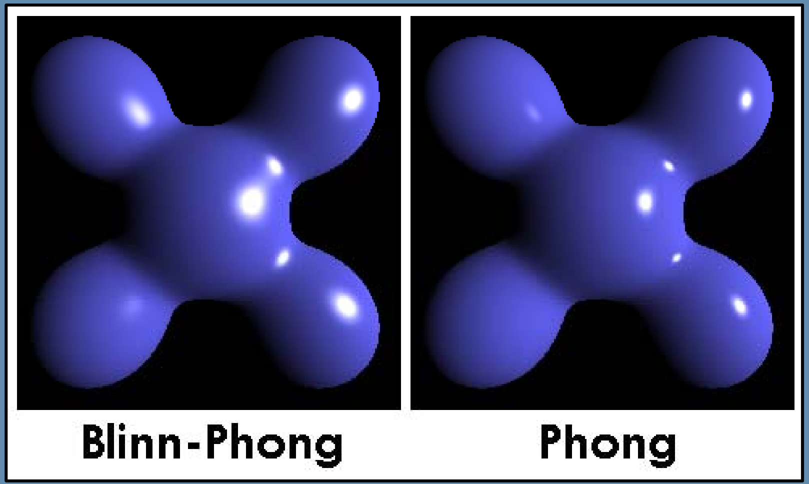
Materials & Lighting: Phong/Blinn Shading

- Calculated as:
  `specular = (R.V)`^{n} *(Phong)*
  where: `R = 2.0×(N.L)×N – L`

- Calculating *R* is slightly heavy. But $\widehat{NH}$ seems close enough:
  `specular = (N.H)`^{n} *(Blinn)*

  where: $$\boxed{H = \frac{L+V}{|L+V|}}$$

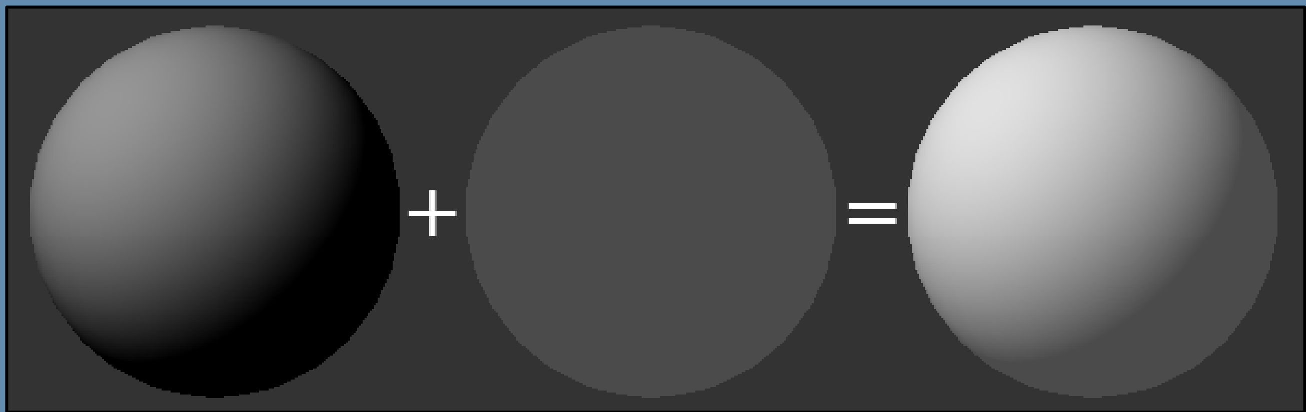




Materials & Lighting: Ambient

- Simulates light scattering in the environment, which results in surfaces lit by indirect light rays.
- Overly simplistic representation:
  Add a constant color!
- The formula thus far is:

$$I_p = k_a i_a + \sum_{\text{lights}} (k_d (L \cdot N) i_d + k_s (R \cdot V)^{\alpha} i_s).$$

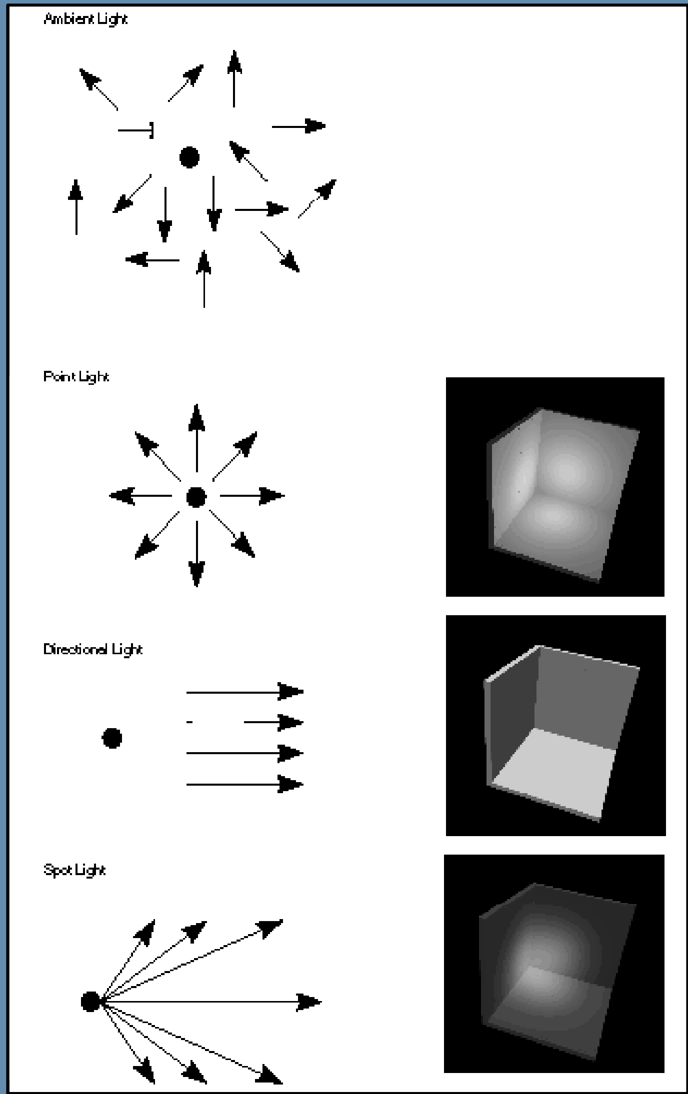


Materials & Lighting: Light Types

- The type of light dictates how its direction and color are calculated during lighting.

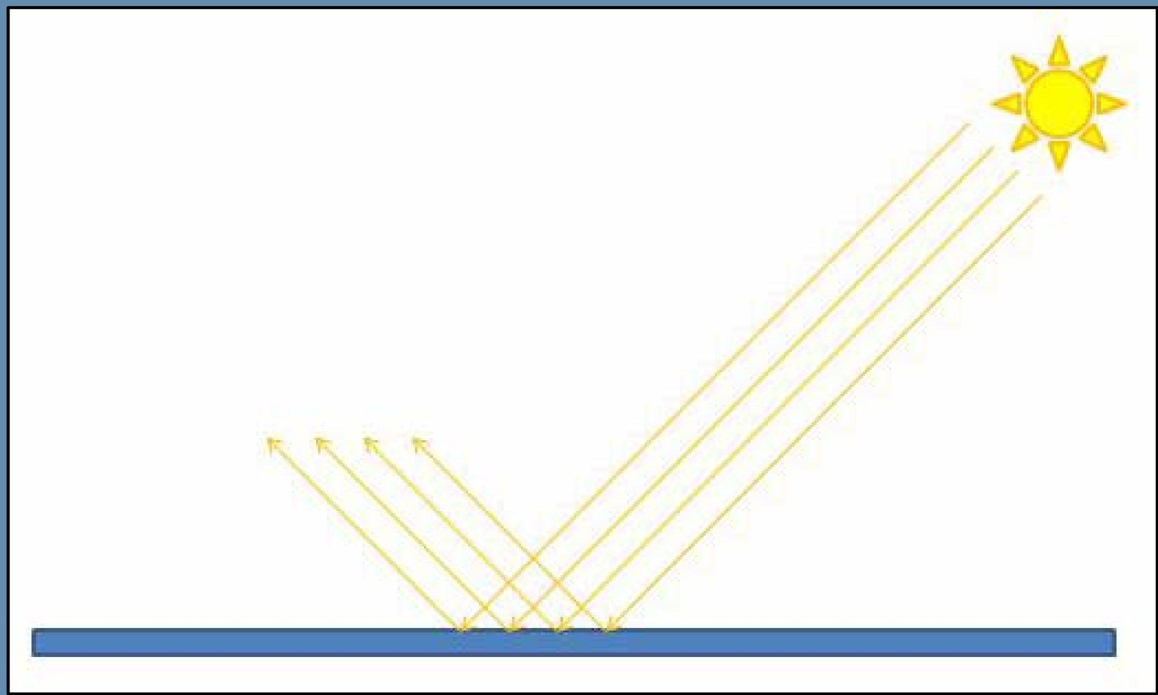
- Common light types:
  - Ambient
  - Directional
  - Point
  - Spot

- Extensions:
  - Hemisphere
  - Image-based
  - Spherical harmonics



Materials & Lighting: Directional Lights

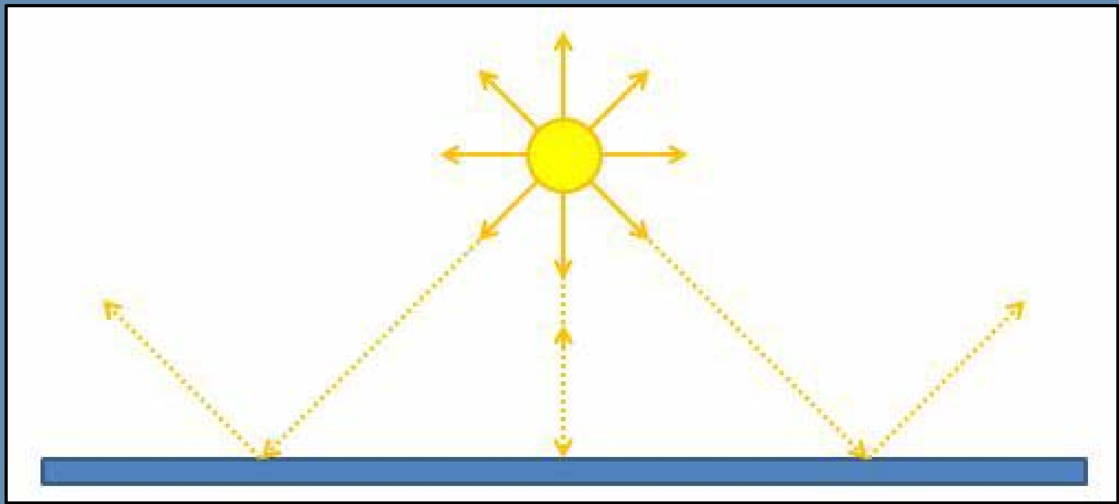
- Single color.
- Parallel rays lighting every point in the whole scene equally.
- Has direction, but no position.
- Useful for simulating sun light.
- Simply represented by the calculation `clamp(N.L)`.



Materials & Lighting: Point Lights

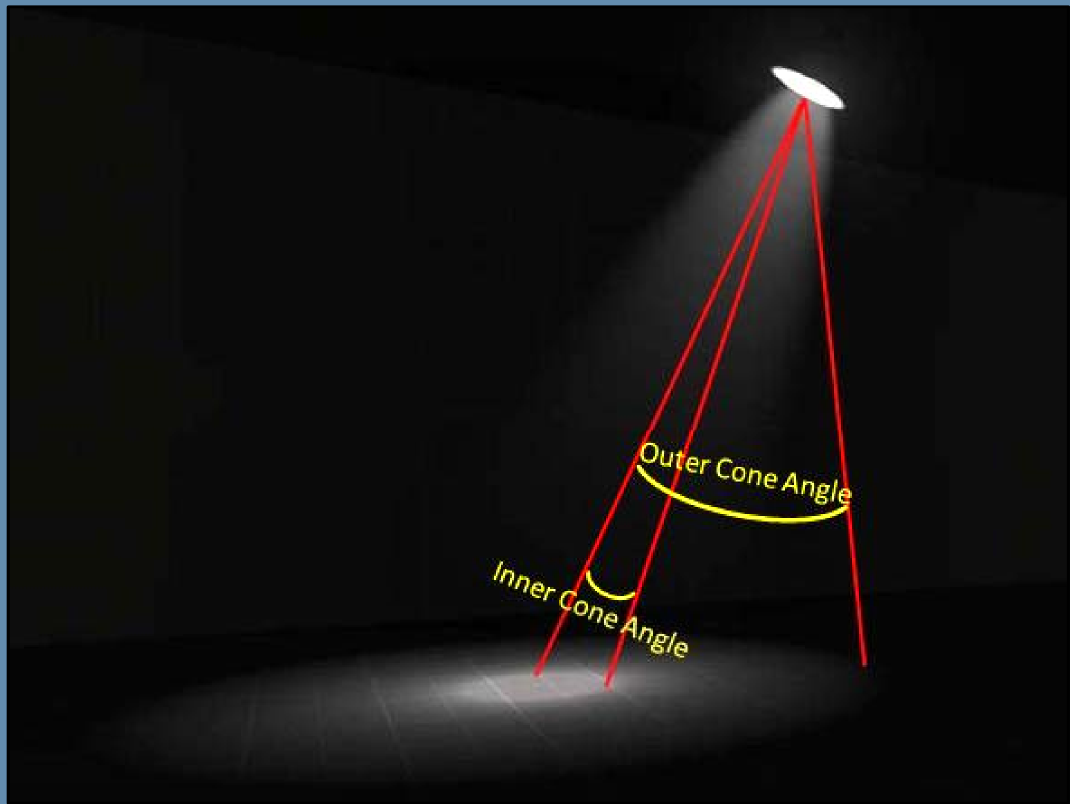
- Single color.
- Rays radiating equally in every direction.
- Has position, but no direction.
- Attenuation based on point distance from light.
- Sample Calculation:

```
float3 RangedDistance = (LightPosition - PointPosition) / LightRange;
float Attenuation = saturate(1.0f - lenSq(RangedDistance));
```



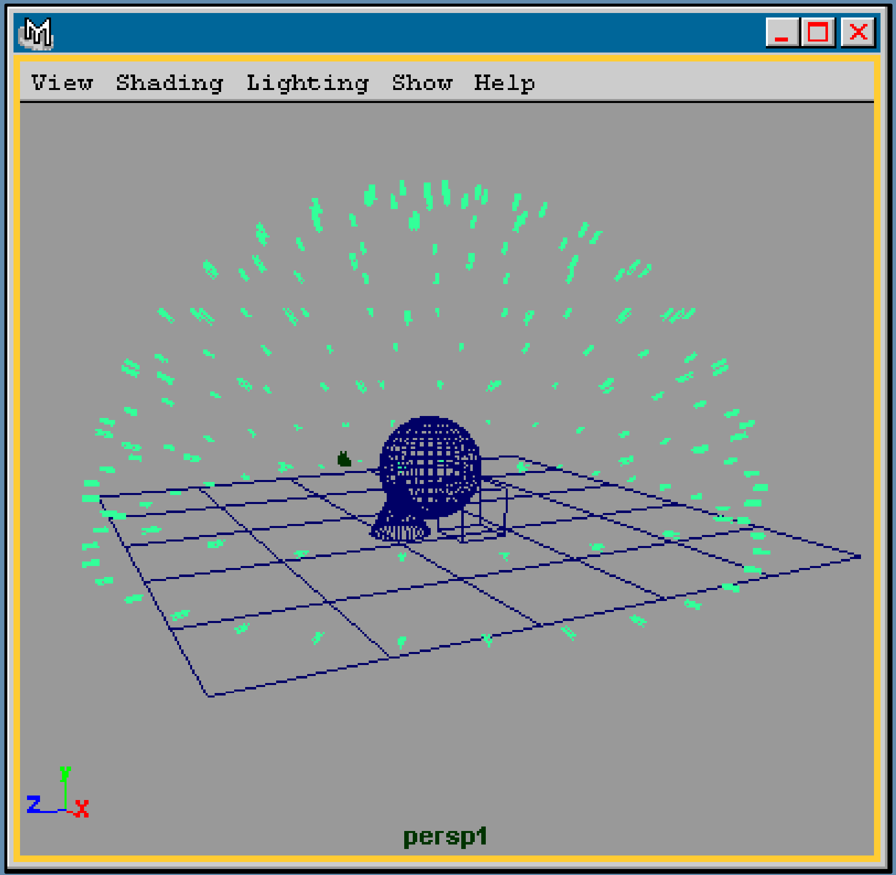
Materials & Lighting: Spot Lights

- Single color.
- Has position and direction!
- Rays radiating within a certain cone with falloff near the edges.
- Spot attenuation is the percentage between (the angle formed between spot direction and ray direction), and (the outer angle minus the inner angle).



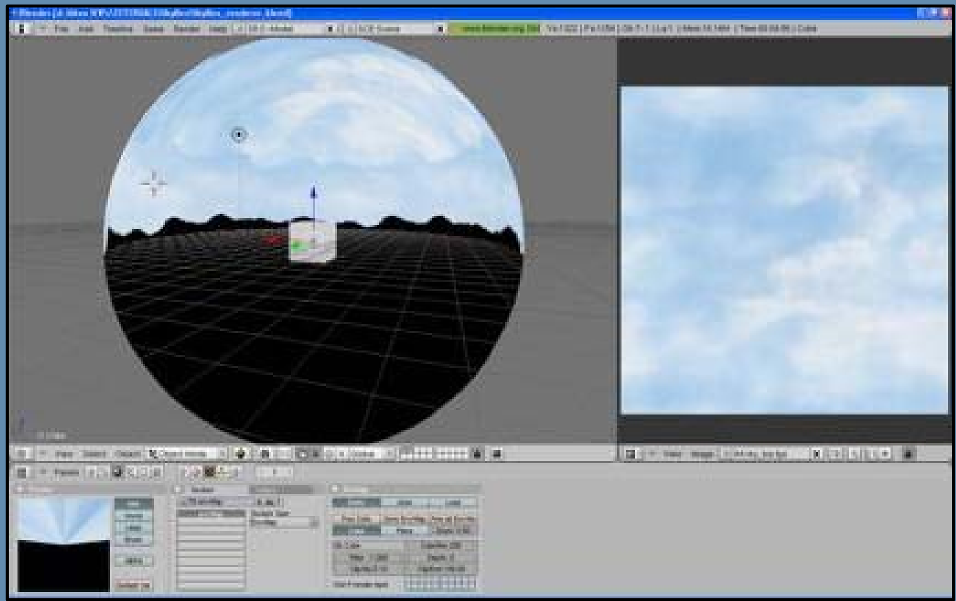
Materials & Lighting: Hemispherical Lighting

- A sphere surrounds the object.
- Light color is a function of polar angles.
- Can be simulated through “many” primitive lights too.



Materials & Lighting: Image-based Lighting

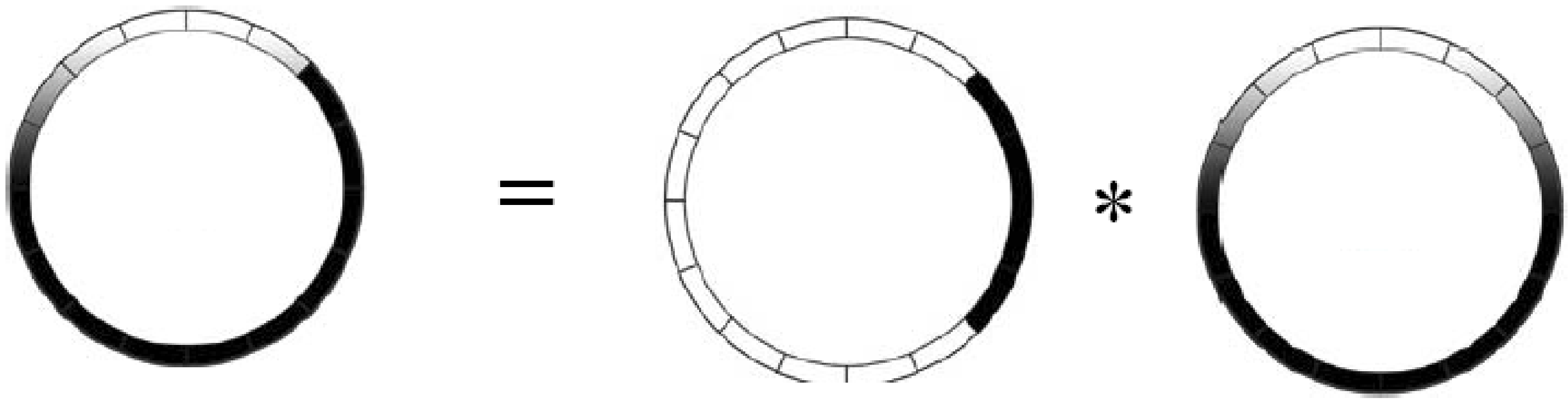
- A textured sphere/cube surrounds the object.
- Light color is a function of polar angles.
- Image reflects the environment of the object.
- Positionless, direction-based.



Materials & Lighting: Spherical Harmonics (1)

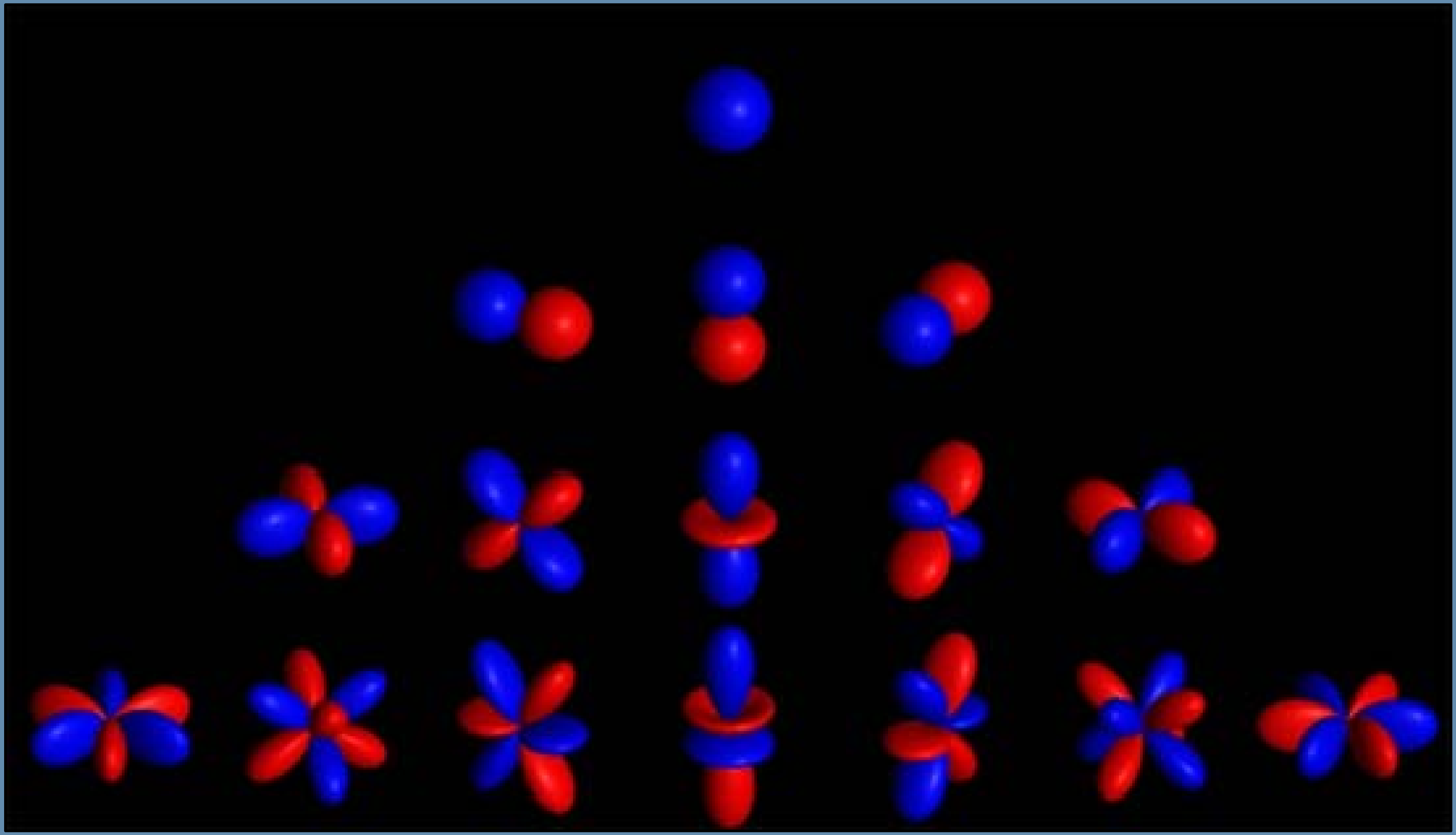
- Precompute lighting response for geometry points over a surrounding sphere.
- Include lighting and visibility calculated by an advanced renderer.
- Calculations made per-vertex or per-texel.

$$T_p(s) = V_p(s) H_{N_p}(s)$$



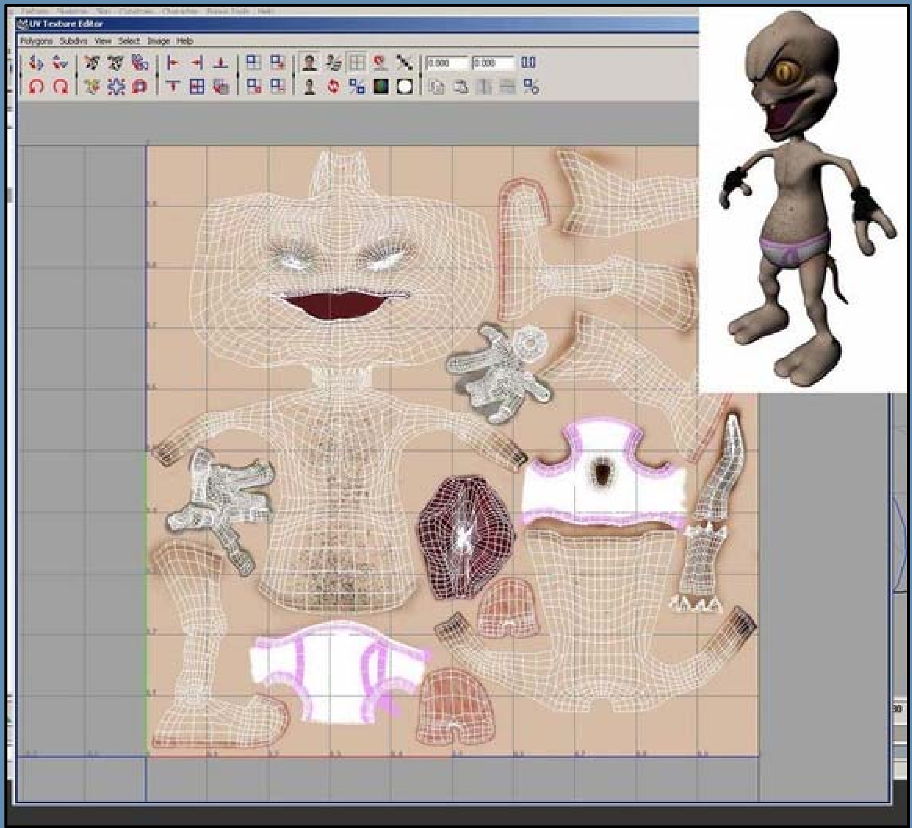
Materials & Lighting: Spherical Harmonics (2)

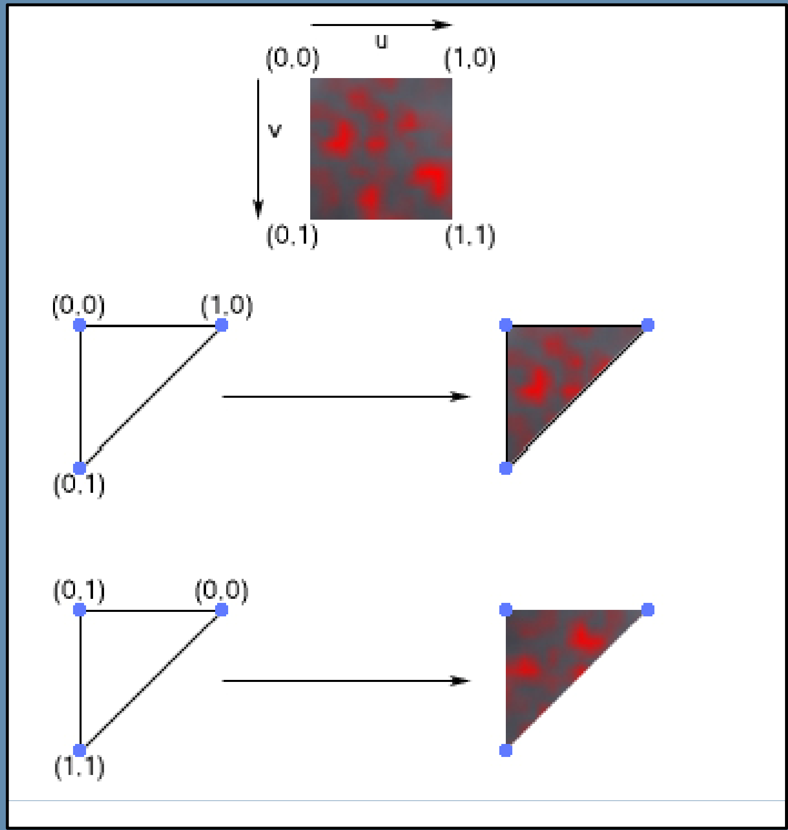
- Compress and store info as spherical harmonics coefficients.
- 9 coeffecients for every light channel for 3rd order spherical harmonics.
- Shader needs only a number of dot-product operations with the light direction to retrieve the matching value.



Texture Mapping

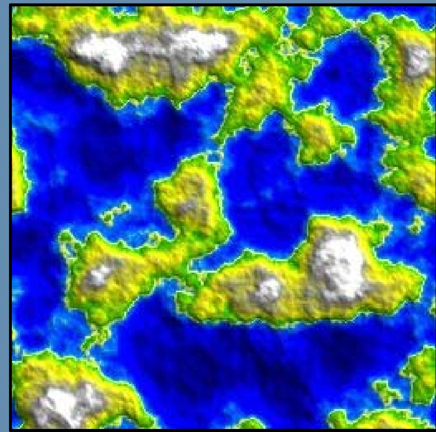
- Adding color detail to geometry with less memory.
- Color information taken from an image, and rasterized to cover triangle areas.
- Textures on a triangle are addressed via normalized UV values stored in each vertex.





Texture Mapping (cont'd)

- Textures can be 1D, 2D, 3D or cube (six faces).
- They can contain stored images, or be procedurally evaluated at runtime (e.g. noise, fractals).
- Hardware imposes certain restrictions in terms of capability/performance (e.g. dimensions and format).

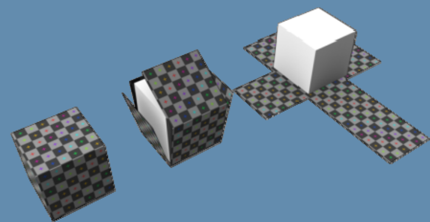


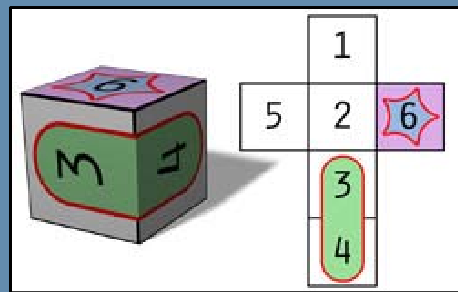
- Sampling an image texture at (U,V):

```
x = (int)(U * texWidth);
y = (int)(V * texHeight);
color = texMem[y * texHeight + x];
```

- Or in HLSL:

```
color = tex2D(texSampler,texCoord);
```

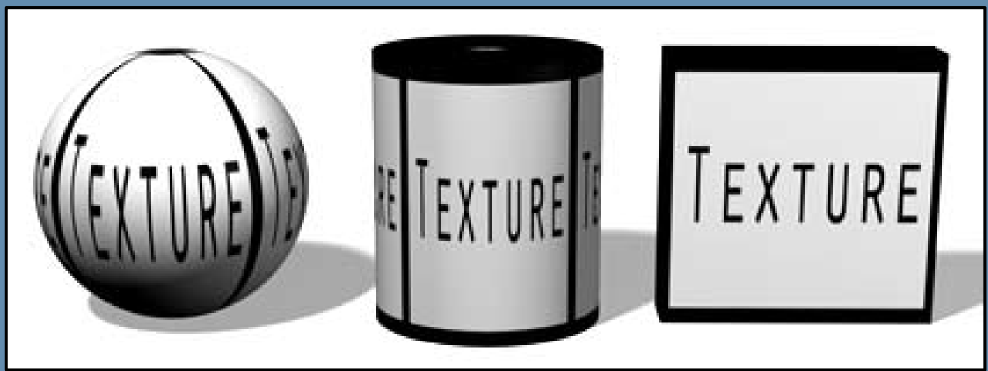


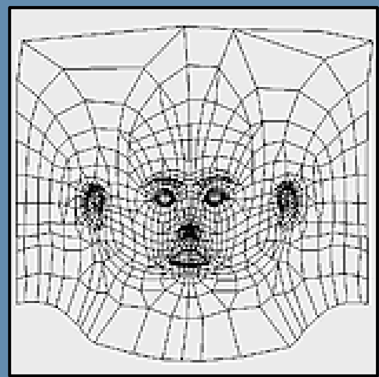


Texture Mapping: UVW Mapping/Projection

- Assigning UV/UVW values to vertices depends on the required results.
- Some simple procedural UV mapping methods:
  - Spherical
  - Cylindrical
  - Planar
- In general, they are hand-authored and stored in the mesh's vertices.
- Planar UV generation example (XZ plane):


```
for each (Vertex vertex in mesh.Vertices)
{
  vertex.texU = vertex.posX * scaleU + offsetU;
  vertex.texV = vertex.posZ * scaleV + offsetV;
}
```

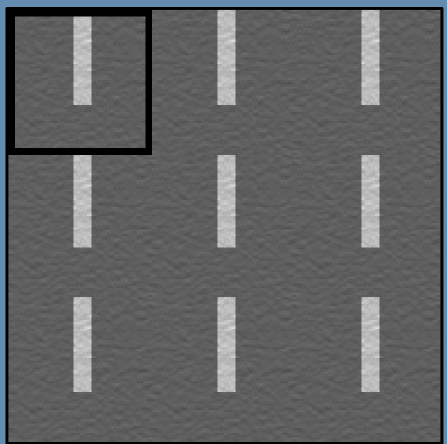




Texture Mapping: Addressing

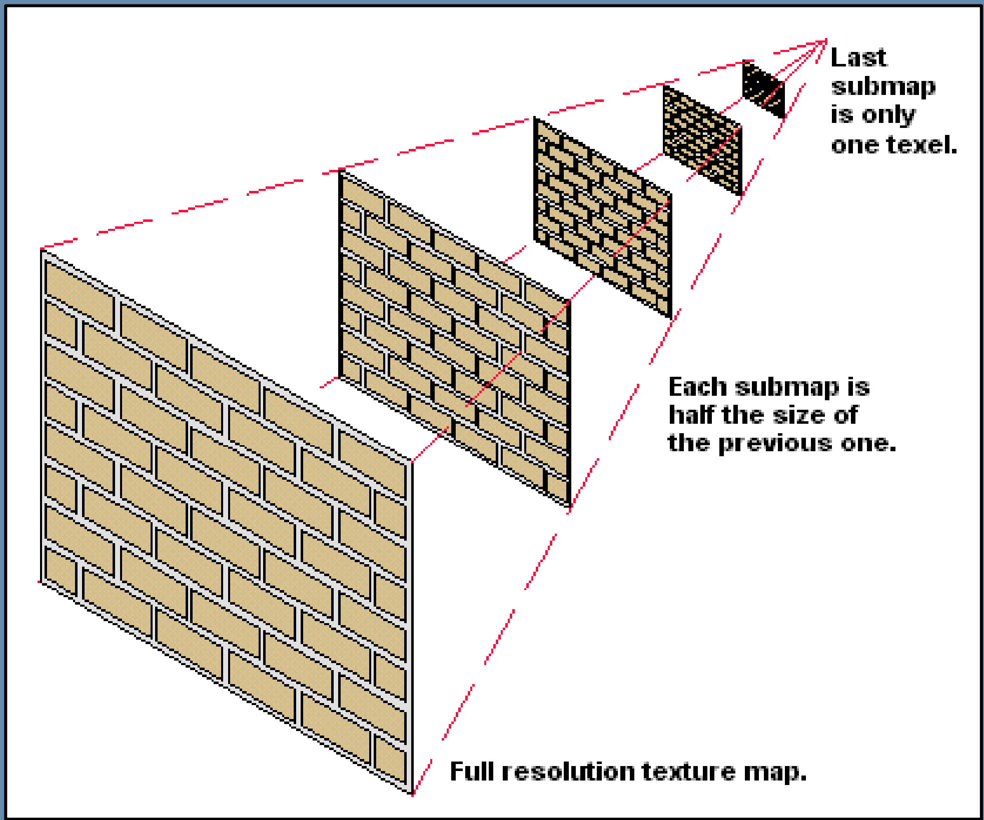
- What should happen when the value of U or V is outside of [0,1] ?
  - Wrap (Repeat)
  - Mirror
  - Clamp
  - Border
- Setting texture addressing mode in HLSL:

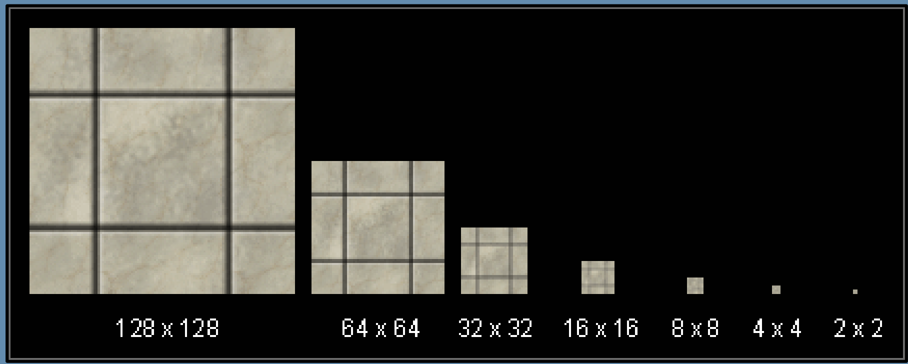
```
sampler mySampler = sampler_state
{
    Texture = <g_Texture>;
    AddressU = Wrap;
    AddressV = Clamp;
    AddressW = Mirror;
};
```




Texture Mapping: Mipmaps

- When sampling the texture for distant objects, artifacts and inefficiencies occur due to undersampling and cache-misses (e.g. reading from a 512x512 image to cover only 25 pixels).
- Mipmaps are a continuous series of half-sized images associated with the texture (pyramid).



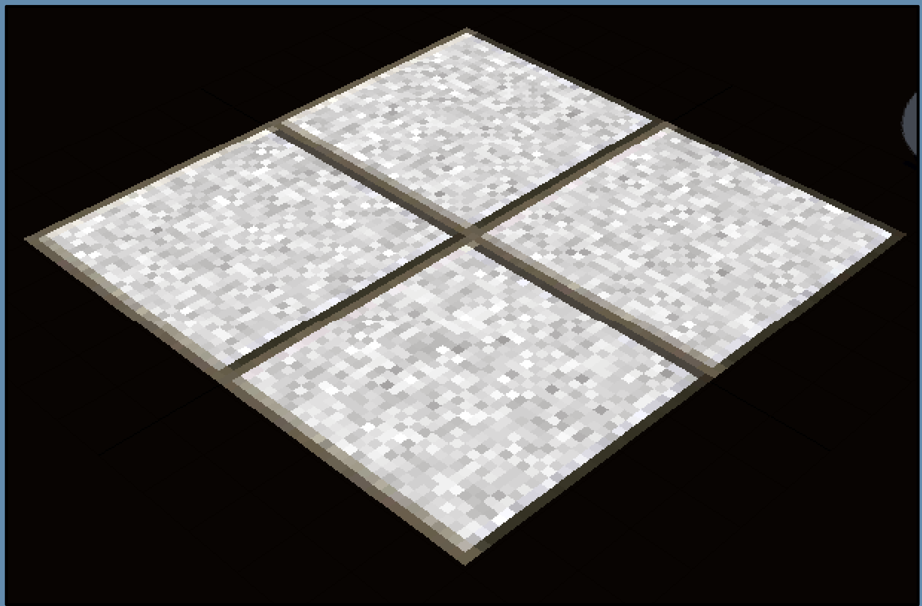


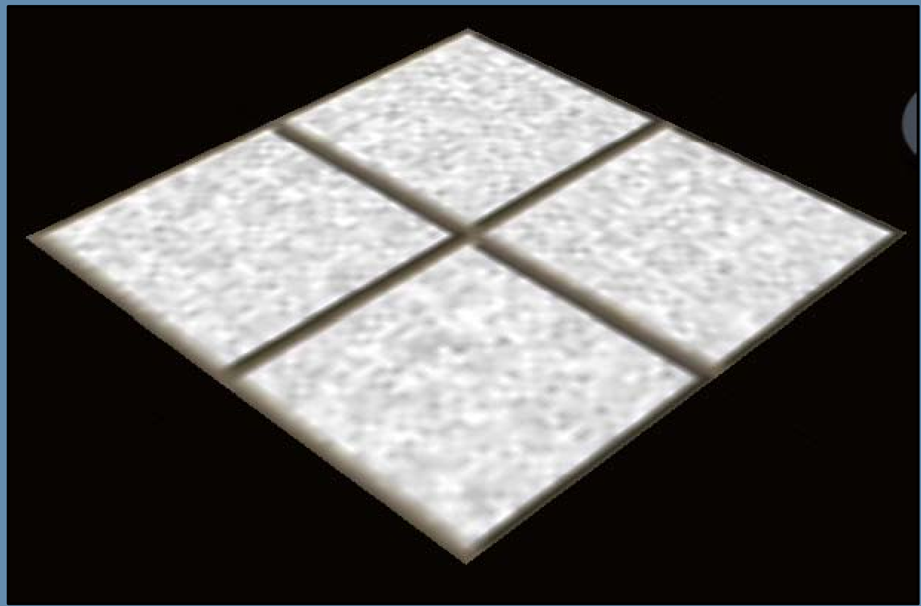
Texture Mapping: Mipmaps (cont'd)

- GPU picks the suitable mipmap to texture the area in question depending on difference of UV values between pixels.
- Mipmaps are usually auto-generated by downsampling the full-resolution texture sequentially, but they can contain totally different images too (for special effects).

Texture Mapping: Filtering

- Textured geometry rarely map textures at 1:1 pixel:texel ratio.
- Textures need to be minified/magnified during sampling.
  - Point sampling: Pick nearest neighbor.
  - Linear filtering: Weighted blend between adjacent texels (box filtered).
  - Anisotropic filtering: Weighted with anisotropic kernel based on slope, and across different mipmaps.





Texture Mapping: Filtering (cont'd)

- Filtering can be specified for each case differently:
  - Magnification
  - Minification
  - Mipmapping

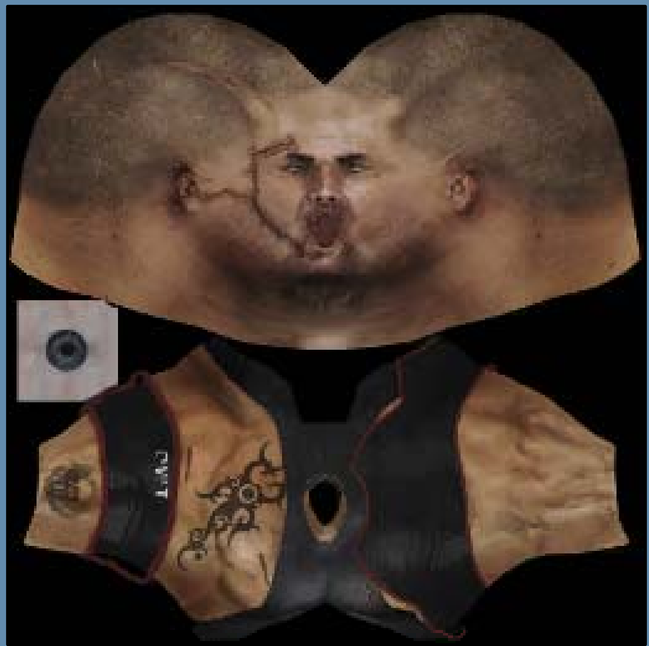
- Common filtering settings:
  - Point: Point Min/Mag/Mip
  - Bilinear: Linear Min/Mag, Point Mip.
  - Trilinear: Linear Min/Mag/Mip.

- Setting texture filtering mode in HLSL:

```
sampler mySampler =
sampler_state
{
    Texture = <g_Texture>;
    MagFilter = Linear;
    MinFilter = Linear;
    MipFilter = Point;
};
```

Texture Mapping: Diffuse Texturing

- Provide detailed color information within geometry polygons (Albedo).
- Diffuse maps are usually unlit, as real-time lighting is applied later, but small bump shadows can be included.
- May have an alpha channel to dictate translucency/transparency.



Texture Mapping: Bump Maps

- Bump maps (a.k.a height maps) provide detail to geometry normals by specifying values of normal *perturbation*.
- Normal at every texel is found by determining slope angle in relationship with surrounding texels.
- Bump map normal is added to surface normal.
- Bump map is stored in a single color channel.

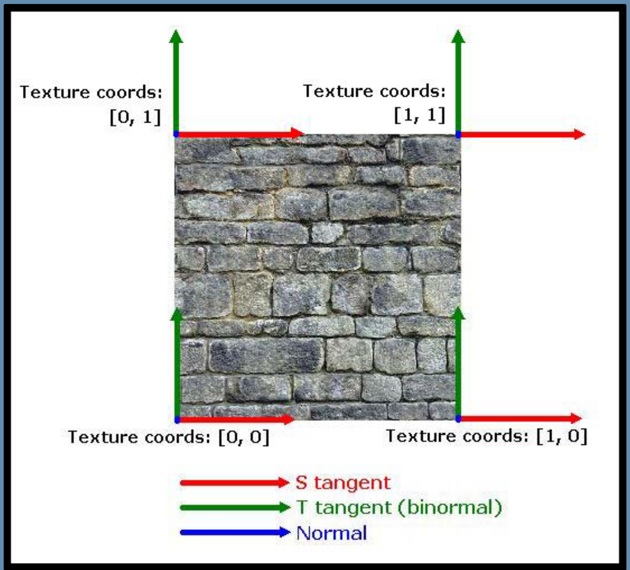
Texture Mapping: Normal Maps

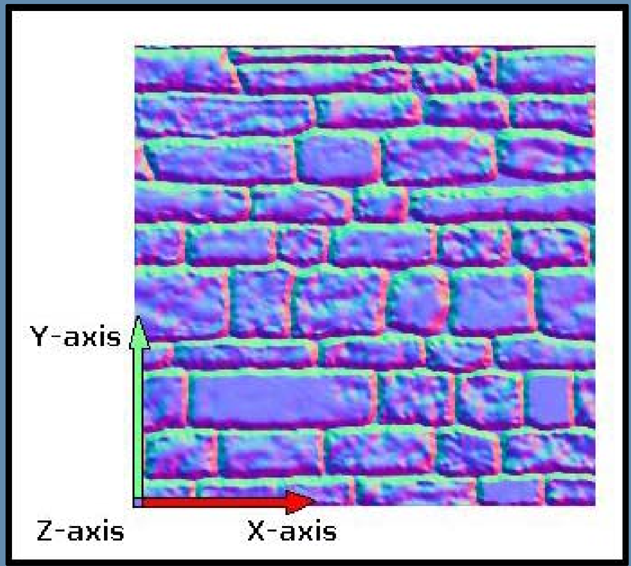
- Normal maps provide detail to geometry normals by *specifying* normals at each texel.
- Normals in a normal map replace normals from vertices.
- Information is 3D and needs 3 channels (more storage than bump maps).
- Can be stored in object-space or tangent-space.
- Direction values range in [-1,1] for each axis. Remapped to [0,1] for storage.

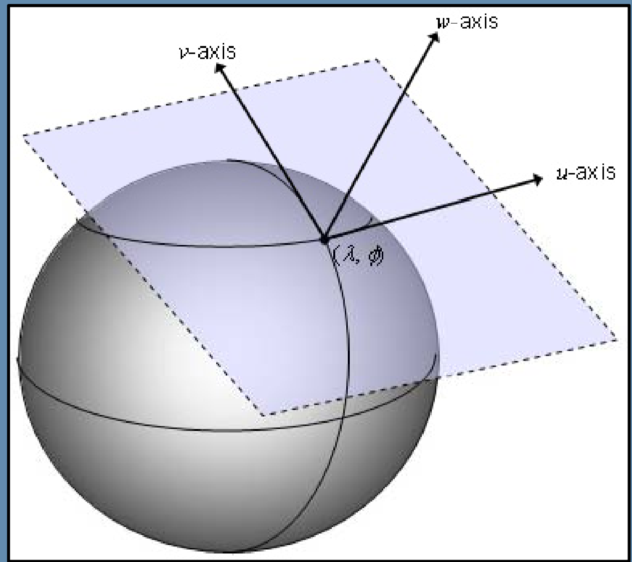
Texture Mapping: Per-pixel Lighting

- Normals in a normal map are commonly stored in tangent-space (the space of the surface the texture is mapped on).
- Must transform normals to same space as light: need a object-to-tangent space matrix (Tangent | Binormal | Normal matrix):

$$\begin{bmatrix} Tx & Ty & Tz \\ Bx & By & Bz \\ Nx & Ny & Nz \end{bmatrix}$$







Texture Mapping: Per-pixel Lighting (cont'd)

- Code example (transform light to tangent space):
- In the vertex shader:

```
// Calculate the light vector (vLightPosition is in object space)
vLightVector = vLightPosition - position.xyz;

// Transform the light vector from object space into tangent space
float3x3 TBNMatrix = float3x3(vTangent, vBinormal, vNormal);
vLightVector.xyz = mul(TBNMatrix, vLightVector);
```

- In the pixel shader:

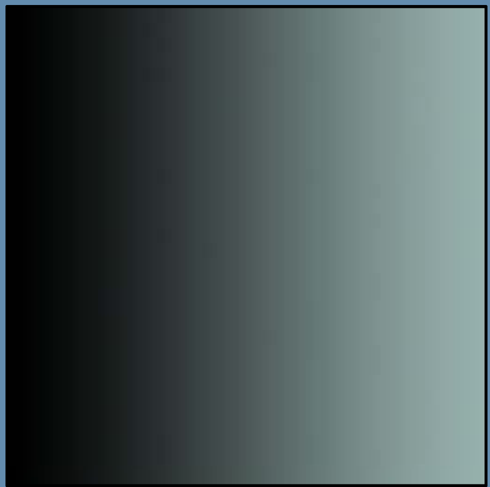
```
// Normalize the light vector after linear interpolation
vLightVector = normalize(vLightVector);

// Since the normals in the normal map are in
// the (color) range [0, 1], we need to uncompress them to "real“
// normal (vector) directions.
// Decompress vector ([0, 1] -> [-1, 1])
float3 vNormalColor = tex2D(normalTexture, normalCoords).rgb;
float3 vNormal = 2.0f * (vNormalColor - 0.5f);
```

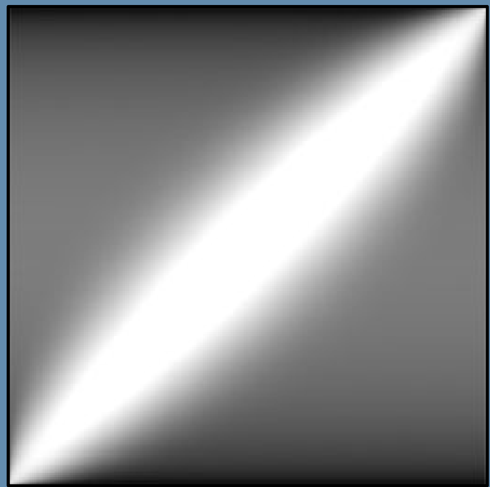
Texture Mapping: Masks/General Purpose (1)

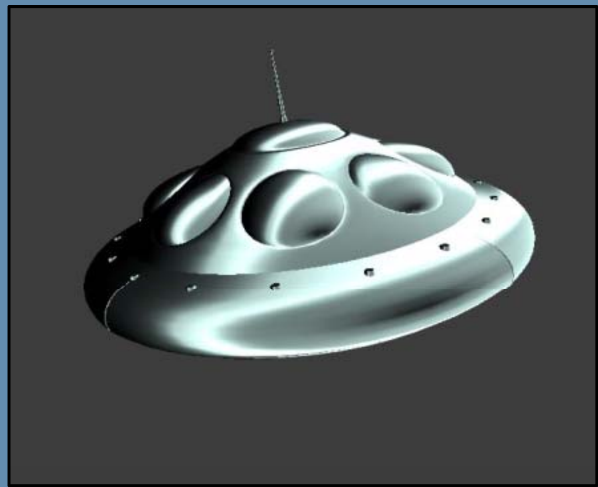
- Masking terms:
  - Translucency
  - Specular
  - Reflection
  - …etc

Texture Mapping: Masks/General Purpose (2)

- Look-up tables:
  - Pre-calculated computations or terms (e.g. `acos()`)



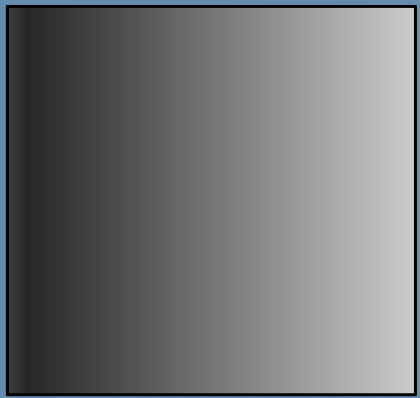
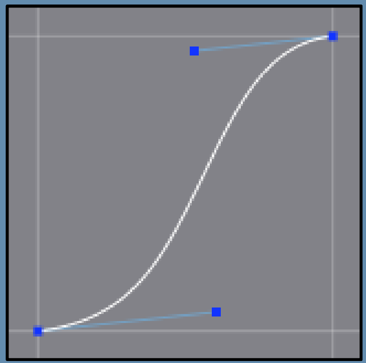
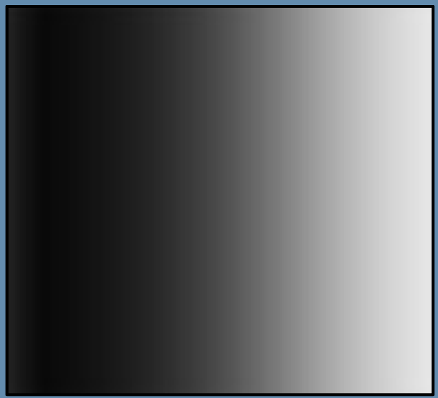




Texture Mapping: Masks/General Purpose (3)

- Color ramps, remapping, color correction:

```
finalColor.r = tex1D(texColorRemapR, texDiffuse.r);
finalColor.g = tex1D(texColorRemapG, texDiffuse.g);
finalColor.b = tex1D(texColorRemapB, texDiffuse.b);
```

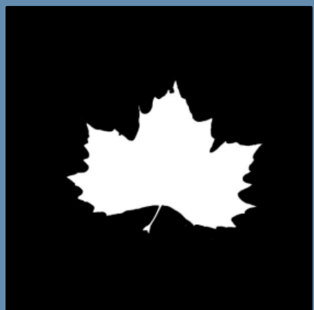





Fog

- Gradually fade colors to a background color:
  `finalColor = lerp(finalColor,fogColor,fogAmount)`
- Fog amount calculation determines fog effect and shape:
  - View-space depth
  - World-space height
  - Fog volumes
- Fog blend can be linear, exponential, or even a custom curve.
- In addition to the visual quality, it is a useful way to decrease rendering distance and hide popping artifacts.

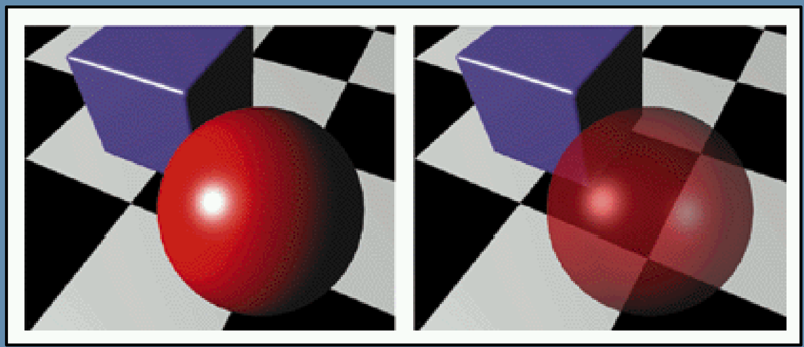
Transparency (alpha testing)

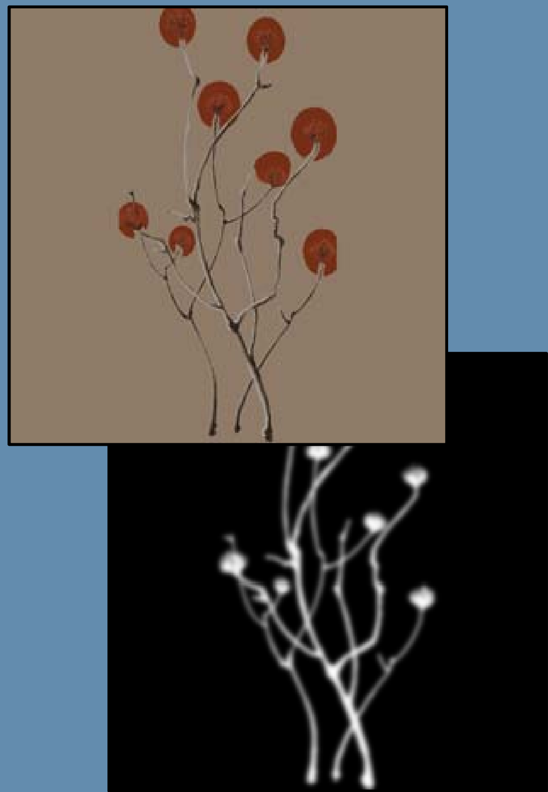
- Use alpha channel as a “cut-out mask”.
- Binary test is done on each pixel to be rendered (alpha testing):
  - Is your alpha value above a certain threshold?
    - Yes $\Rightarrow$ pixel continues rendering and goes to further stages in the pipeline.
    - No $\Rightarrow$ pixel is killed right away.
- Pixels that fail the alpha test *do not write* any values to the depth buffer.
- Do not confuse with alpha blending.


Translucency (alpha blending)

- Blend color with *background* by a specified amount
- Blending amount can be constant across the object
- Or read from a texture
- All pixels write to the depth buffer (even those with alpha=0)



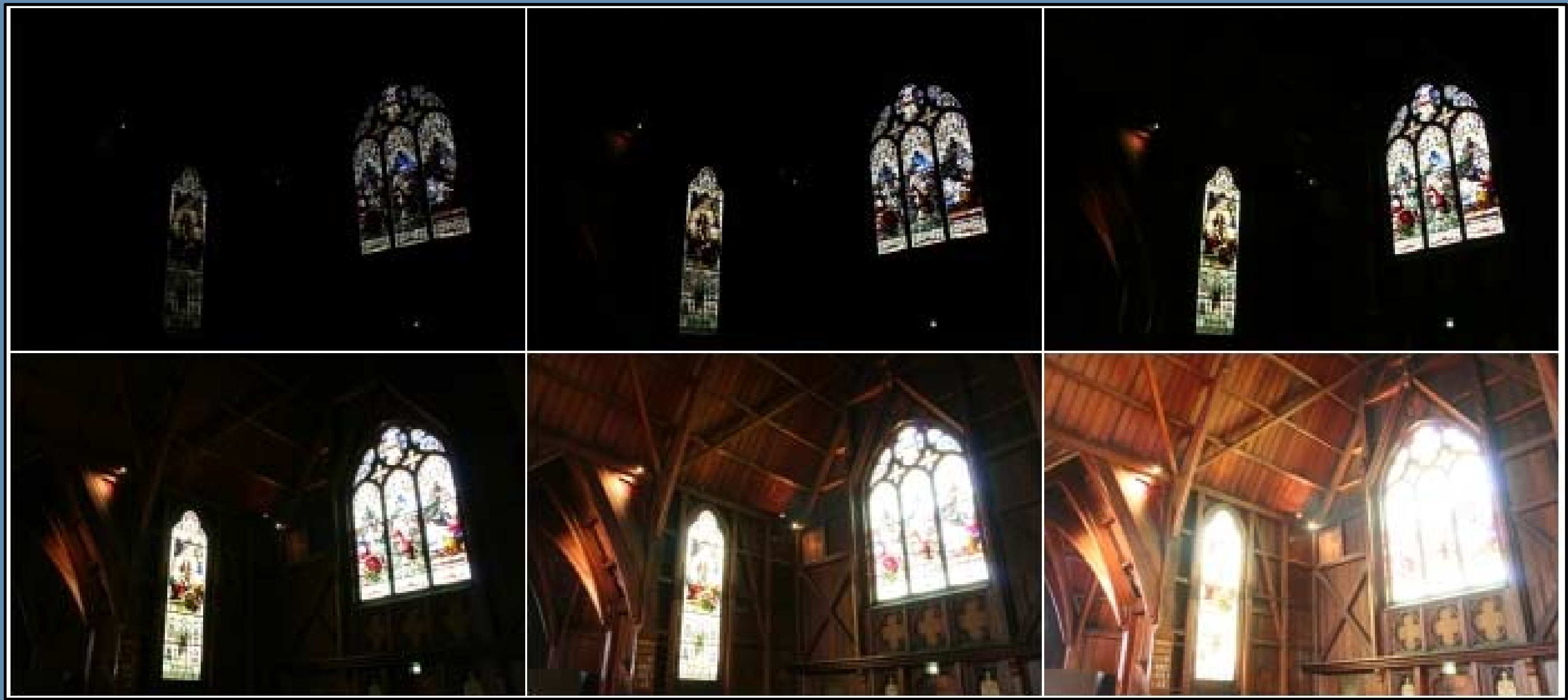


HDR Rendering

- Store/calculate colors outside of [0,255] [0,1] range.
- Express a wider range of color relationships (e.g., “very bright” objects).
- More correct lighting calculations (no saturation):
  - No more 1+1=1

HDR Rendering: Tone Mapping

- Current display devices are capable of only displaying [0,1].
- Tone mapping brings HDR images back to [0,1] for display on LDR devices.
- A number of mapping approaches exist.
  - Simple example: take minimum and maximum color values in the screen, and map them to [0,1] respectively, with all colors in-between linearly mapped within [0,1].



Global Effects

- Shadows
- Light maps
- Radiosity
- Ambient Occlusion
- Reflections and Environment Mapping

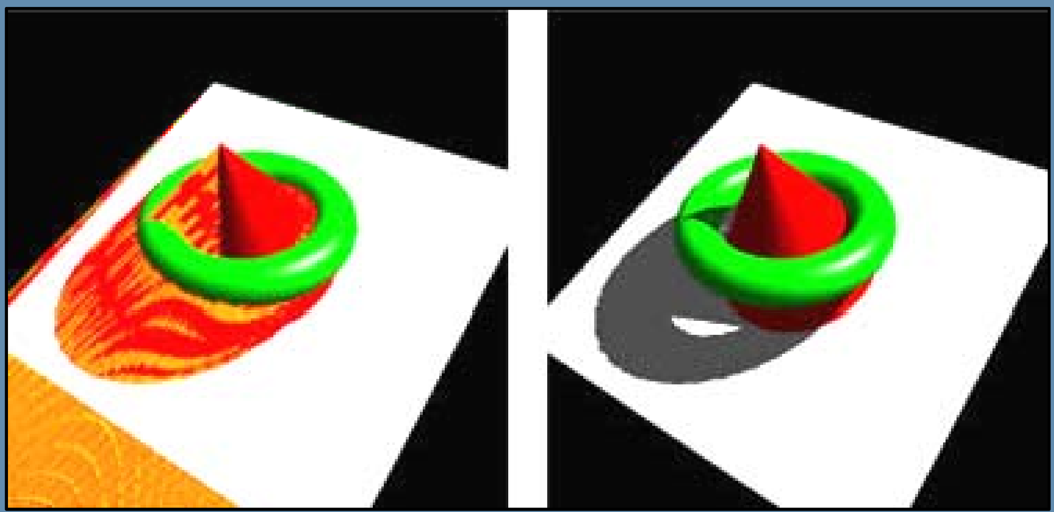
Shadows

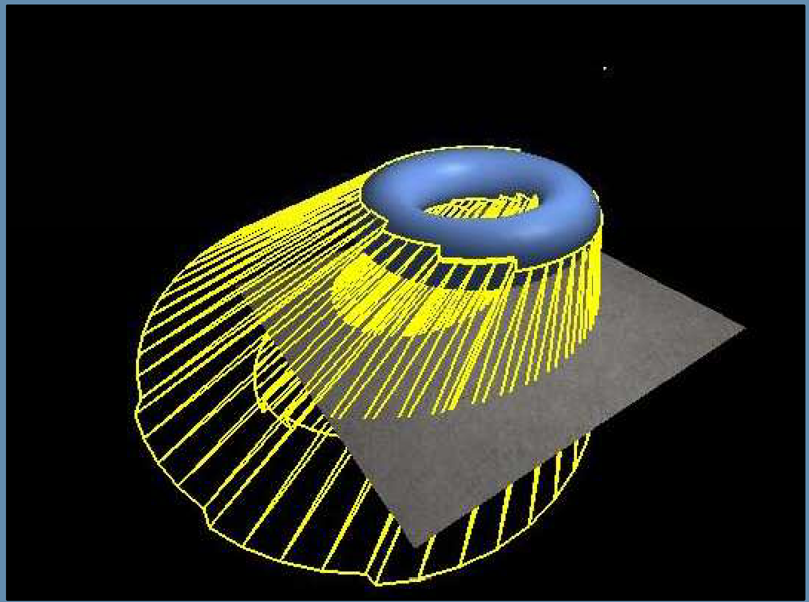
- Important to “stage” objects in the scene.
- Dynamically calculated: shadow volumes, shadow maps, …etc.
- Statically baked: light maps.
- If an object is shadowed from one light, then it does not “see” it.
- A shadowed scene has:
  - Light
  - Shadow caster
  - Shadow receiver

Shadows : Shadow Volumes

- For every light:
  - For every object:
    - Extend volume from object boundaries and light position.
  - Draw entire scene, and check if screen pixel falls inside a volume.
    - Yes $\Rightarrow$ Avoid accumulating light contribution.
    - No $\Rightarrow$ Accumulate light contribution.

- Dependent on shape complexity.
- Consumes a lot of fill rate.





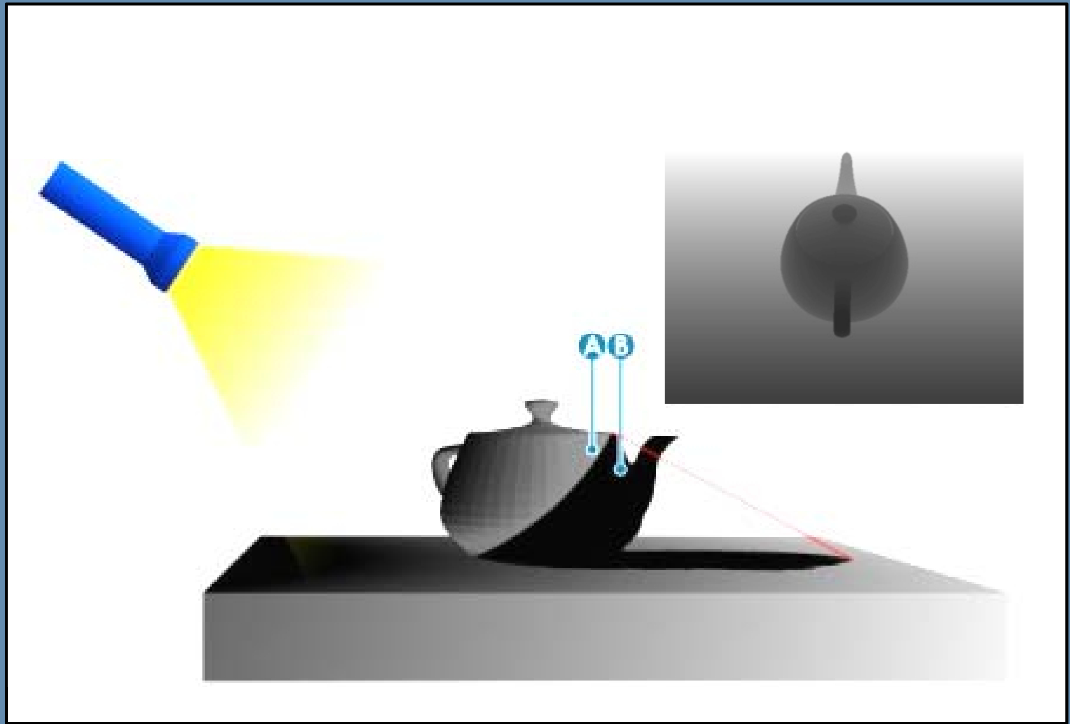
Shadows : Shadow Maps

- To a separate "shadow" depth buffer, draw all objects from the light's point-of-view.

  ⇒ Stores what is visible from the light's point of view.

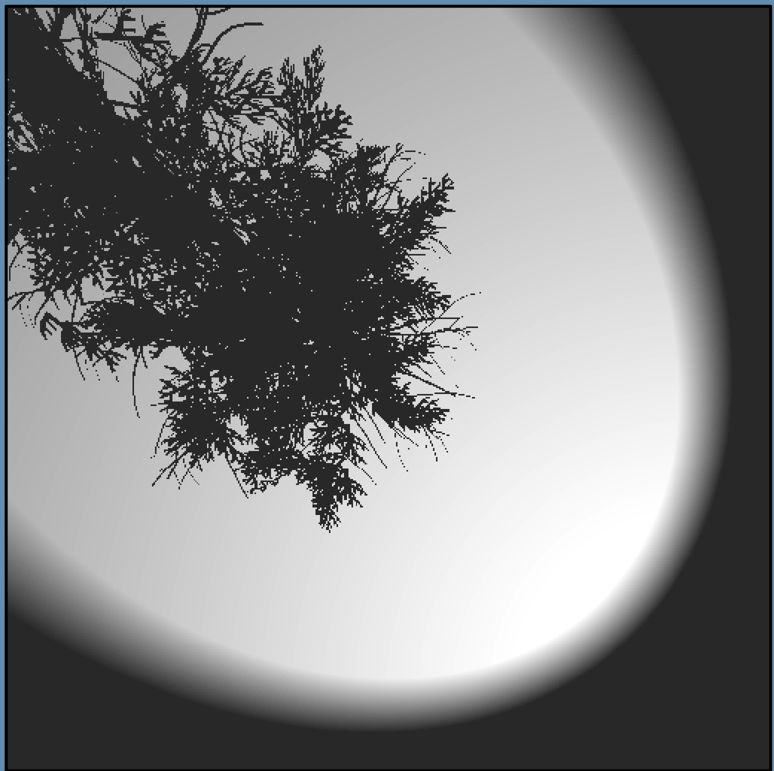
- Draw objects to screen normally. For every pixel, object asks the shadow map: do you see this pixel of me?
  - Yes ⇒ Pixel is lit by that light.
  - No ⇒ Pixel is shadowed from that light.

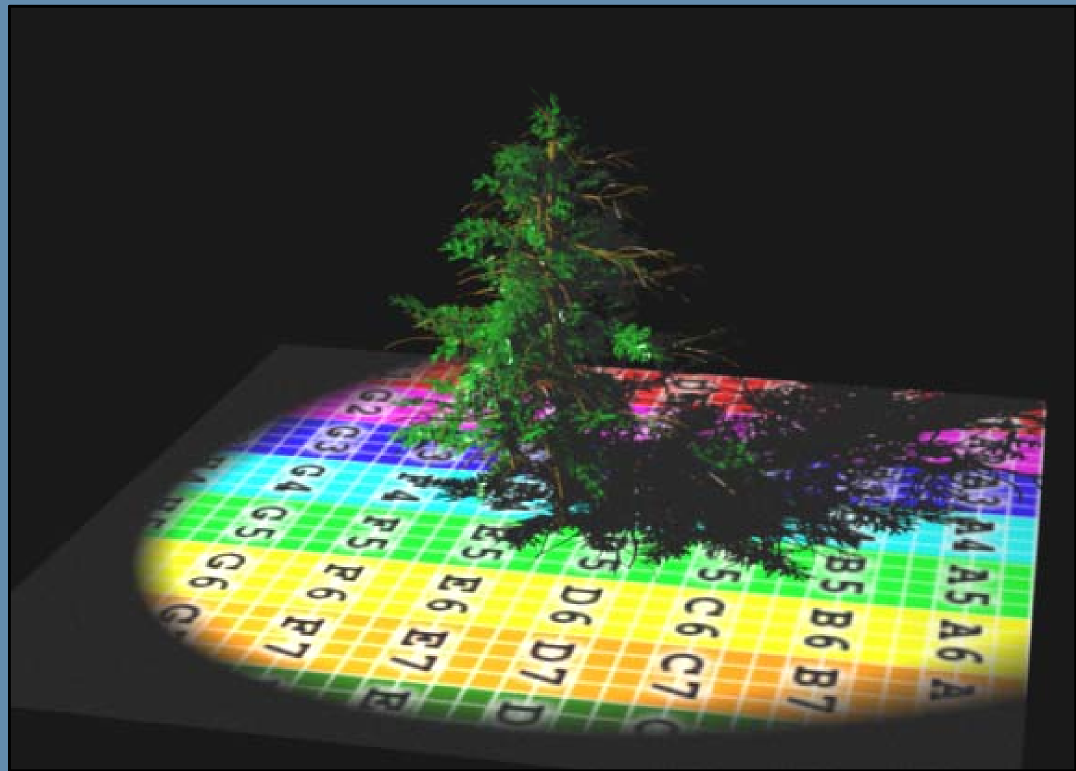
- Irrelevant of geometrical complexity.



Shadows : Light Maps

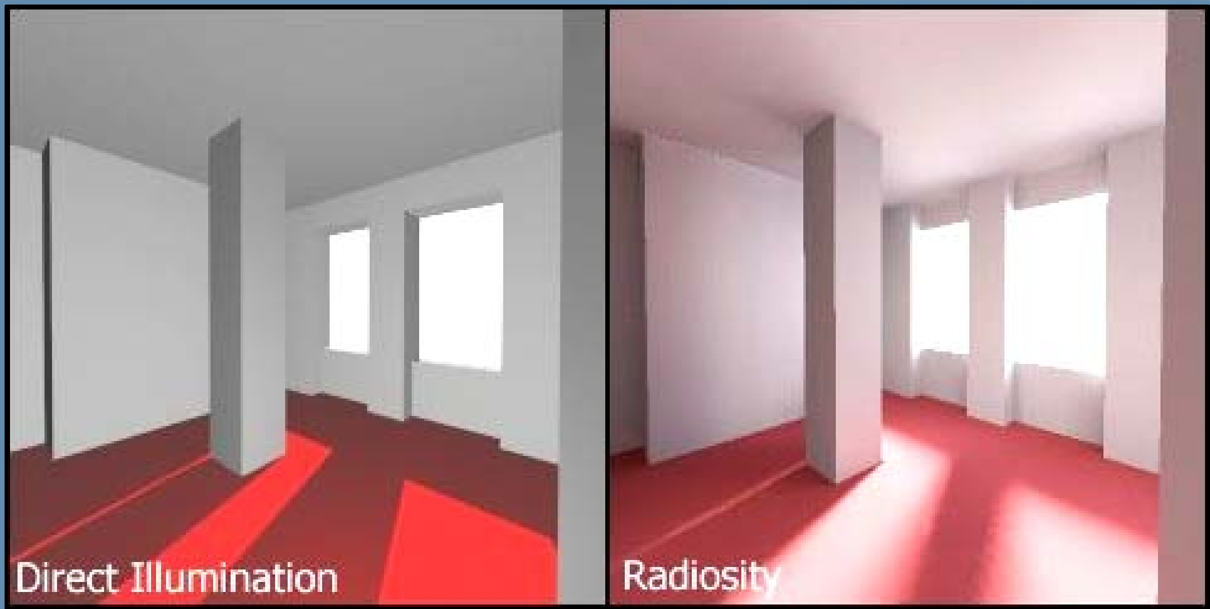
- Calculate lighting beforehand, and store it for run-time use.
- Applicable to static scenes (static lights + static geometry).
- Can consume large amounts of memory.
- Usually compressed into an atlas based on detail resolution.





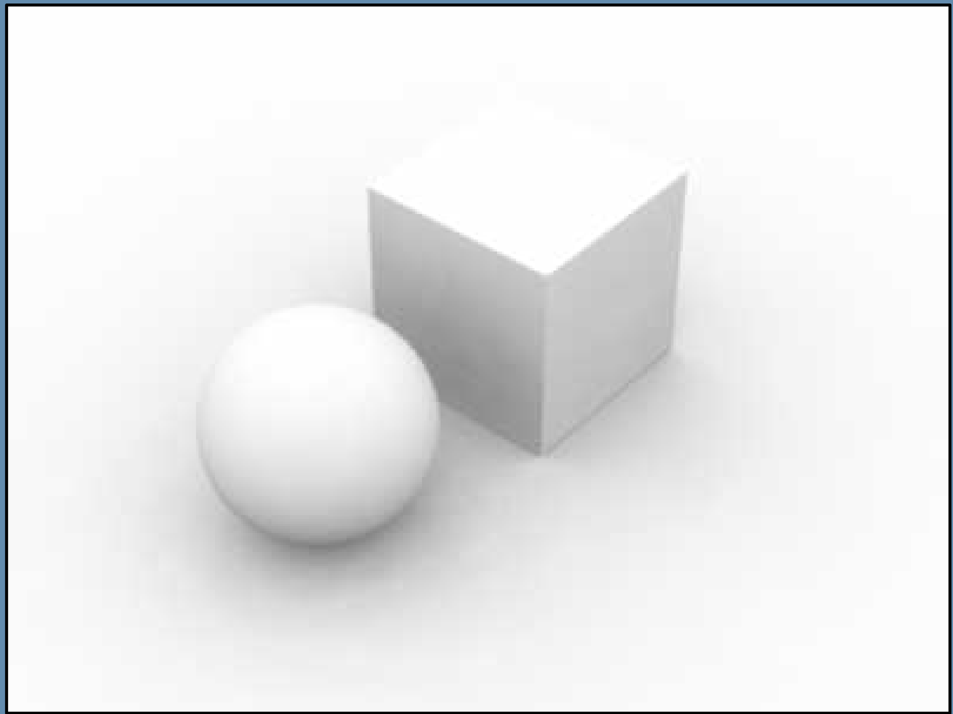
Radiosity Lighting

- Tracing diffuse reflectance between scene objects.
- Can be faked in real-time by adding colored lights sampling the surrounding environment.
- Irradiance via radiosity.



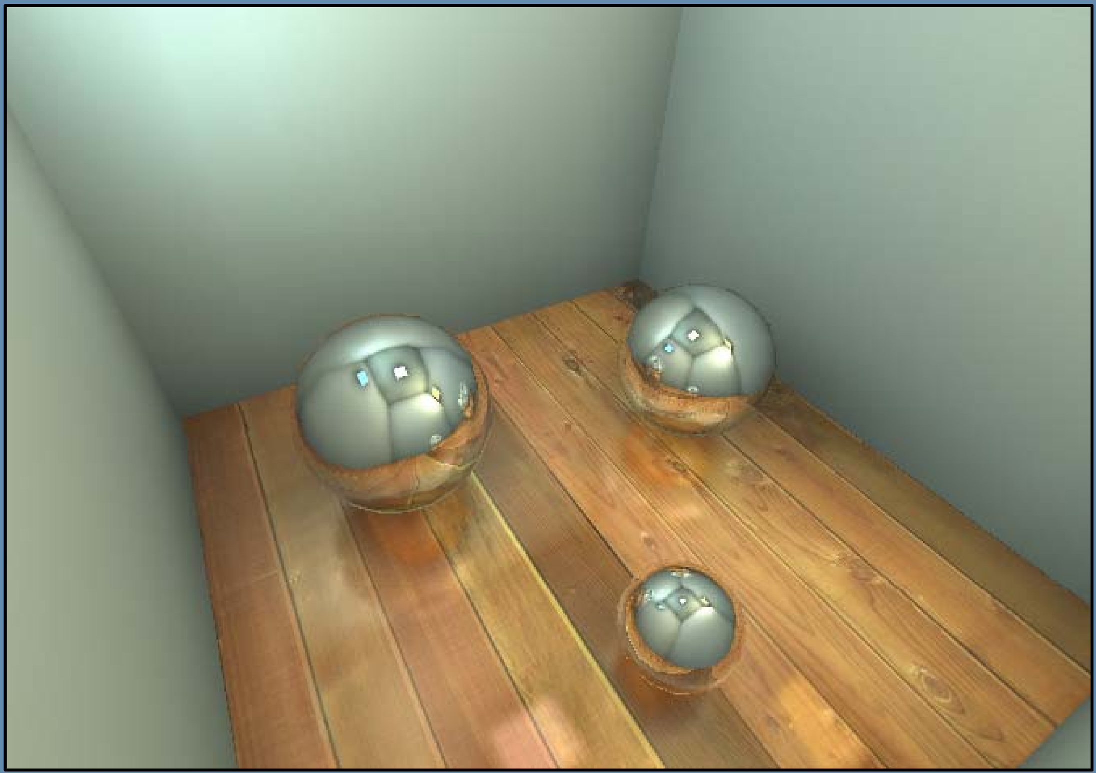
Ambient Occlusion

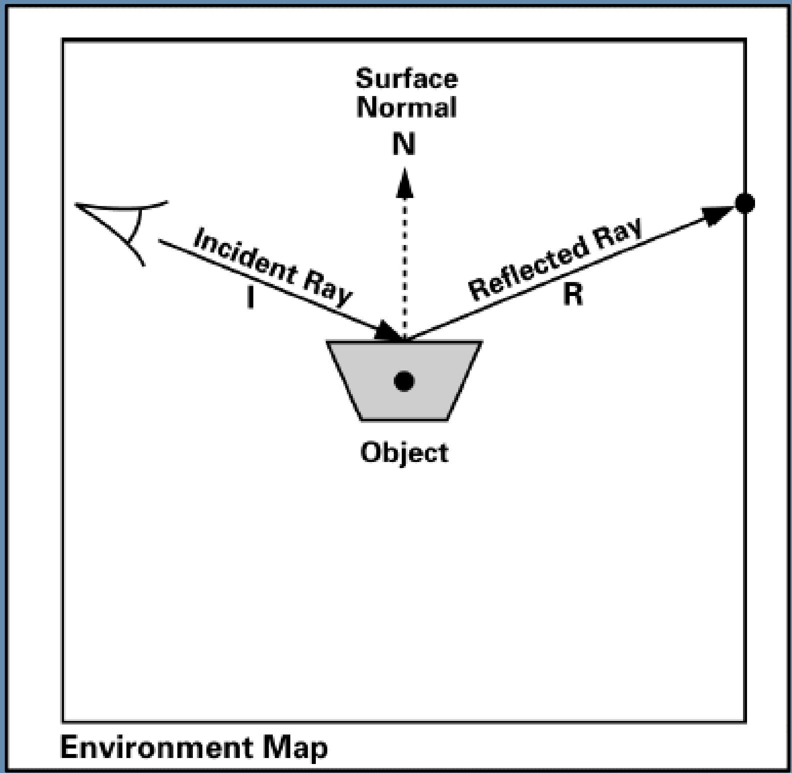
- The ambient term in the common lighting formula was found to be a little bit too simplified.
- A single point can receive light reflected from many surfaces.
- Areas obstructed by other surfaces are less likely to receive bounced light rays.
- Modulate ambient term by how much indirect lighting a point can receive $\Rightarrow$ area visibility test.



Environmental Mapping: Reflections

- Reflective materials act as mirrors to their surrounding environment.
- Naturally achievable with a ray-tracer.
- Polygon projection renderers must do some tricks to achieve it.
  - Environment cube maps
  - Spherical environment mapping

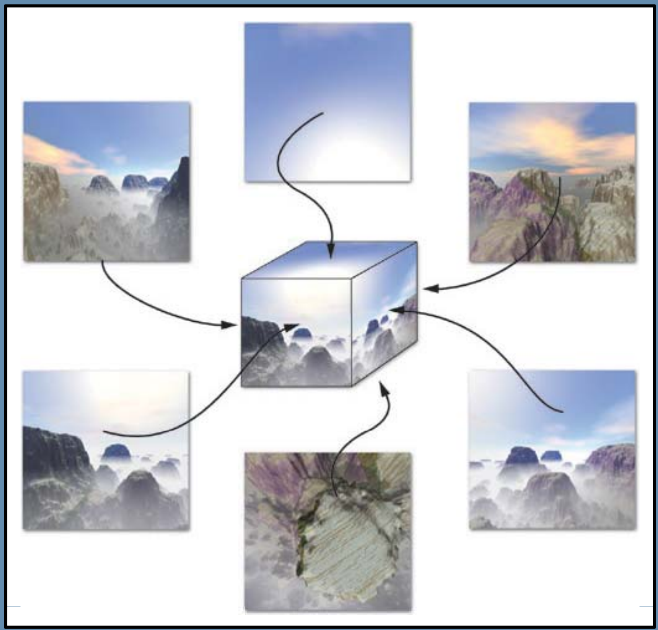


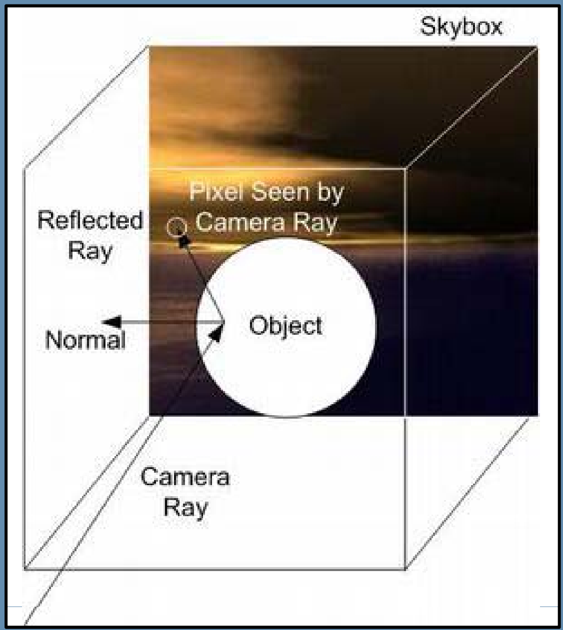


Environmental Mapping: Cube Maps

- 6 images sampling a cube surrounding point of interest.
- Dynamic updates are relatively cheap and feasible:
  - Render scene to the six sides of the cube map

$$\begin{bmatrix} U \\ V \\ W \end{bmatrix} = \begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix}$$





Environmental Mapping: Spherical Mapping

- Single image sampling a sphere surrounding point of interest.
- Good for static reflections.
- Dynamic generation requires highly tessellated geometry to support curved lines.

right = M^{LocalToView}[0]

up = M^{LocalToView}[1]

$$\begin{bmatrix} U \\ V \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{right_x}{2} & \frac{right_y}{2} & \frac{right_z}{2} & 0.5 \\ \frac{up_x}{2} & \frac{up_y}{2} & \frac{up_z}{2} & -0.5 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} r_x \\ r_y \\ r_z \\ 1 \end{bmatrix}$$

Image Space

- Post Processing
- Image Filtering
- Image Space Effects
- Deferred Shading

Post-Processing

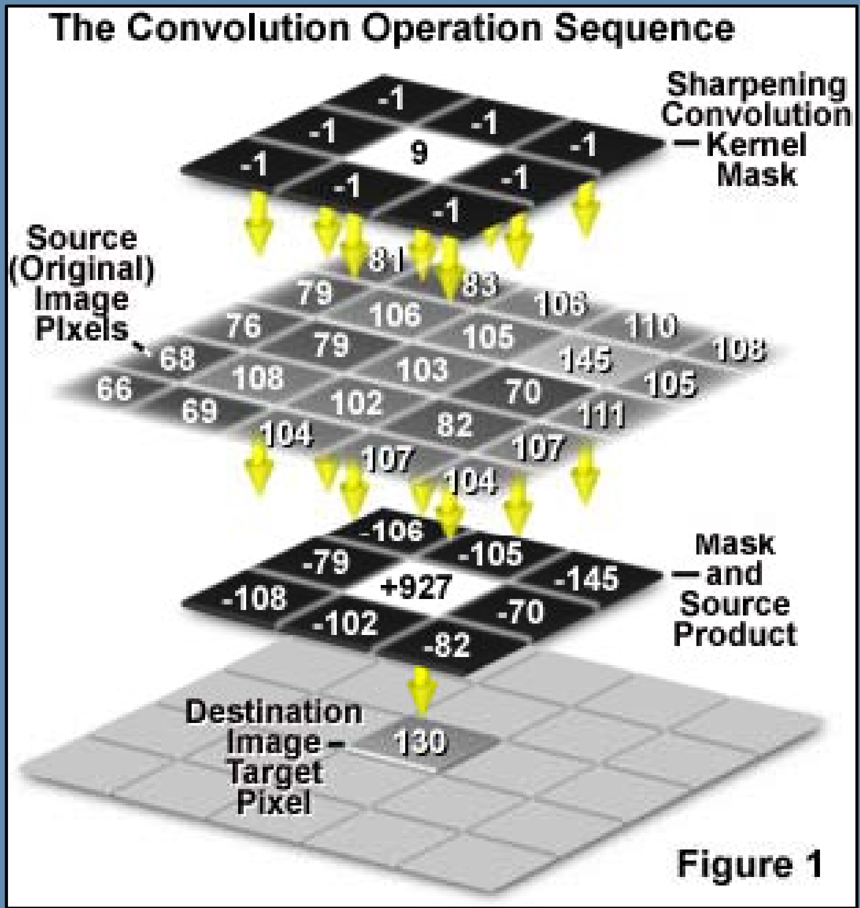
- Apply additional passes of processing over pixels that have been already rendered before.
  - Purely image-based processing.

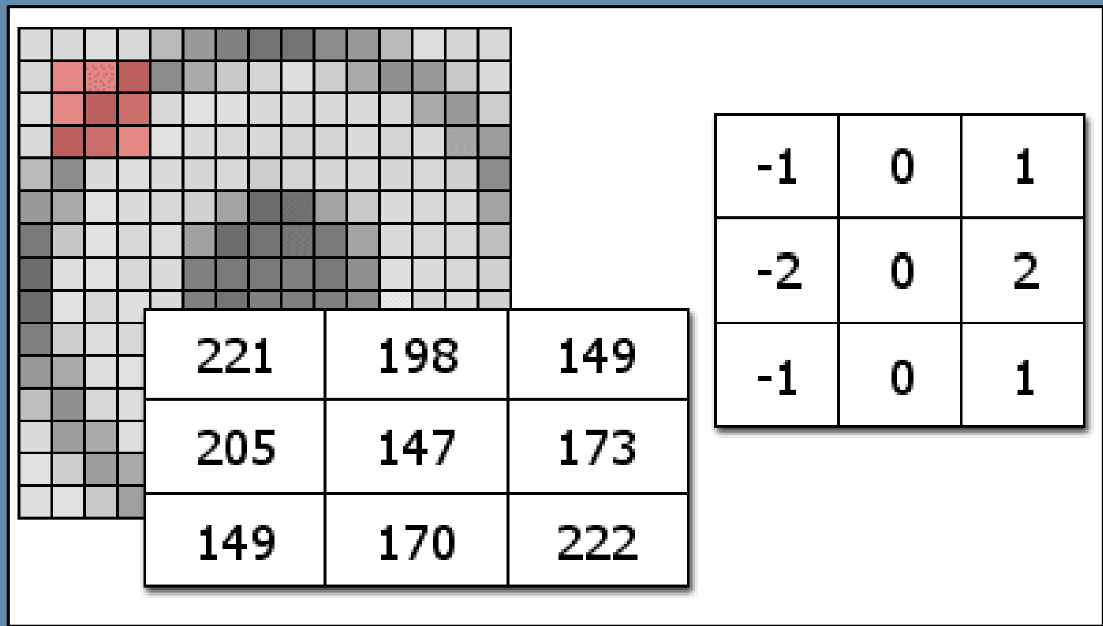
```
for (i=0; i<NumPixels; i++)
{
    Pixel px = SourceImage.Pixels[i];
    CurrentRenderTarget.Pixels[i].rgb =
              (px.Color.r + px.Color.g + px.Color.b) / 3;
}
```

- Output result is stored in a new buffer.

Post-Processing : Image Filtering

- Application of image space convolution (spatial domain).
- Each pixel in the source image is passed through a “kernel”.
- Kernel can sample surrounding pixels within a certain “radius”.





Filters : Sharpness

$$\begin{pmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

Filters : Emboss

$$\begin{pmatrix} 2 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

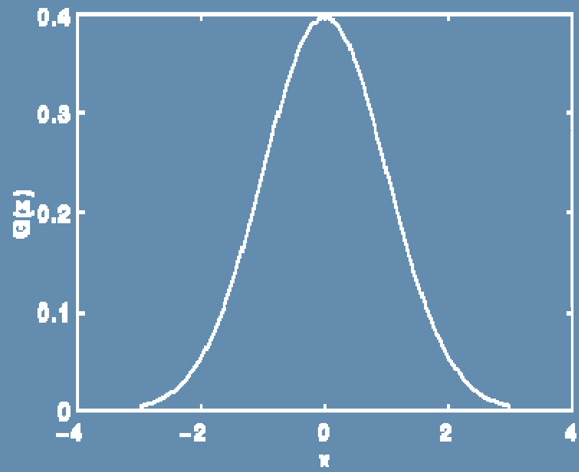
Filters : Blur

- Reduces noise and detail.
- Used in many effects:
  - Depth of field, out of focus
  - Bloom
  - Fighting hard edges (anti-aliasing)
- Each pixel is averaged with its surroundings to a certain distance.
- Kernel size determines amount of blurriness.
- Apply it on a down-sampled image to achieve even bigger kernel sizes.

1	2	1
2	4	2
1	2	1

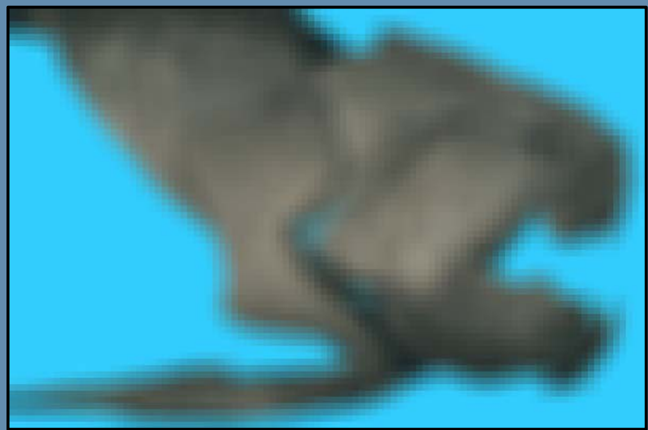
Filters : Box vs. Gaussian Blur

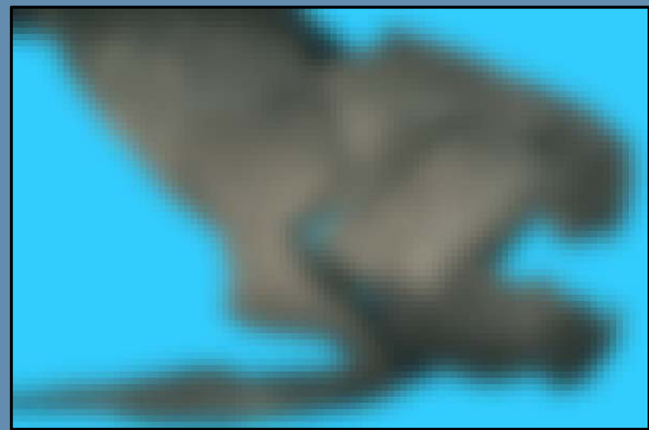
- Kernel samples concentrate on center.
- Can be separated to two passes.



$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

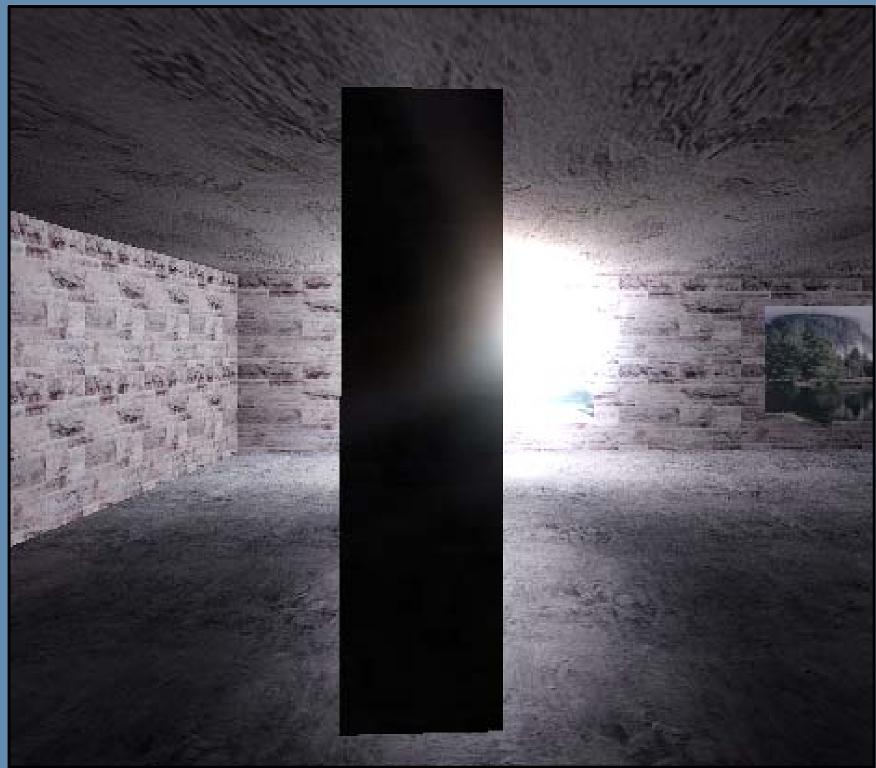
$$\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \times \begin{pmatrix} 1 & 2 & 1 \end{pmatrix}$$





Other Effects : Bloom

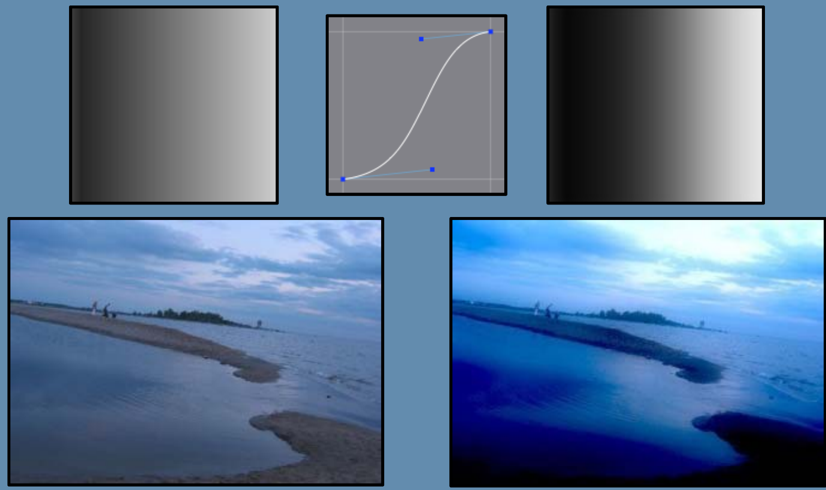
- Resembles camera over-exposure.
- Blur only very bright areas.
- Add blurred image over original.
- Different blur kernels can be used to simulate different effects.



Other Effects : Color Remapping

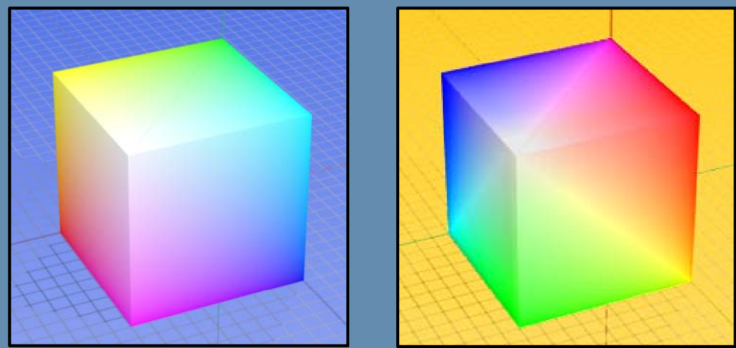
- Remap colors using 3 1D textures:

```
pix.r = tex1D(texColorRemapR, pix.r);
pix.g = tex1D(texColorRemapG, pix.g);
pix.b = tex1D(texColorRemapB, pix.b);
```



- Remap colors using 1 3D texture (volume):

```
pix.rgb = tex3D(texColorRemap, pix.rgb);
```



Other Effects : Screen-space Ambient Occlusion

- Calculates the concavity of a point on the surface at each pixel.
- Usually via:
  - Neighbor normal angles.
  - Neighbor depth differences.
- Point is concave => Darker.

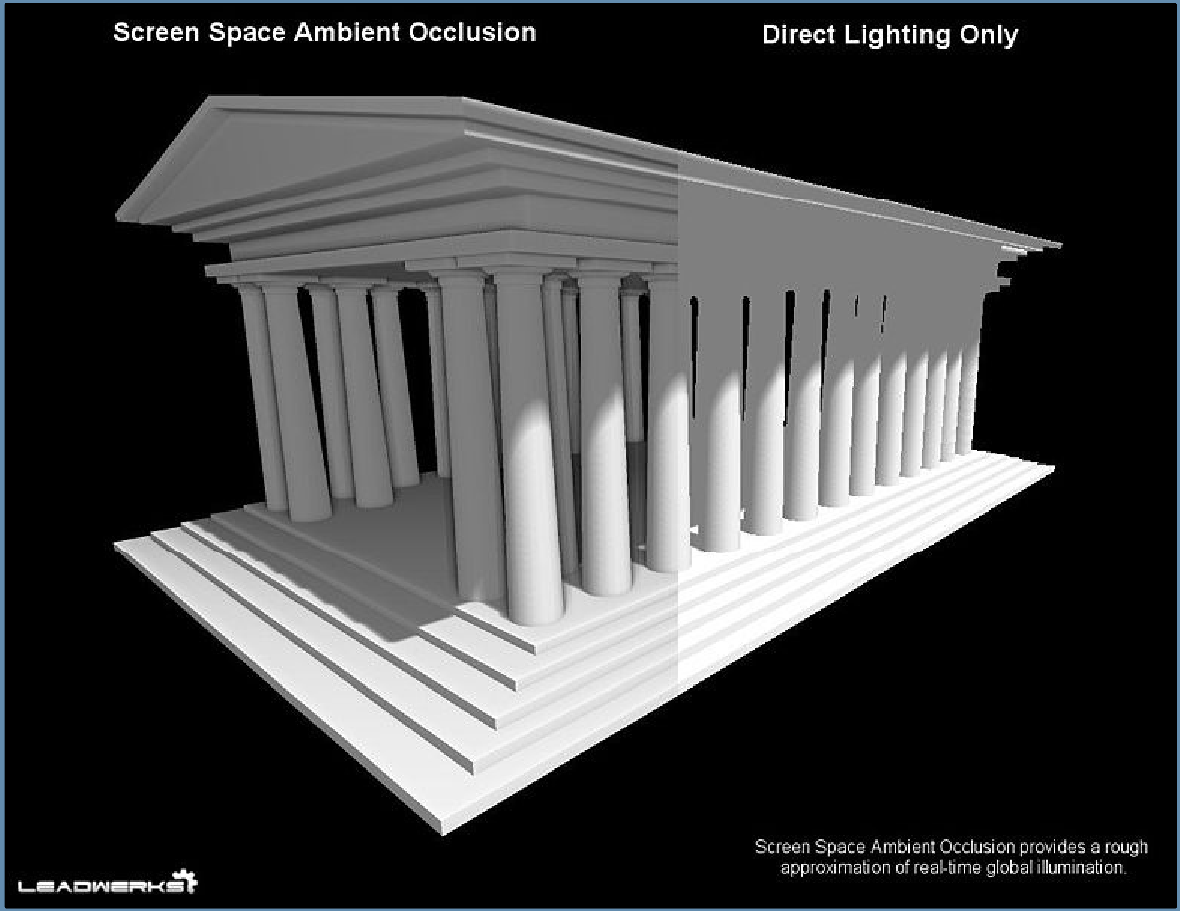
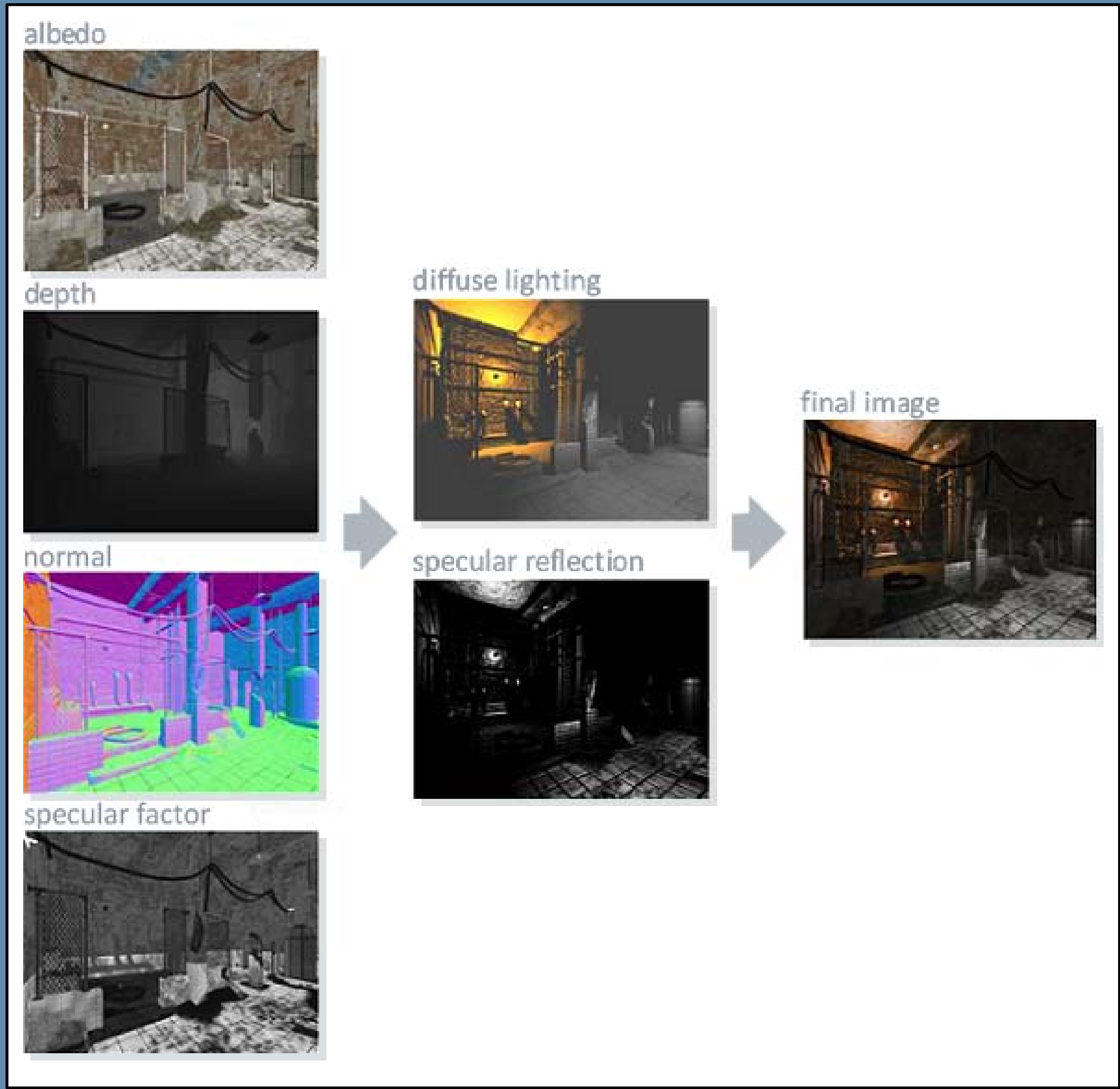


Image Space Lighting : Deferred Shading

- Rasterize render data in intermediary image buffers:
  - Diffuse color
  - Depth
  - Normals
  - …etc
- Apply lighting passes in screen space
  - Render light volumes
  - Apply lighting in screen space



Questions?

Credits

- Sergei Savchenko
- Jean-Sebastien Perrier
- Homam Bahnassi

Thank You!