Creating Photorealistic and Non-Photorealistic Effects for Games

Bruno P. Evangelista (UFMG)
Alessandro R. Silva (UFMG)

Revision 2

Introduction

- The fast evolution of the programmable graphics hardware (GPUs) are making the games very realistic!
  - Is it still possible to distinguish between a game scene and a real-life picture?

Introduction

Introduction

- The modern GPUs enable us to create many rendering effects
  - We can use these effects to create very realistic environments
  - Or even non-realistic environments
- In this lecture we will discuss and show some effects that are commonly used in games

Agenda

- **Rendering Pipeline and Shaders (Quick Review)**
- **Shader Languages**
- **Effects**
  - **Per-Pixel Illumination**
  - **Environment Reflection/Refraction**
  - **Texturing/Multi-texturing**
  - **Procedural Texture Generation**
  - **Simulation of Detailed Surfaces**
- **Pos-Processing Effects**
  - **Radiometry**
  - **Bloom**
  - **Cartoon Rendering**

Rendering Pipeline

- For many years, graphics APIs such as DirectX and OpenGL used a fixed rendering pipeline
  - The processes executed within each stage of the rendering pipeline were pre-programmed in hardware and cannot be modified
    - For example: Transformations, lighting and so on...
  - It was only possible to configure a few parameters on the pipeline
- Result: Games with resembling graphics!

Rendering Pipeline

- Meantime...
  - The cinema industry already had tools capable of programming the rendering of the scenes
- RenderMan
  - Shader language specification created by Pixar in 1988
  - Nowadays there are some open-source and commercial implementations
- However, those tools were used just for offline rendering! =(

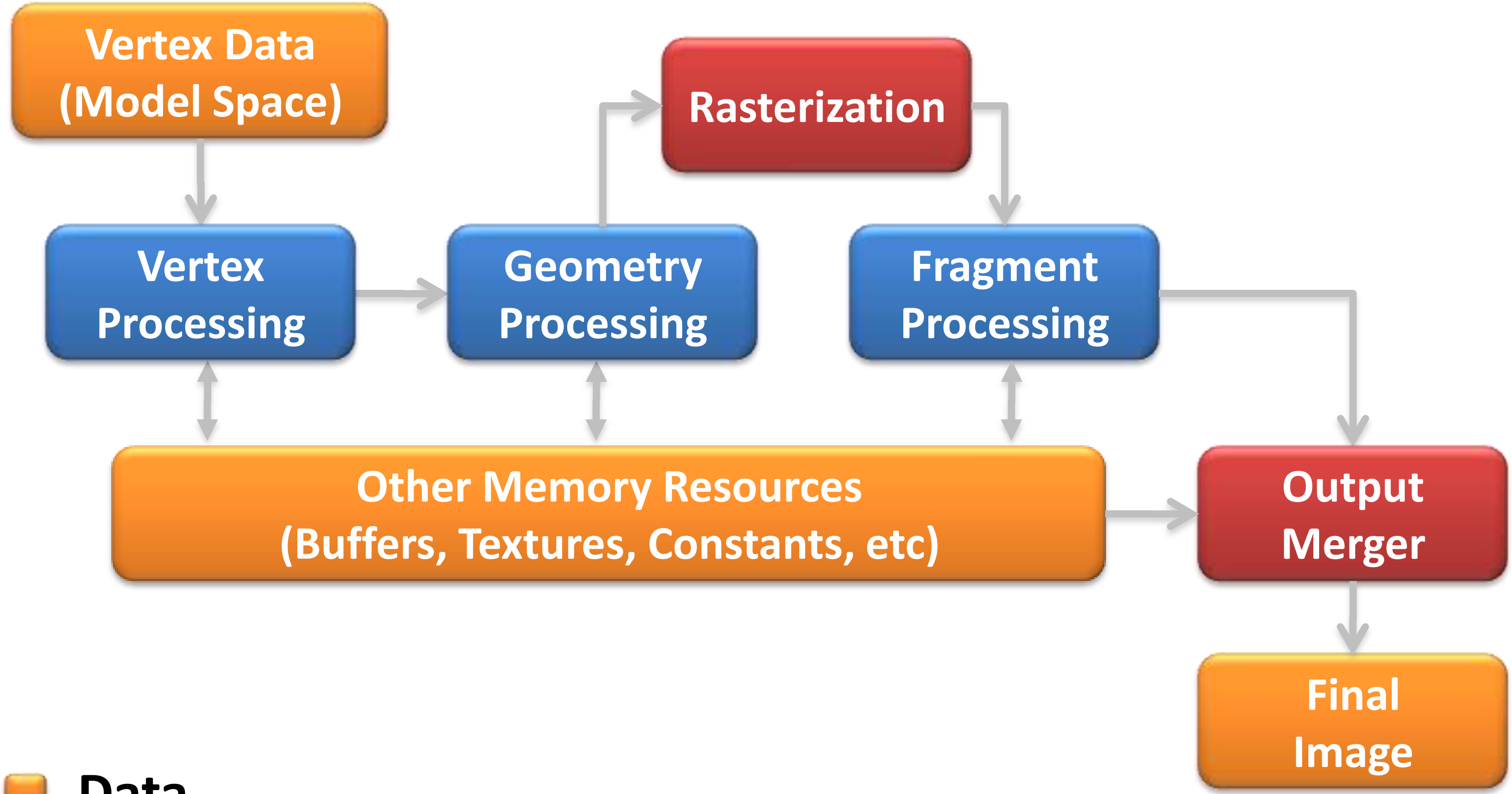
Shaders

- Small programs that run into the GPU
- Allow the programming of some stages of the rendering pipeline
- New things that you can do:
  - Real world illumination models
  - Rendering of very detailed surfaces
  - Pos-processing over the scenes
- And we can use everyting in real-time!

Shaders

- Shaders that run in different stages of the GPU have different names
  - Vertex Shader – Vertex Processing Stage
  - Pixel Shader – Pixel Processing Stage
  - Geometry Shader – Geometry Processing Stage

Rendering Pipeline



- 🟧 **Data**
- 🟦 **Programmable stage**
- 🟥 **Non-Programmable stage**

What you can do with shaders?

Videos and Examples

Shader Languages

- Offline Rendering
  - RenderMan – PRMan Pixar/Other implementations
  - Gelato – nVidia
- Real-time Rendering
  - HLSL (High Level Shading Language) – Microsoft
    Used on DirectX e XNA
  - GLSL (OpenGL Shading Language) – 3D Labs
    Used on OpenGL
  - Cg (C for Graphics) – nVidia
    Can be used on both DirectX and OpenGL

HLSL

- Has a small set of instrinsic functions
  - Arithmetic operations, texture access and flow control
- Has some C/C++ data types, besides vectors and matrices
  - bool, int, half, float, double
  - vectors (floatN, boolN, ...), matrices (floatNxM, ...)
  - texture, sampler, struct
- A shader code looks like a matematical equation

Per-Pixel Illumination

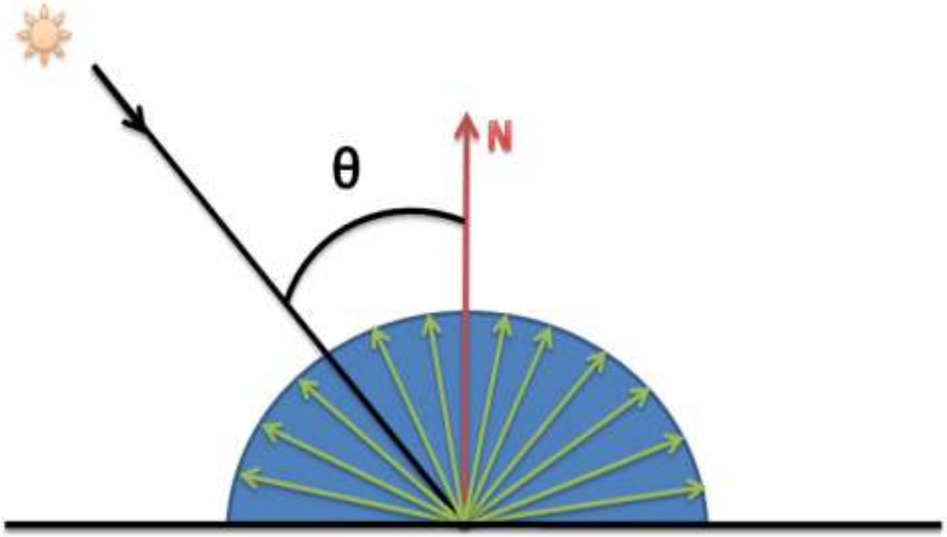
- The Blinn-Phong algorithm is commonly used in the graphics APIs for lighting
  - Empirical model
  - Light is represented by three separated components: ambient, diffuse and specular
- Light componentes
  - Ambient: Light equally scattered in the scene
  - Diffuse: Light that interacts with the surfaces
  - Specular: Light that is perfectly reflected by the surface

$$I_{total} = I_{ambient} + \sum_{LIGHTS} I_{diffuse} + I_{specular}$$

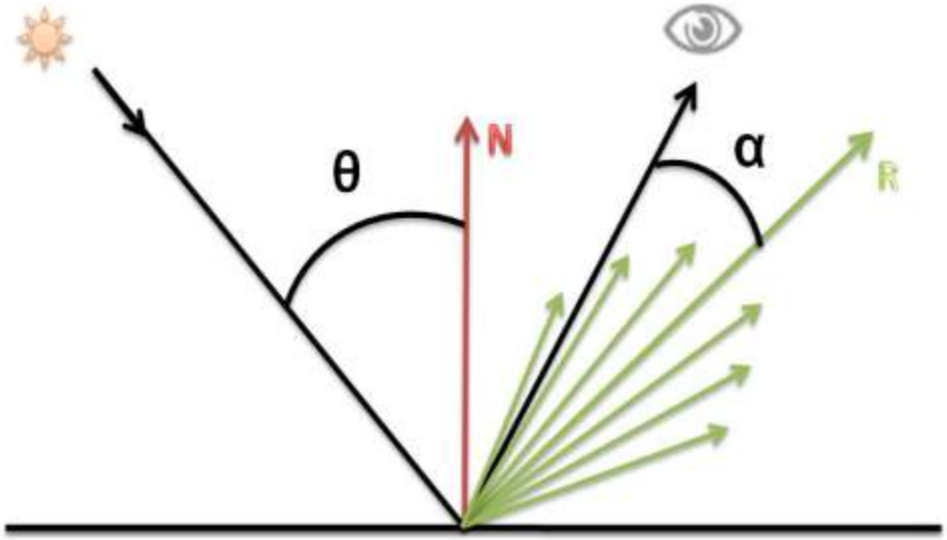
Diffuse and Specular

- Diffuse: Light that is equally reflected in all directions (isotropic)
- Intensity can be calculated according to Lambert's law

- Specular: Light that is reflected preferred in one direction
- Preferred reflection direction calculated according to Snell's law



$$I_{diffuse} = K_d L_d (N.L)$$



$$I_{specular} = K_s I_s (R.V)^{shininess}$$

Implementing as a Pixel Shader

$$I_{total} = I_{ambient} + \sum_{LIGHTS} I_{diffuse} + I_{specular}$$

$$I_{diffuse} = K_d L_d (N.L)$$

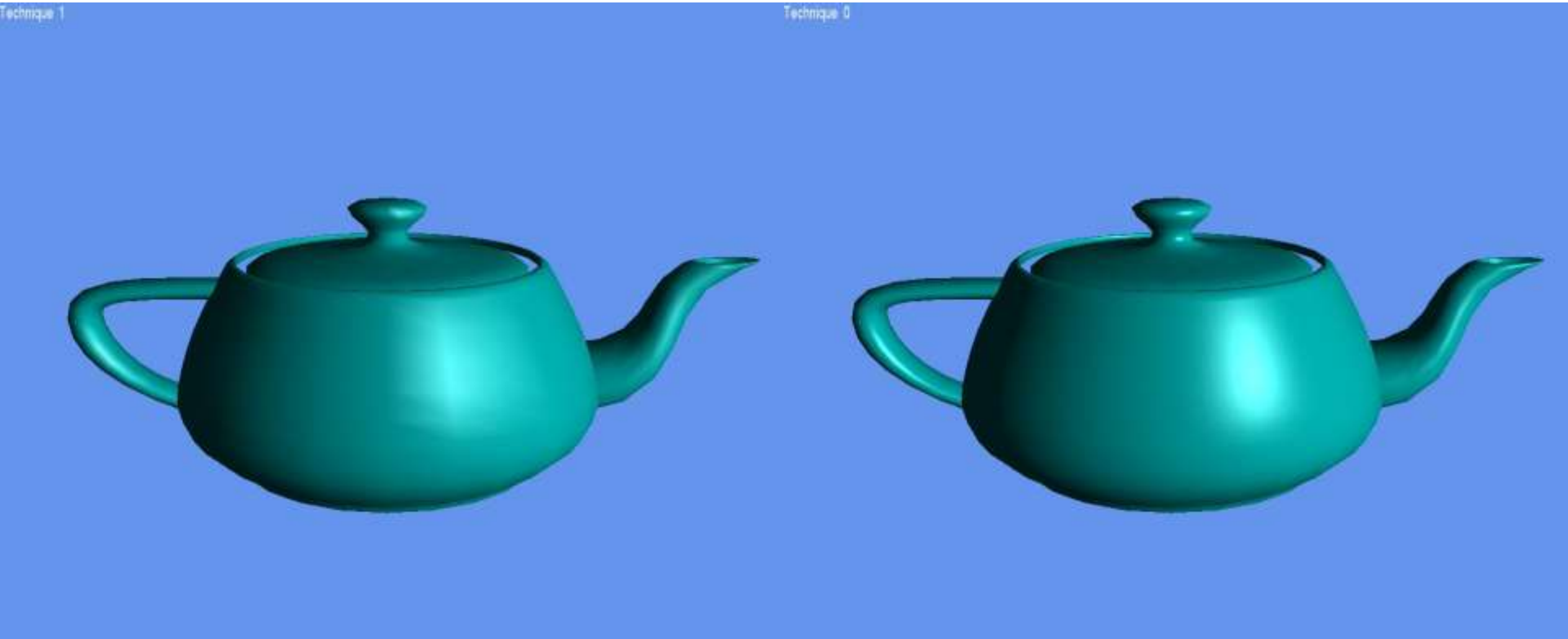
$$I_{specular} = K_s I_s (R.V)^{shininess}$$

```
void phongLighting(in float3 normal, in float3 lightVec,
  in float3 eyeVec, in float3 lightColor,
  out float3 diffuseColor, out float3 specularColor)
{
  float diffuseInt = saturate(dot(normal, lightVec));
  diffuseColor = diffuseInt * materialKd * lightColor;

  float3 reflectVec = reflect(-lightVec, normal);
  float specularInt = saturate(dot(reflectVec, eyeVec));
  specularInt = pow(specularInt, materialShininess);
  specularColor = specularInt * materialKs * lightColor;
}
```

Per-Pixel Illumination

- **Demo – XNA**



Attenuation

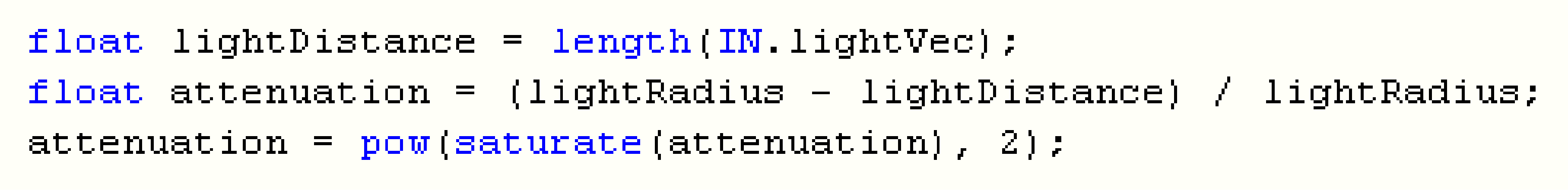
- We can define a range for point and spot lights and use it to attenuate the light intensity
  - The attenuation determines how fast the light intensity decreases
  - Attenuation can be constant, linear, quadratic, etc…
  - Using attenuation the reflect light appears more smooth over the surface

Implementing as a Pixel Shader

Quadratic Attenuation Function

$$L_{intensity} = Max\left(\frac{L_{range} - \ distance}{L_{range}}, 0\right)^2$$

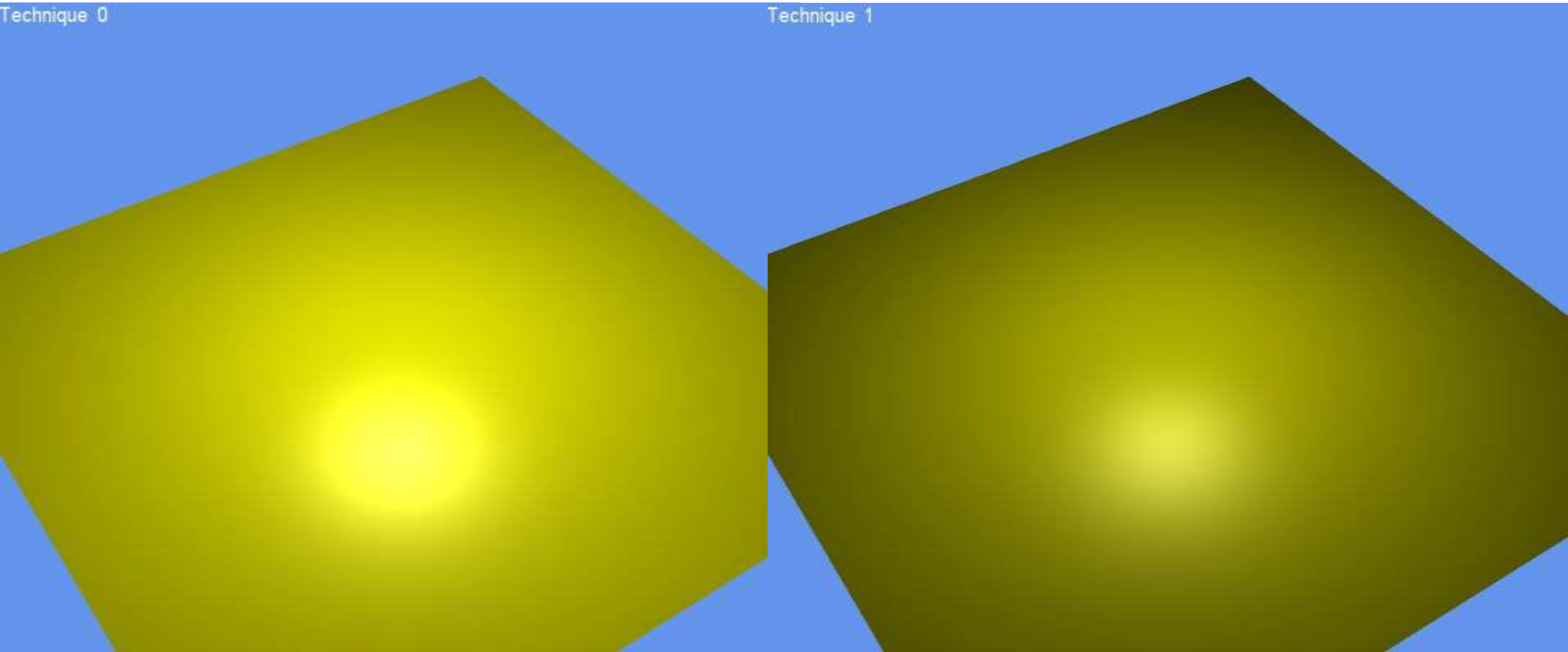
Shader Code



```
float lightDistance = length(IN.lightVec);
float attenuation = (lightRadius - lightDistance) / lightRadius;
attenuation = pow(saturate(attenuation), 2);
```

Attenuation

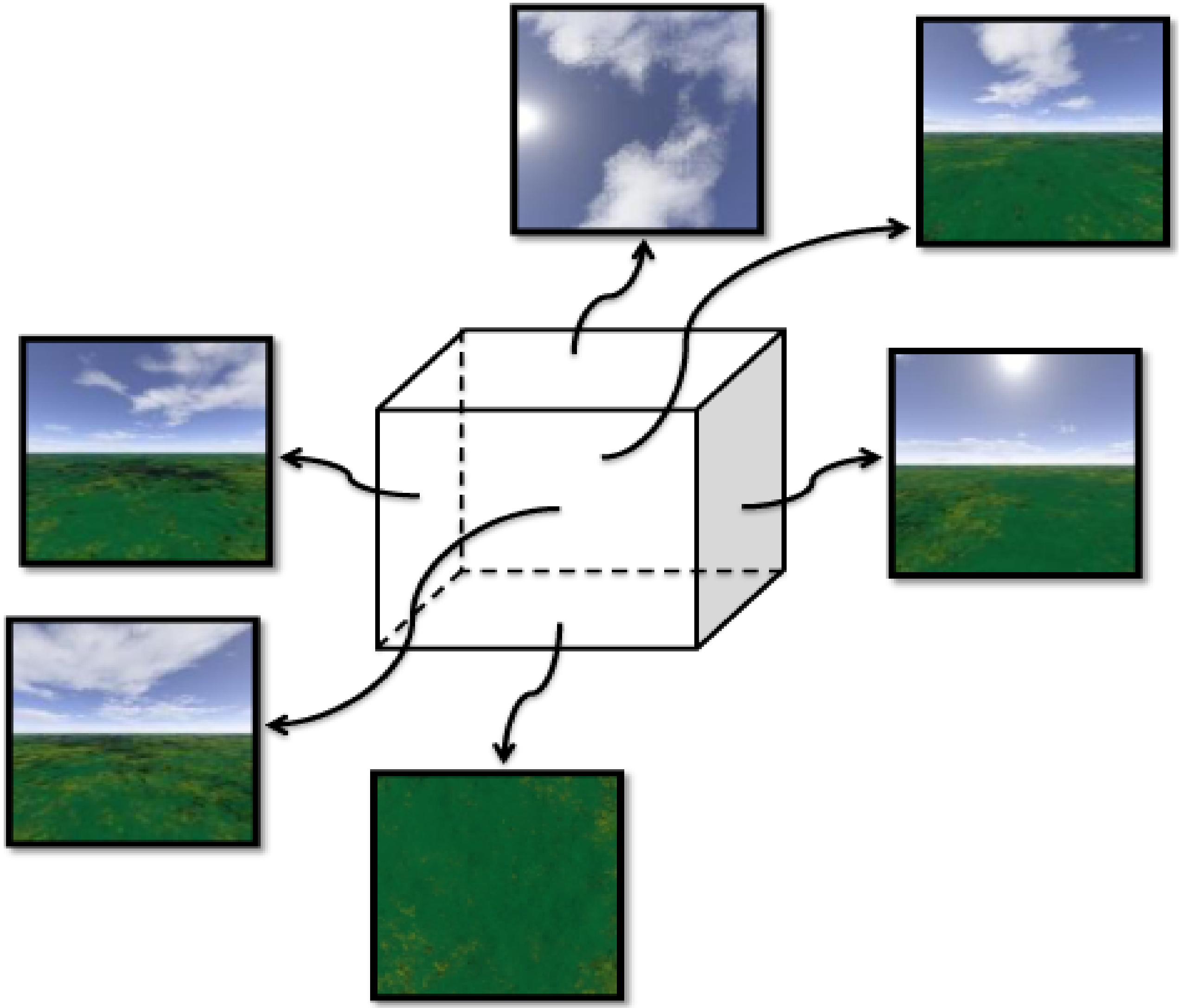
- **Demo – XNA Multiple Lights**



Reflection/Refraction

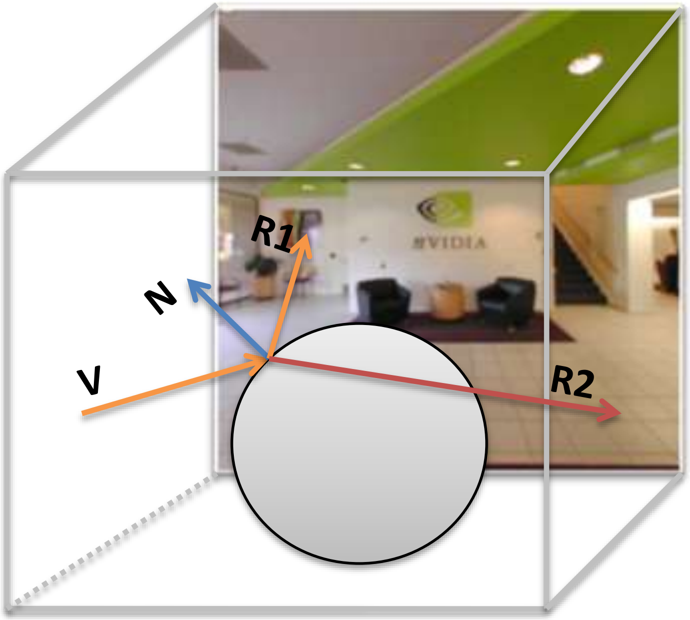
- Environment reflection and refraction are usually achieved through pre-computed or dynamic maps
  - The surround enviroment is rendered to textures
  - These textures are mapped in a solid that covers the entire scene (usually a Cube or a Sphere)
  - Reflection and refraction vectors are used to access these textures
- It is possible to render the enviroment textures on the fly
  - Dynamic Cube Mapping

Cube Mapping

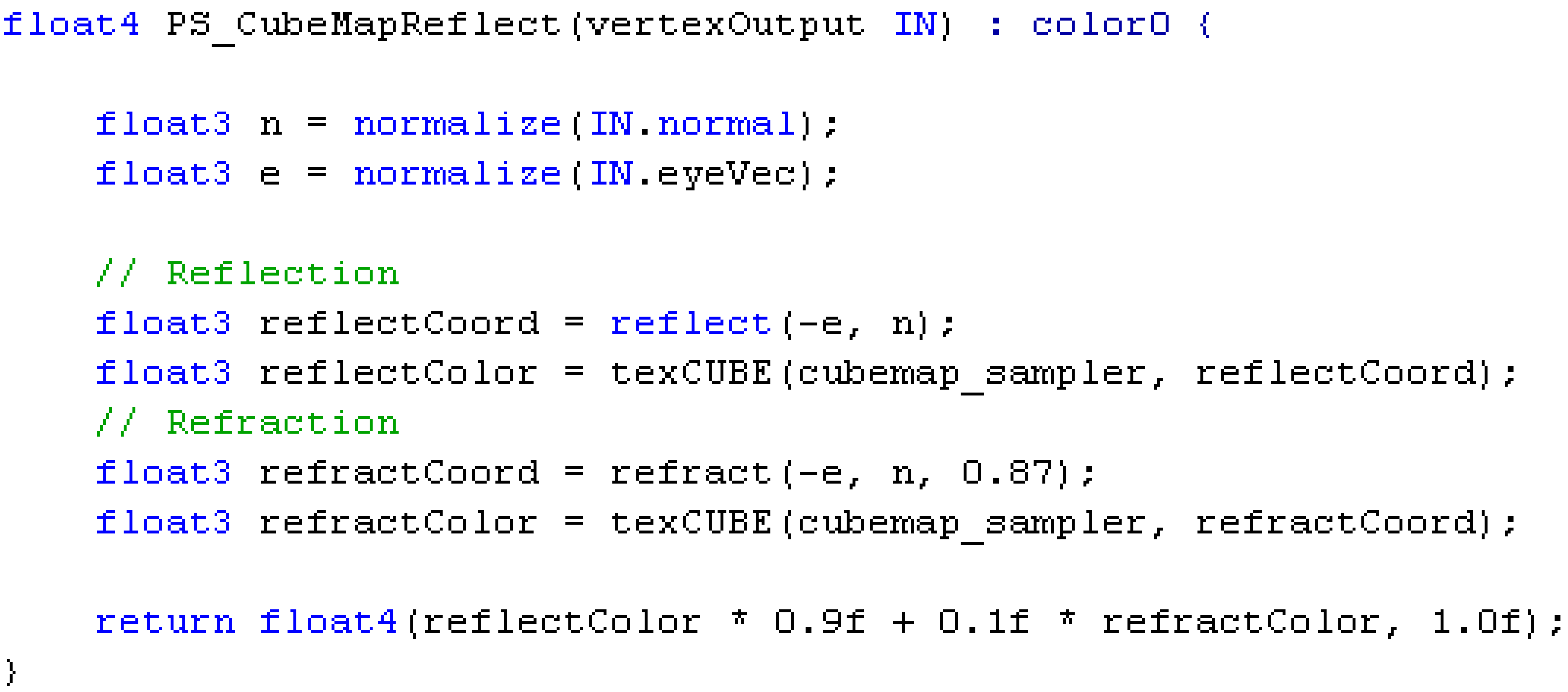


Cube Mapping

- The cube map texture is accessed as a 3-D texture
  - You can use a reflection or refraction vector to access it



Implementing as a Pixel Shader



```
float4 PS_CubeMapReflect(vertexOutput IN) : color0 {

    float3 n = normalize(IN.normal);
    float3 e = normalize(IN.eyeVec);

    // Reflection
    float3 reflectCoord = reflect(-e, n);
    float3 reflectColor = texCUBE(cubemap_sampler, reflectCoord);
    // Refraction
    float3 refractCoord = refract(-e, n, 0.87);
    float3 refractColor = texCUBE(cubemap_sampler, refractCoord);

    return float4(reflectColor * 0.9f + 0.1f * refractColor, 1.0f);
}
```

Final = 90% reflection + 10% de refraction

Yes, you can use Fresnel term here

Cube Mapping

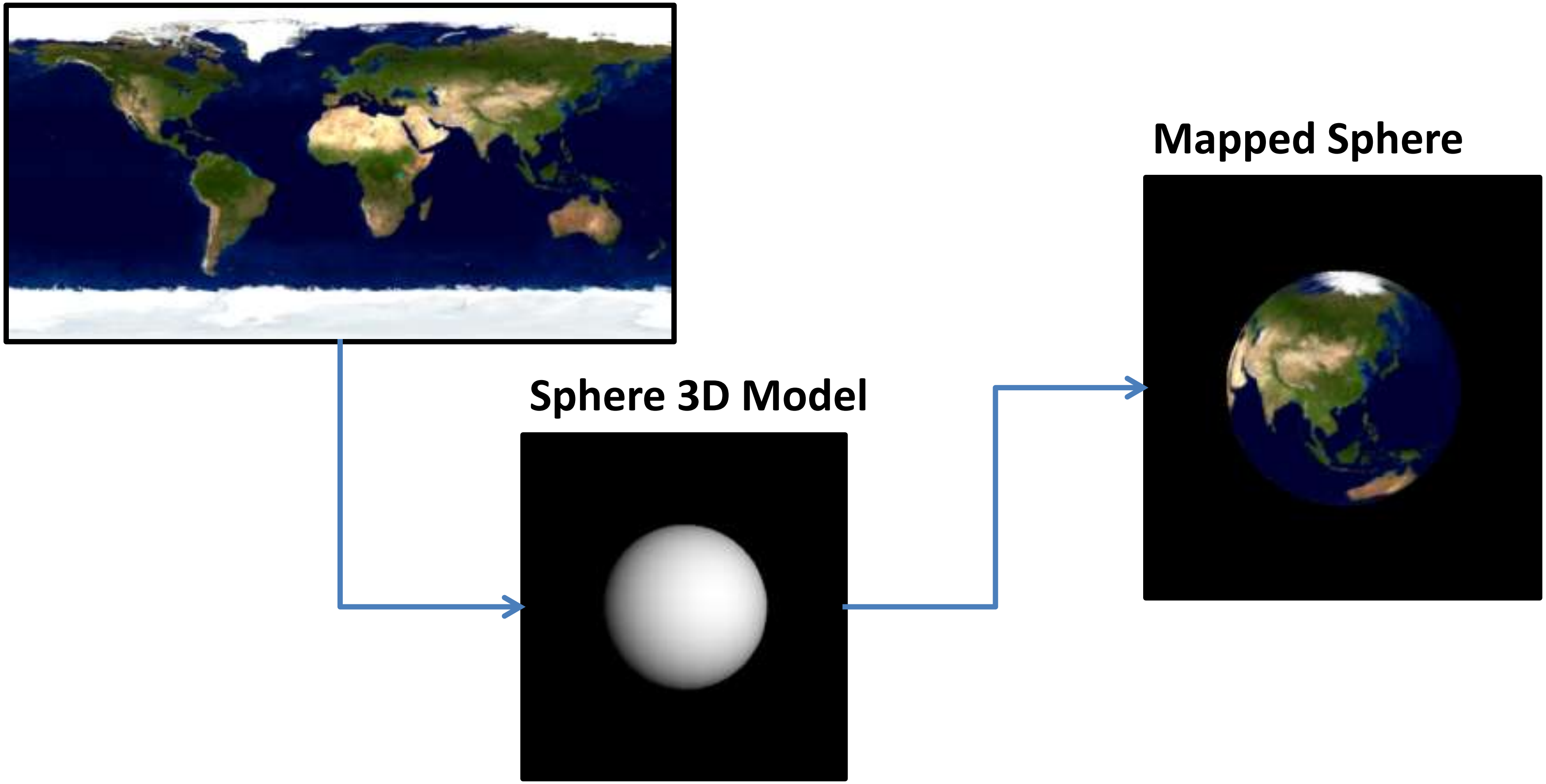
- **Demo – FX Composer**

Textures

- Can be seen as a function that maps a coordinate to a color
- The function can be implemented either:
  - From an image (texture access)
  - From an algorithm (procedural)

Textures

- Mapping a texture over a sphere



Earth Texture

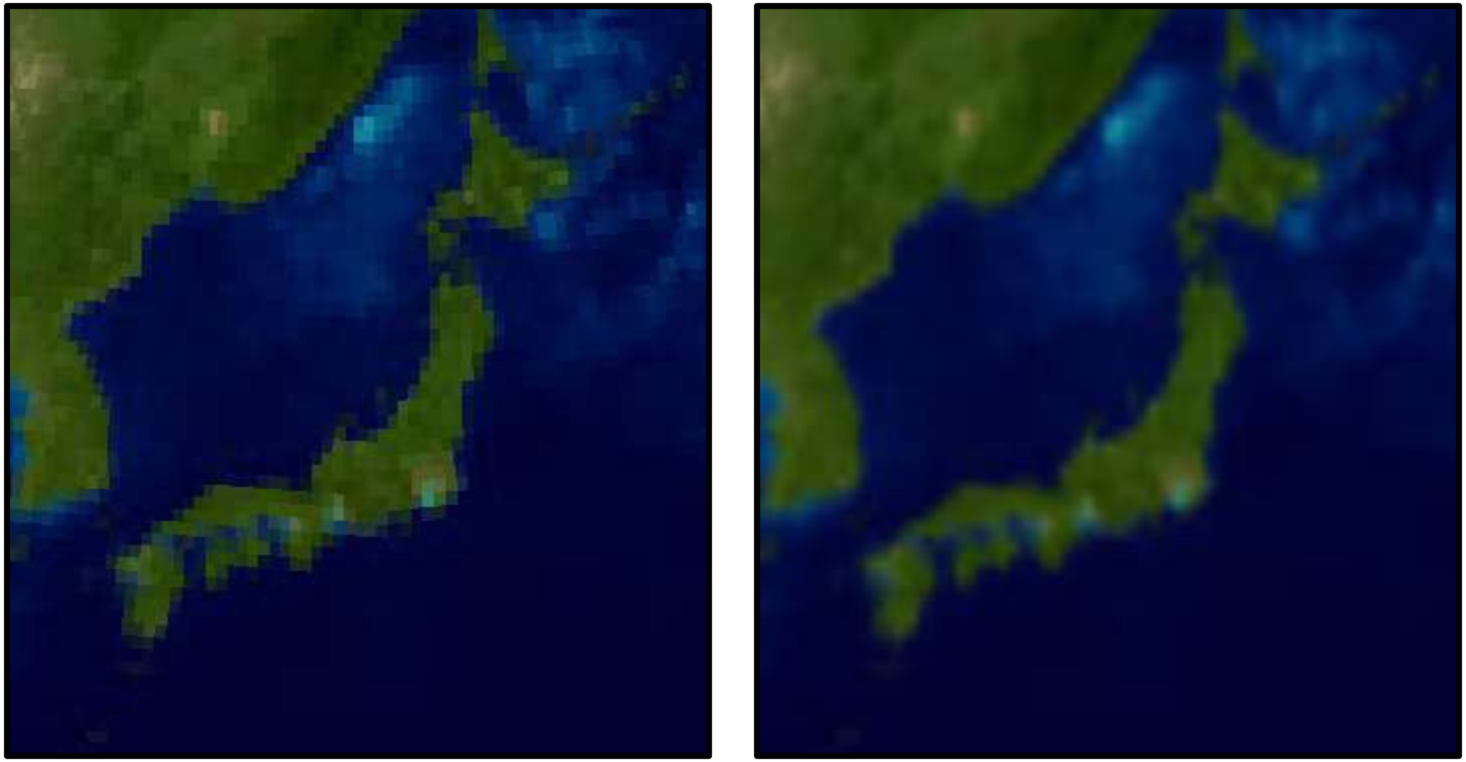
Sphere 3D Model

Mapped Sphere

Earth texture from: http://www.oera.net/How2/TextureMaps2.htm

Textures

- **Sampling configurations**



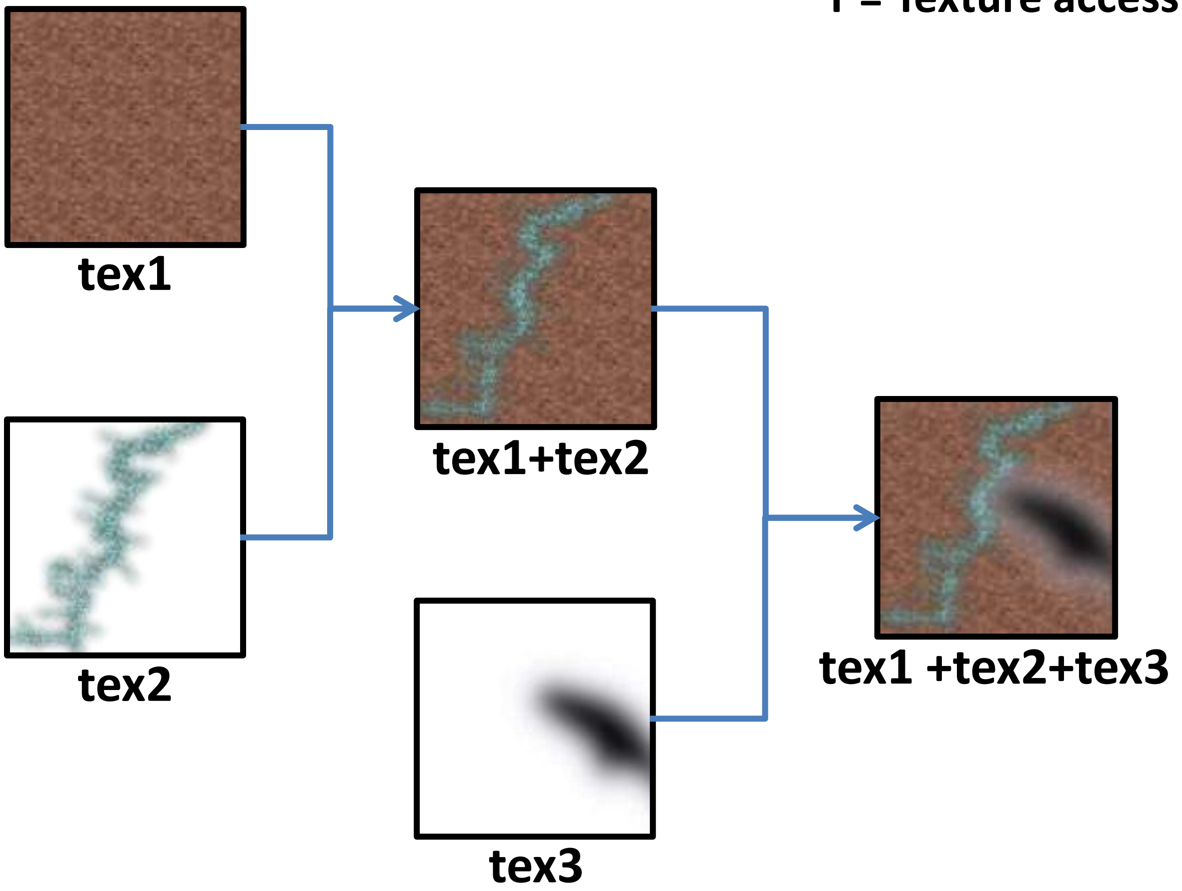
Nearest Filter **Linear Filter**

Multi-Texturing

- **Combining textures**

$$MT_{[n]}(coord) = \sum_{i=1}^{n} T_i(coord) * \alpha_i$$

MT = Multi-texture access
T = Texture access



Textures

- **Demo – FX Composer**
  - **A_GlobalSingle**
  - **B_Multitexture**

Procedural Textures

- Textures that are generated by algorithms
- There are some base models that we can use:
  - Noise (Perlin, 85)
  - Cellular (Worley, 96)
  - Analytical functions
  - Others...

Procedural Textures

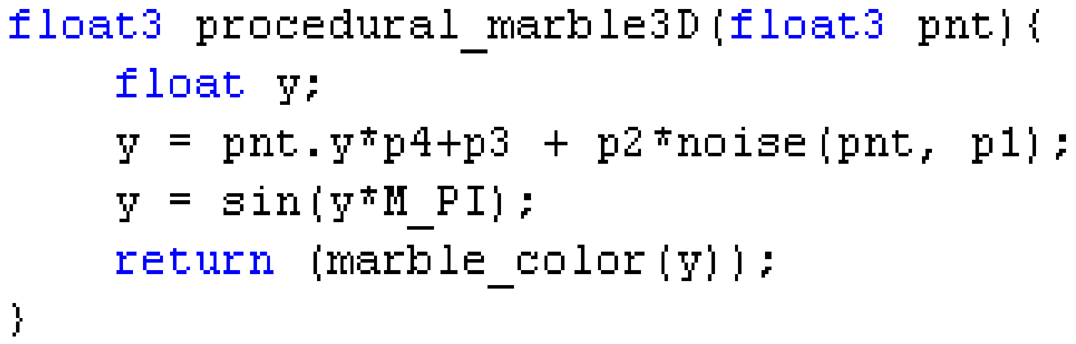
- Interesting features
  - Parametric control
  - Compact representation (you just need to store a few parameters)
  - Some algorithms are easy to implement
- Problems
  - Aliasing
  - Control over details
  - Performance

Procedural Textures

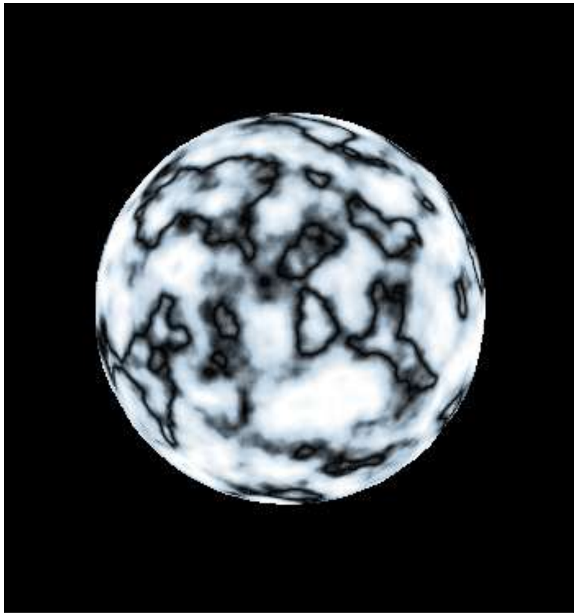
- Noise (Perlin, 85)
  - Tries to model random behavior observed on nature
- Desired properties
  - Statistically invariant to orientation and translation:
    - Maintains the noise appearance over the space
  - Limited dynamic range:
    - Allows the noise to be sampled at different scales without aliasing

Procedural Textures

- Marble generated from one 3D noise function
  - Maps the marble structure over the space



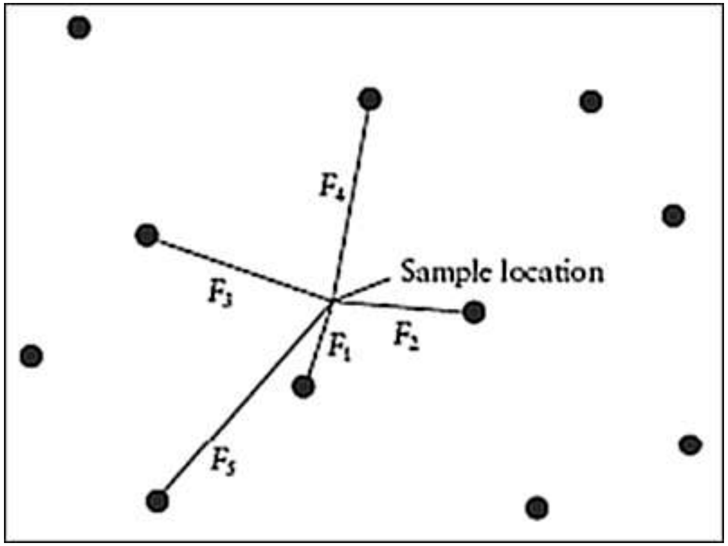
```
float3 procedural_marble3D(float3 pnt){
    float y;
    y = pnt.y*p4+p3 + p2*noise(pnt, p1);
    y = sin(y*M_PI);
    return (marble_color(y));
}
```

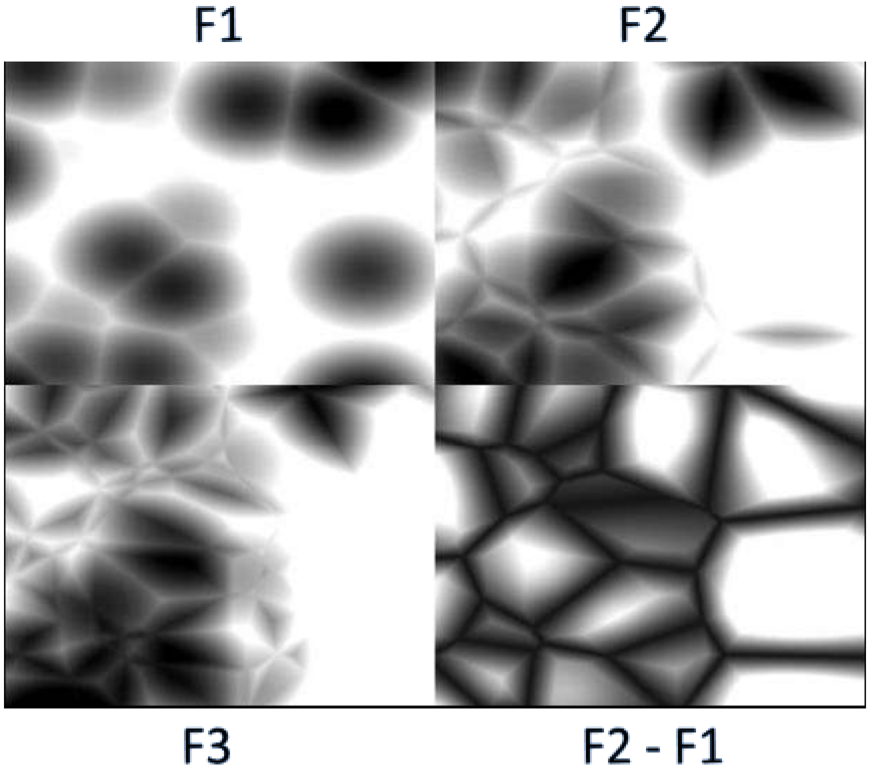


P1, p2, p3 and p4 are the parameters to the marble texture generation based on the noise function

Procedural Textures

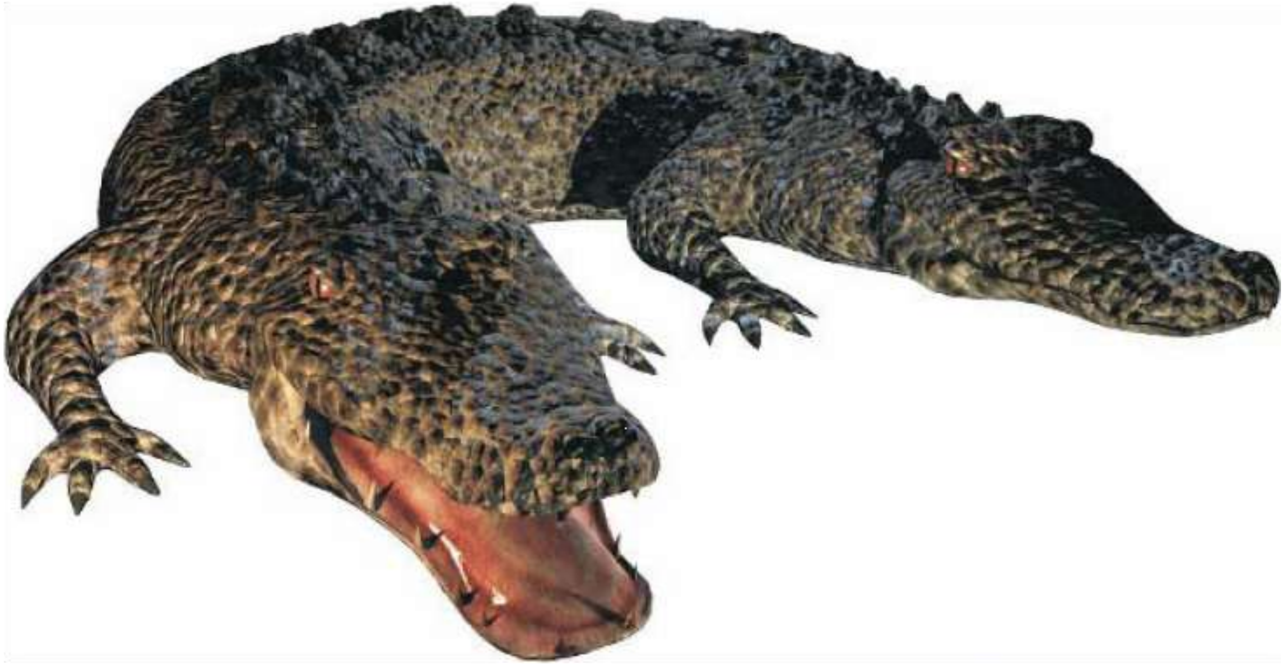
- Cellular texture (Worley, 96)
  - $F_n(x)$ = distance towards the nth closest point
  - $F_n(x) <= F_{n+1}(x)$





Images from Texturing & Modelling Book

Procedural Textures



Images from Texturing & Modelling Book

Procedural Textures

- **Demo – FX Composer**
  - **C_ProceduralSquare**
  - **D_ProceduralMarble**

Simulation of Detailed Surfaces

- In the real world, objects are often composed of highly detailed surfaces
  - Mesostructure: small scale details: bumps, roughness, etc...
  - Microstructure: micro details that are visually indistinguishable but might change how the light is reflected
- Problems
  - Require millions of triangles to be computationally represented (boundary representation)
  - Storage and processing of a big amount of data, unfeasible for real-time rendering

Lucy model

Stanford University

28,055,744 faces
Rendered with VRay Standalone http://www.vrayrender.com/
Model from Standford scanning repository http://graphics.stanford.edu/data/3Dscanrep/

Scanned model

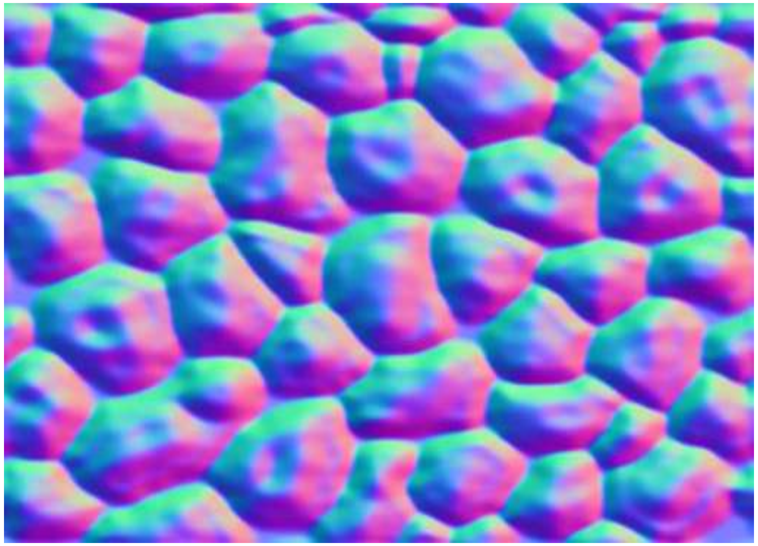
- **116 million triangles**
- **325 MB uncompressed**

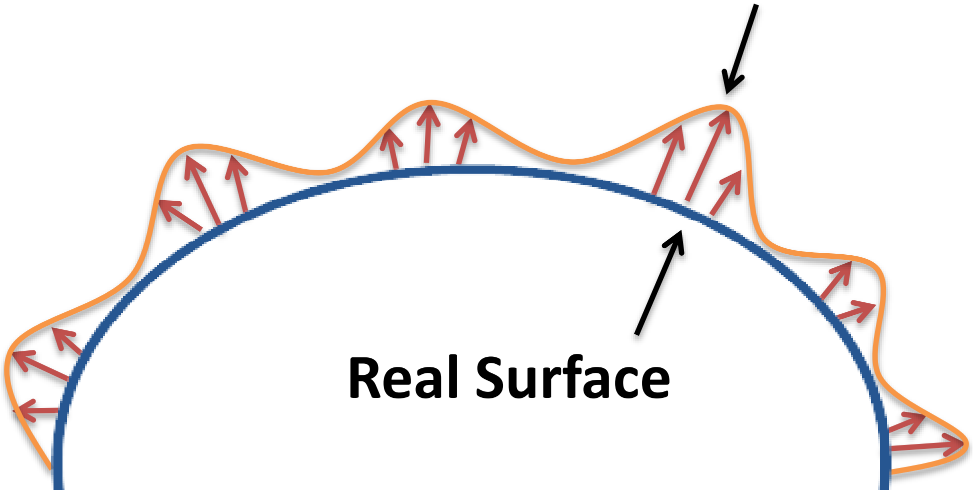
Detalied Surfaces

- Solution
  - Simulate the surface details without increasing its surface complexity
- Some well known techniques
  - Bump Mapping
  - Normal Mapping
  - Offset Parallax Mapping
  - Relief Mapping
  - Parallax Offset Mapping
  - Cone Step Mapping

Normal Mapping

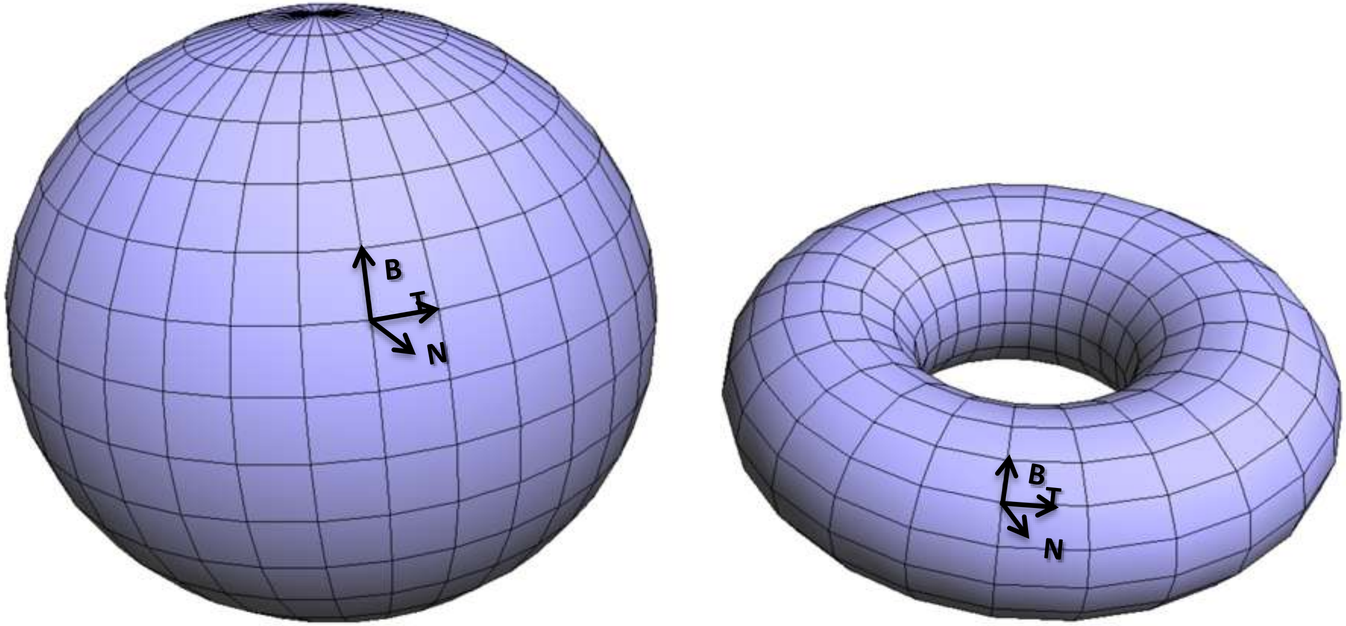
- The surface normals are stored in a texture (normal map) that is mapped to the surface
  - The normal's XYZ axes are mapped to the texture's RGB channels
  - Lighting is computed using the normal map





Normal Mapping

- The computation is made in the tangent space
  - Tangent base formed by the Tangente, Binormal and Normal vectors
- Why?
  - Normal map became independent of the surface

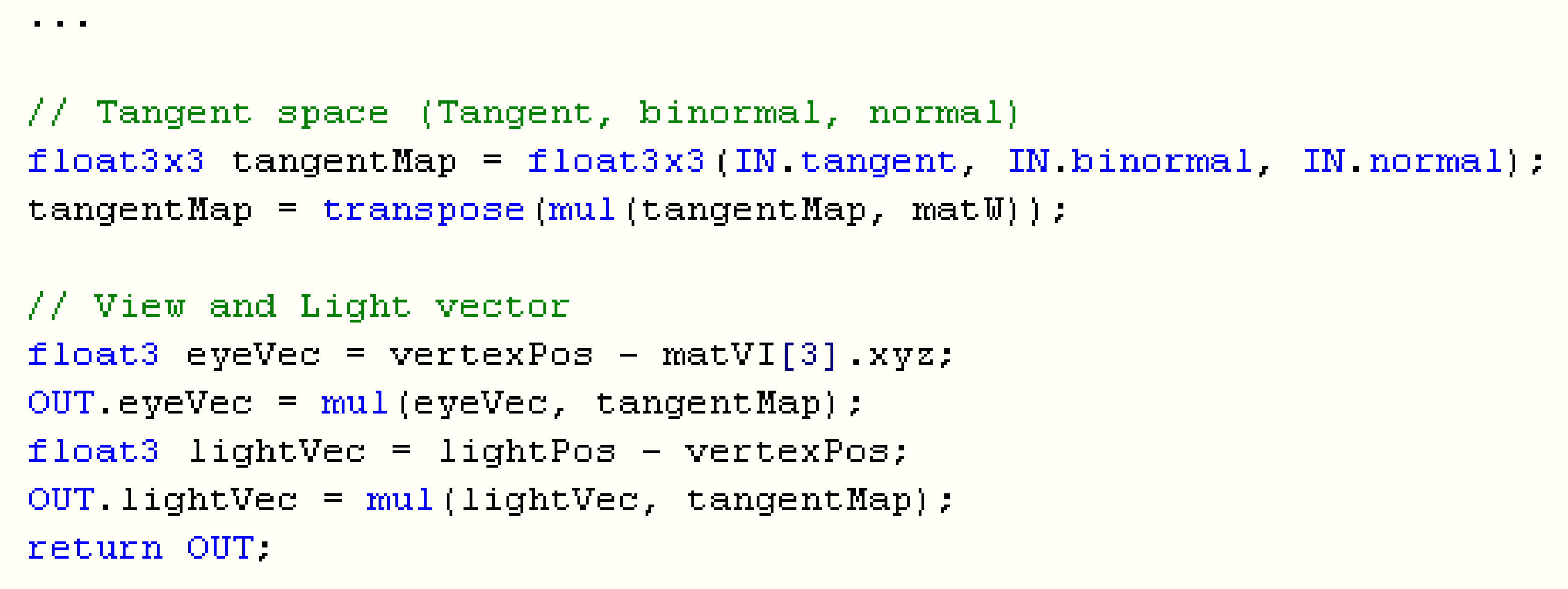


Normal Mapping Shader

- **Vertex Shader – Steps:**
  - **Transform the view and light vectors to the tangent space**
- **Fragment Shader – Steps:**
  - **Read the pixel’s normal from the normal map**
  - **Light the pixel using its new normal, the view vector and the light vectors**

Implementing

Vertex Shader

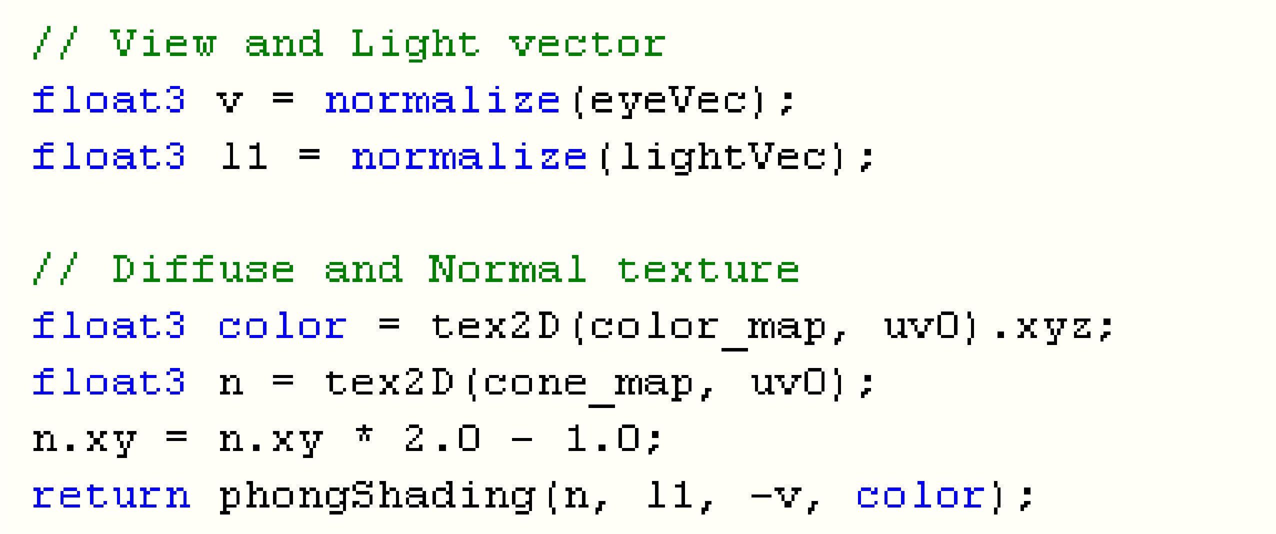


```
...

// Tangent space (Tangent, binormal, normal)
float3x3 tangentMap = float3x3(IN.tangent, IN.binormal, IN.normal);
tangentMap = transpose(mul(tangentMap, matW));

// View and Light vector
float3 eyeVec = vertexPos - matVI[3].xyz;
OUT.eyeVec = mul(eyeVec, tangentMap);
float3 lightVec = lightPos - vertexPos;
OUT.lightVec = mul(lightVec, tangentMap);
return OUT;
```

Pixel Shader

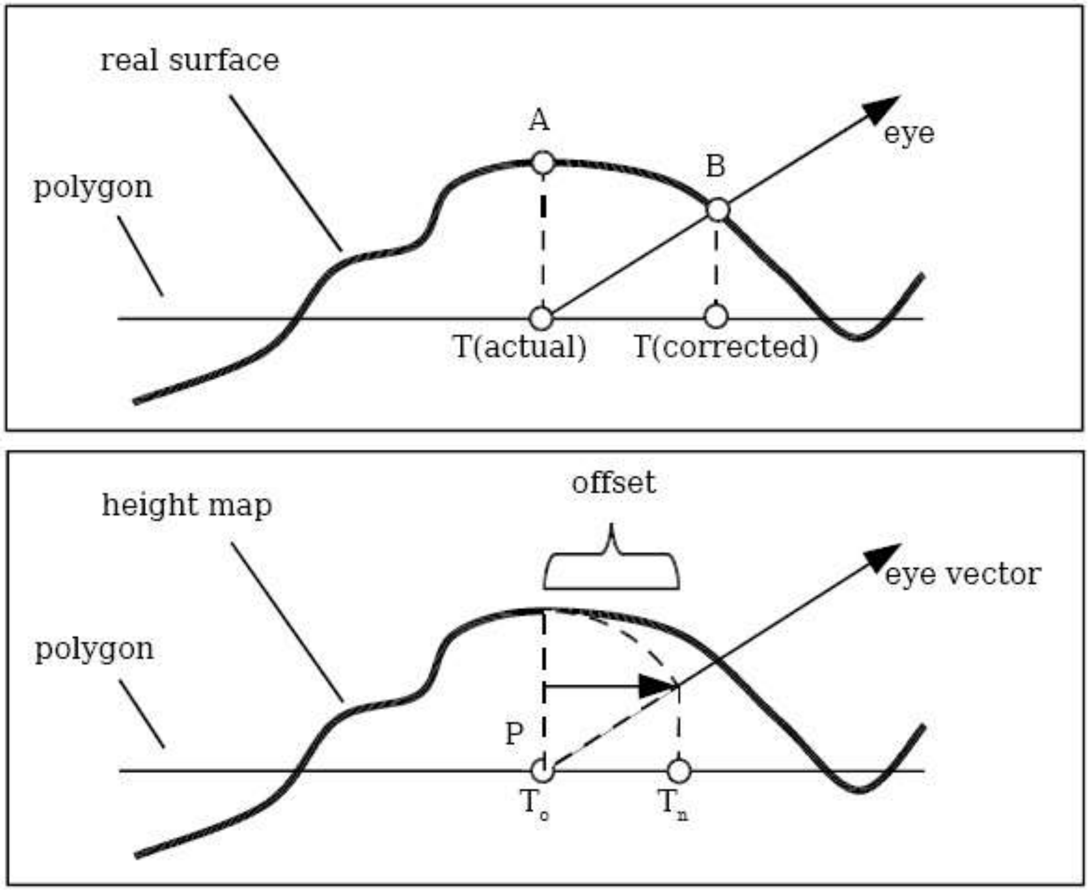


```
// View and Light vector
float3 v = normalize(eyeVec);
float3 l1 = normalize(lightVec);

// Diffuse and Normal texture
float3 color = tex2D(color_map, uv0).xyz;
float3 n = tex2D(cone_map, uv0);
n.xy = n.xy * 2.0 - 1.0;
return phongShading(n, l1, -v, color);
```

Offset Parallax Mapping

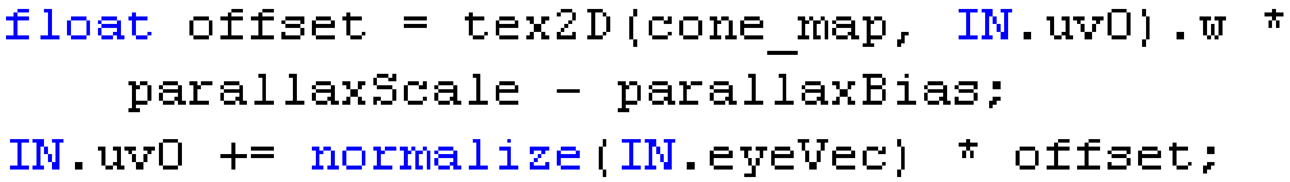
- **An heuristic to handle the parallax effect**
  - **Improve the normal mapping result**



Images from Parallax mapping with offset limiting [Welsh 04]

Implementing as a Pixel Shader

- **Calculate the correct texture coordinate based on the parallax offset**



```
float offset = tex2D(cone_map, IN.uv0).w *
    parallaxScale - parallaxBias;
IN.uv0 += normalize(IN.eyeVec) * offset;
```

Relief Mapping/POM Mapping

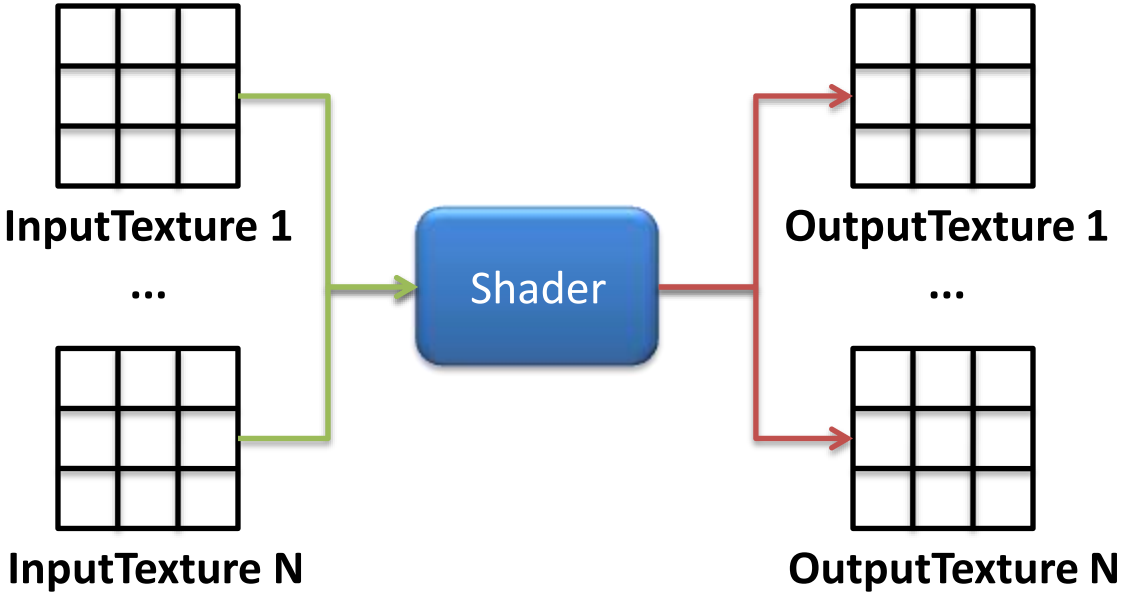
- A powerful technique to render very detailed surfaces accurately
  - Uses a ray-tracing algorithm for the ray-heightfield intersection on the GPU
  - Needs a lot of iterations to find the correct viewed poisition over the surface

Detalied Surfaces

- **Demo – Detailed Surfaces**

Post-Processing

- Effects that are applied over the rendered image (or render targets)
  - A pos-processing shader may have many input and output textures



InputTexture 1

…

InputTexture N

Shader

OutputTexture 1

…

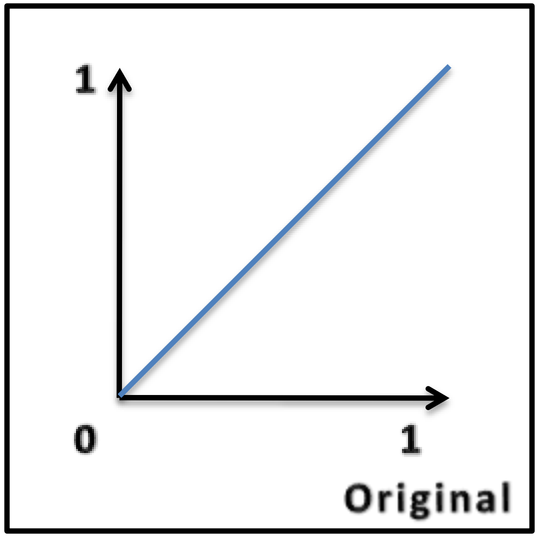
OutputTexture N

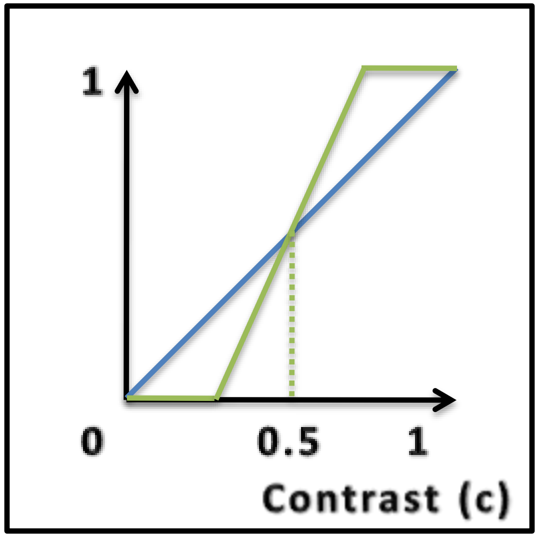
Post-Processing

- Digital Image Processing (DIP) algorithms that can be applied
  - Radiometric transformations
    - Contrast, brightness, and grayscale conversion
  - Filters
    - Blur, edge detection
  - Image composition
    - Radial motion blur

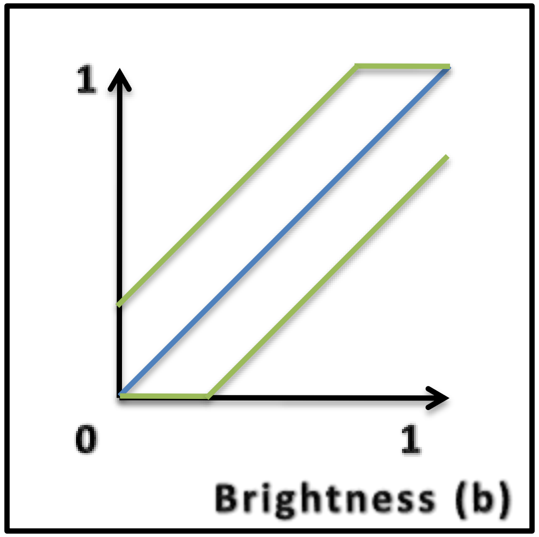
Radiometric Transformations

- **Contrast and Brightness**





$F(x,y) = [F(x,y)-0.5]*c + 0.5$



$F(x,y) = F(x,y) + b$

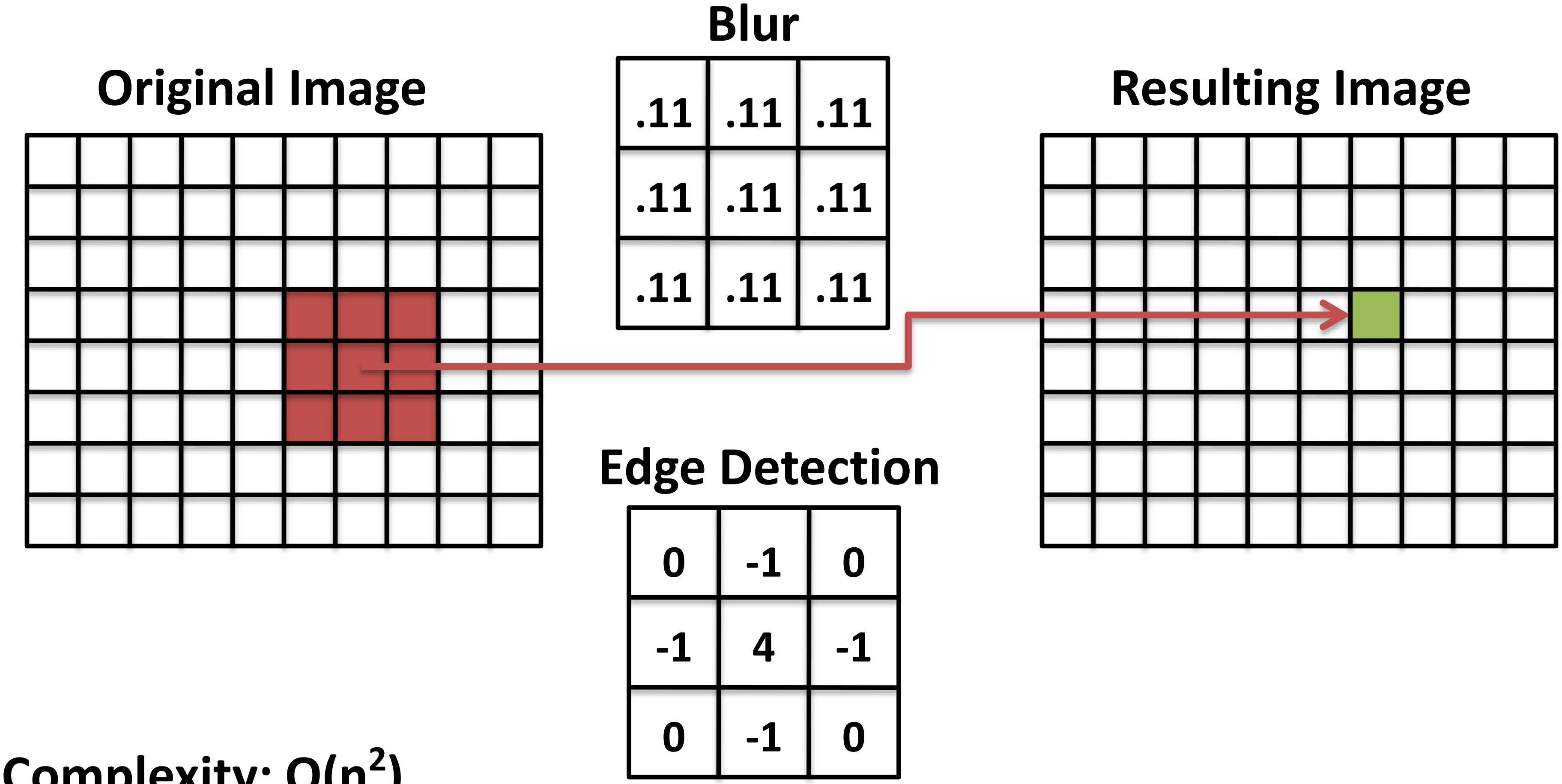
Radiometric Transformations

- Grayscale conversion
  - Considering the HSV color space
    - Gray = V = (R + G + B)/3
    - Gray = V = Max(R, G, B)
  - Considering the humam perception (YIQ)
    - The YIQ color space is used in the NTSC signal
    - Gray = Y = R*0.299 + G*0.587 + B*0.114

Radiometric Transformations

- Demo – FX Composer
  - E_ppBrilhoContraste
  - F_ppCinza

Filters



Original Image

Blur

.11	.11	.11
.11	.11	.11
.11	.11	.11

Edge Detection

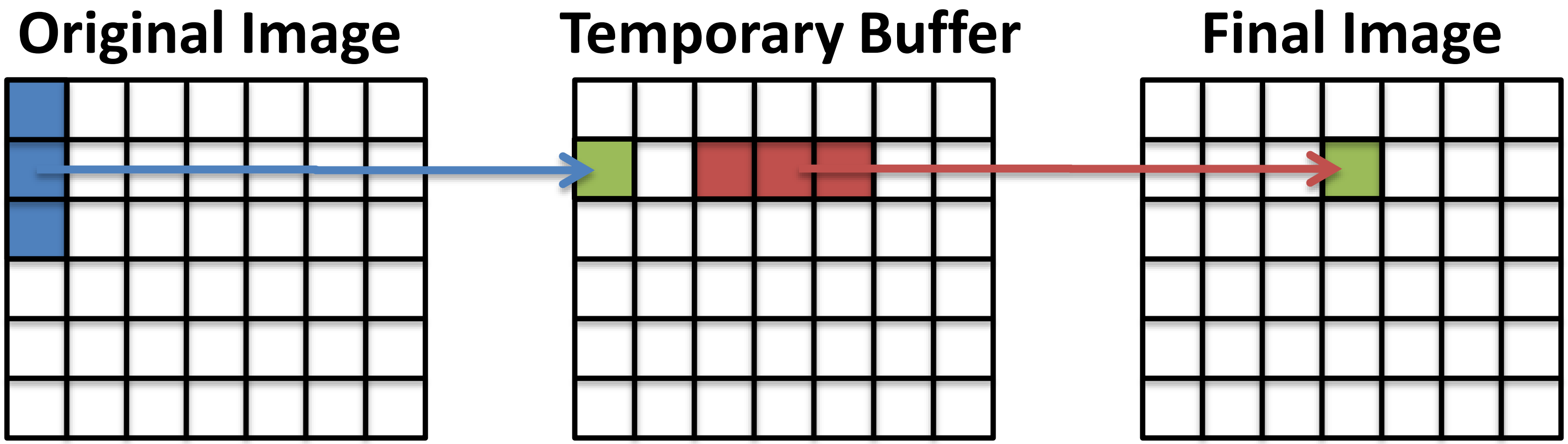
0	-1	0
-1	4	-1
0	-1	0

Resulting Image

Complexity: O(n^2)

Blur Filter

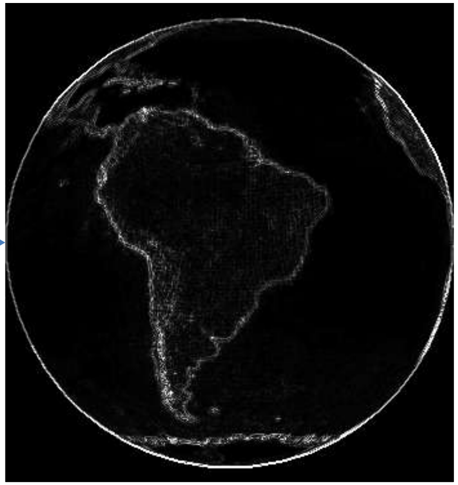
- This filter can be optimized using two passes simpler passes



Complexity: O(n)

Filters

Original Image



Edge Detetion

Vertical Blur

Horizontal Blur

Filters

- **Demo – FX Composer**
  - **G_ppBlur**
  - **H_ppLaplace**

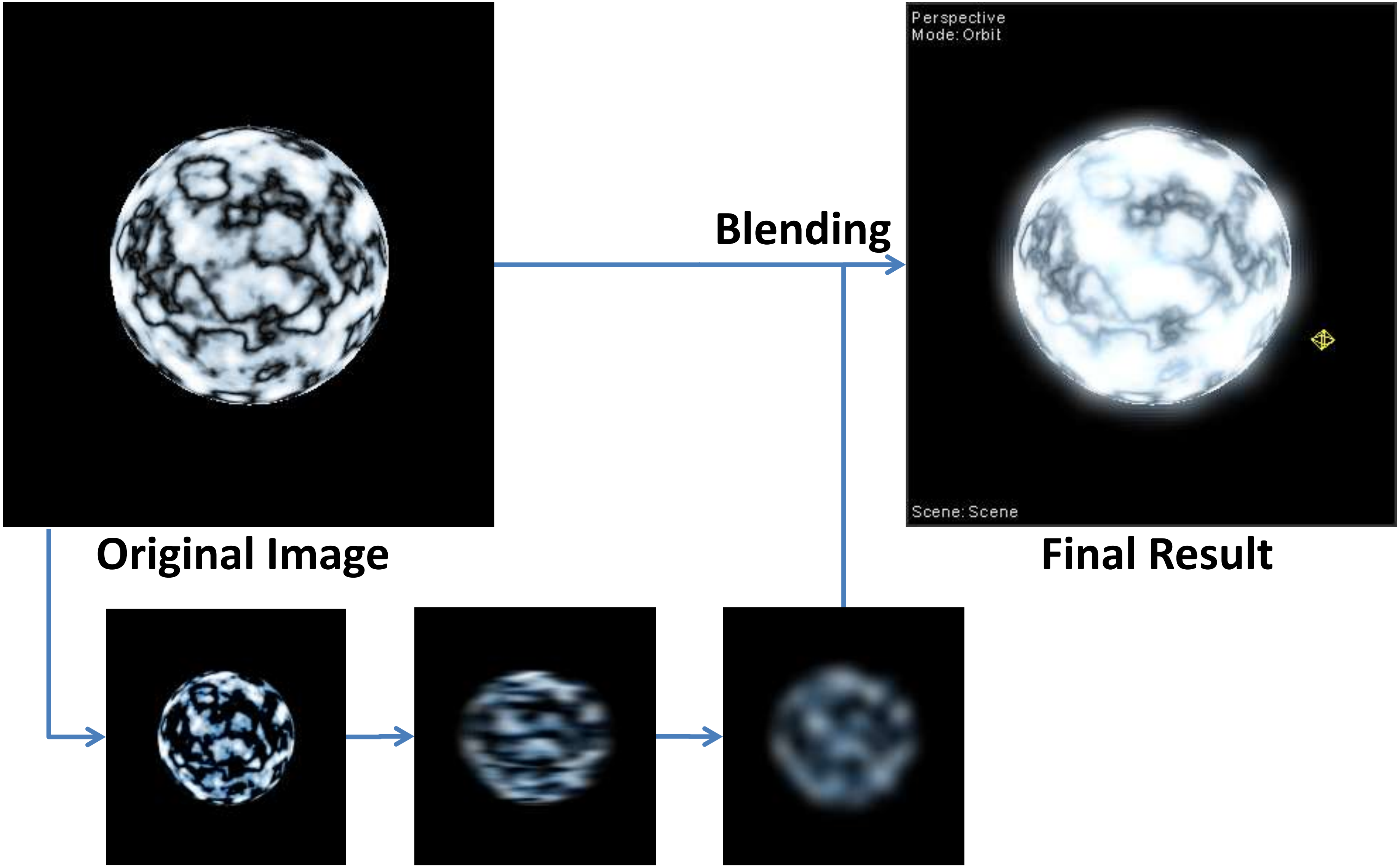
Post-Processing

- Composed effects
  - Some effects are composed by many rendering passes, where it is necessary to use some auxiliary buffers (or textures)
  - In this case, it is necessary to have a rendering flow control (usually implemented in software)
    - The rendering flow control should save resources (video memory) and manage the render targets
- Examples
  - Bloom, Cartoon Rendering, outros...

Bloom

- Try to simulate the light expansion over bright surfaces
  - Effect perceived in the real world
  - Lights, Reflections and so on
- It is often used in with HDR (High Dynamic Range) techniques

Bloom



Blending

Original Image

Final Result

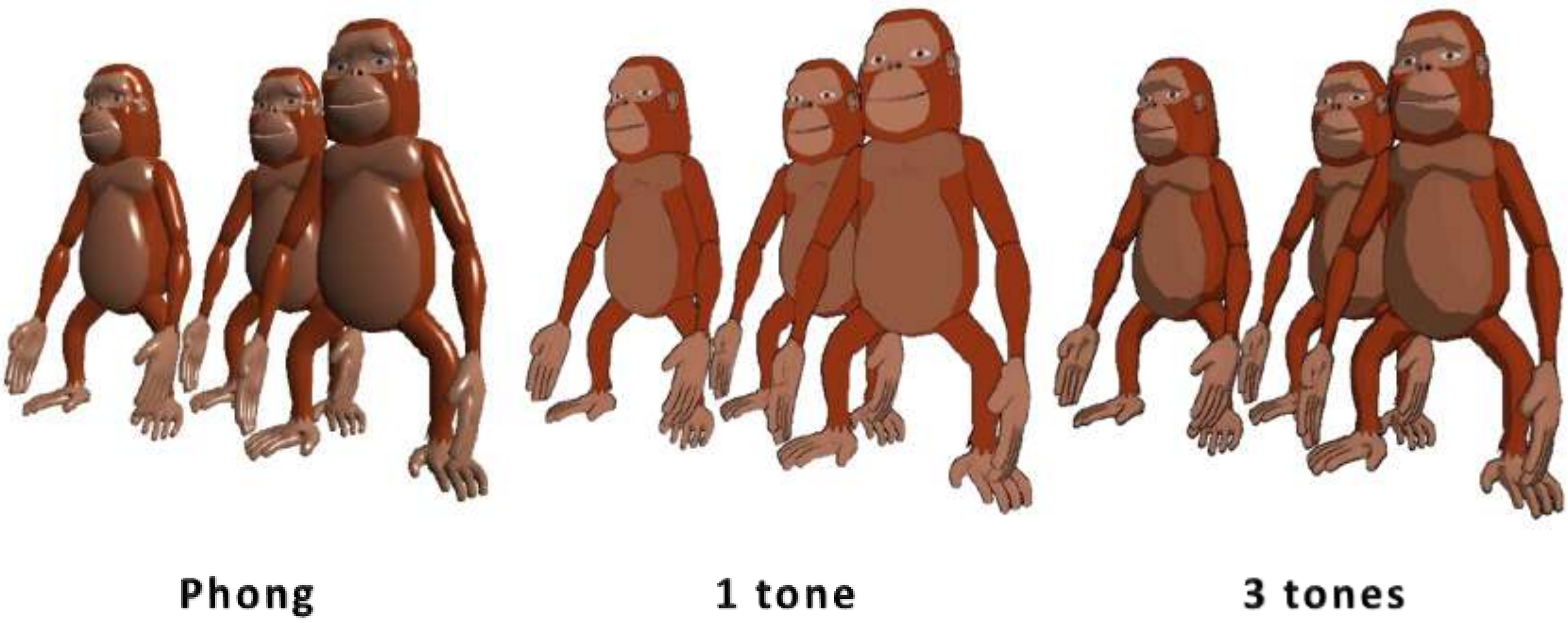
Threshould

Horizontal Blur

Vertical Blur

Cartoon Rendering

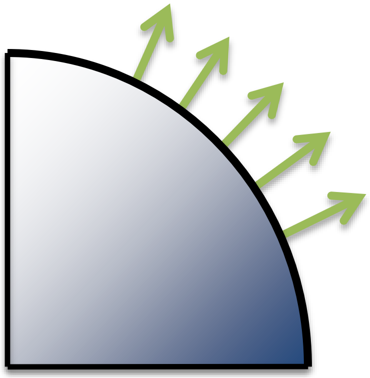
- Render a scene like a hand drawn cartoon
  - Fixed and small number of tones
  - Edge outline



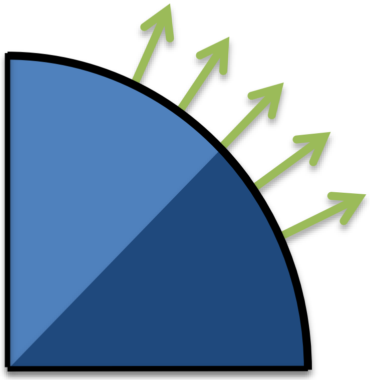
Cartoon Rendering

- Maps the result of a phong lighting model to a fixed number of tones (usually two or three)

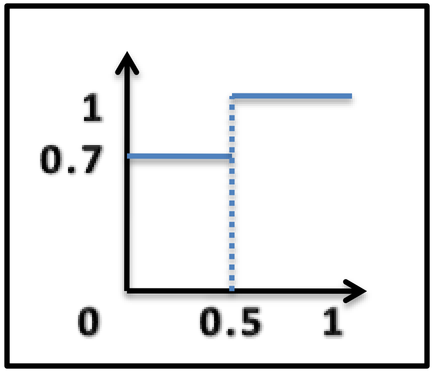
```
IF Phong(x, y, z) < 0.5
    Intensity = 0.7,
ELSE
    Intensity = 1
```



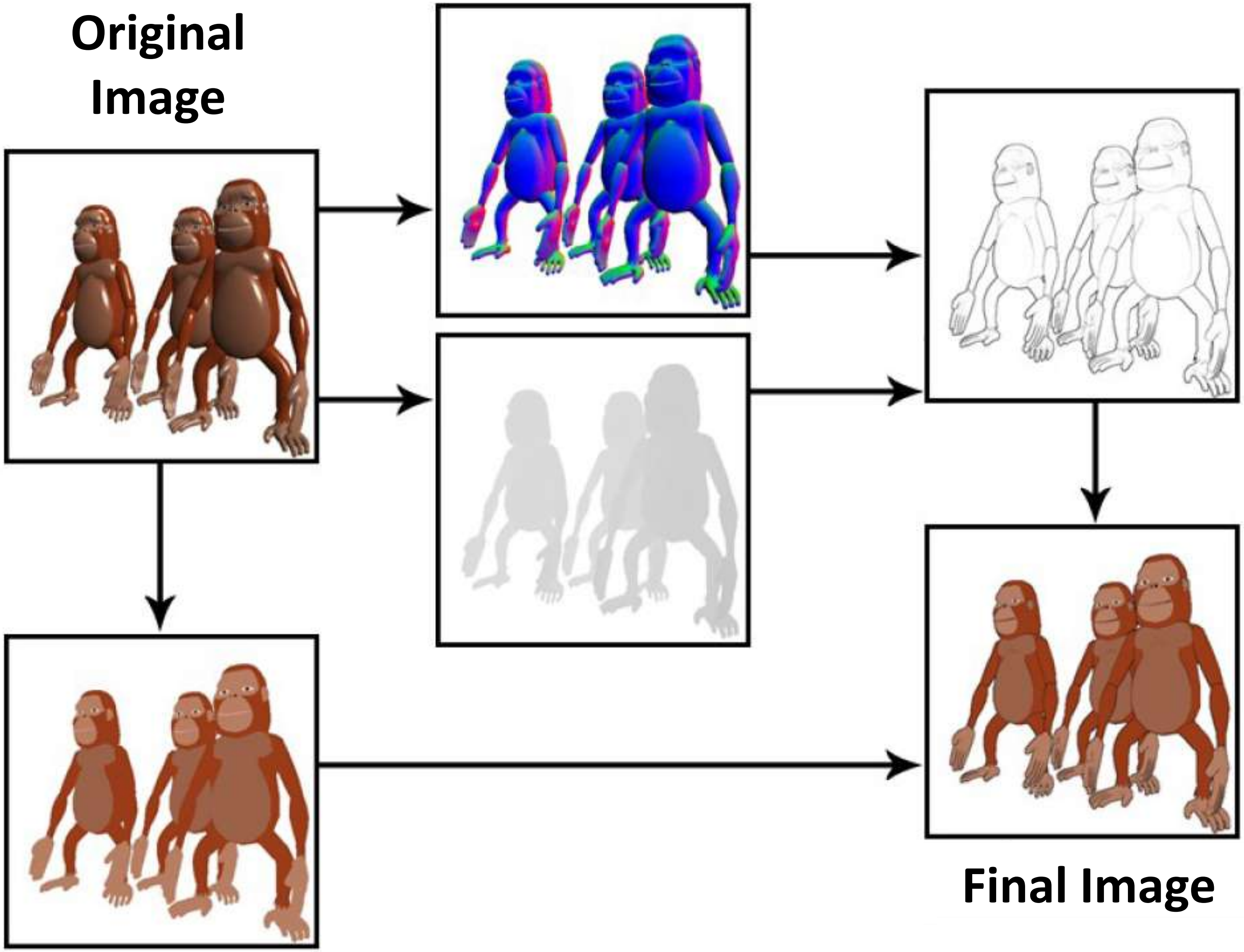
Original Lighting



Quantized Lighting
Using two tones



Cartoon Rendering



Original Image

Final Image

Post-Processing

- **Demo – FX Composer**
  - **I_Bloom**
  - **Cartoon**

Conclusions

- In this lecture we presented some effects that are used in current commercial games
    - All these effects are implemented on modern GPUs
- In a near future the GPU will be completly programmable and its fixed stages removed
    - Modern APIs doesn’t support the fixed pipeline anymore (DirectX 10, XNA and OpenGL ME)
    - Nowadays there are still a few stages that remains fixed: Rasterization and Output Merger

Thank You!

Bruno P. Evangelista

www.BrunoEvangelista.com

bpevangelista@gmail.com

Alessandro R. Silva

www.AlessandroSilva.com

alessandro.ribeiro.silva@gmail.com

“For what will it profit a man if he gains the whole world and forfeits his soul? Or what will a man give in exchange for his soul?” Matthew 16:26