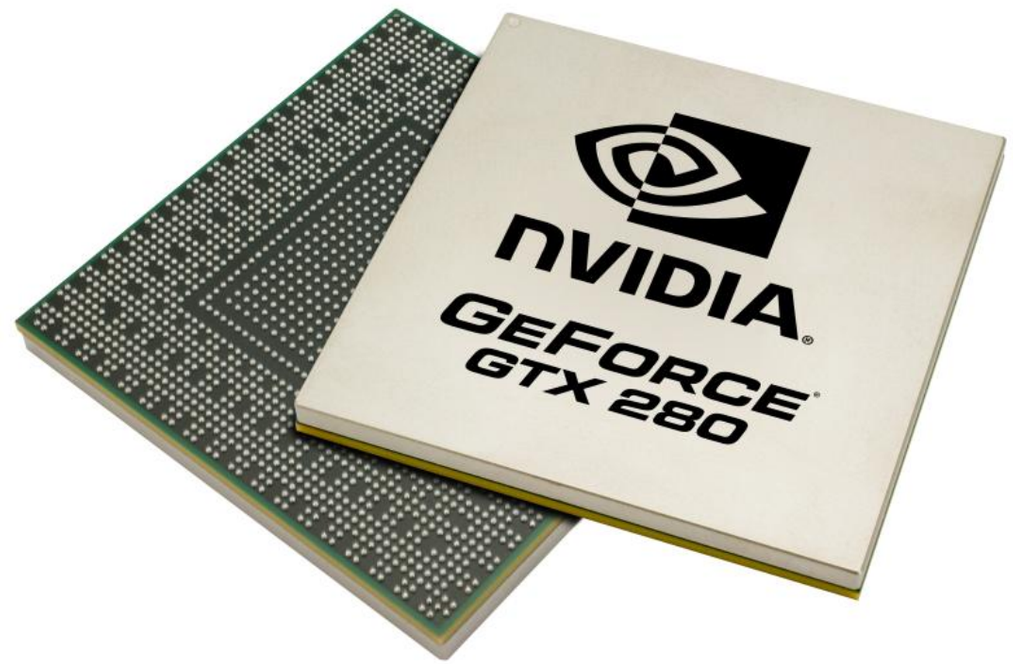
Class Evaluation

- The class will start 5 minutes late
- Please give us the feedback on 6.837
- Thank you!!!

http://web.mit.edu/subjectevaluation

Graphics Hardware



MIT EECS 6.837

Wojciech Matusik

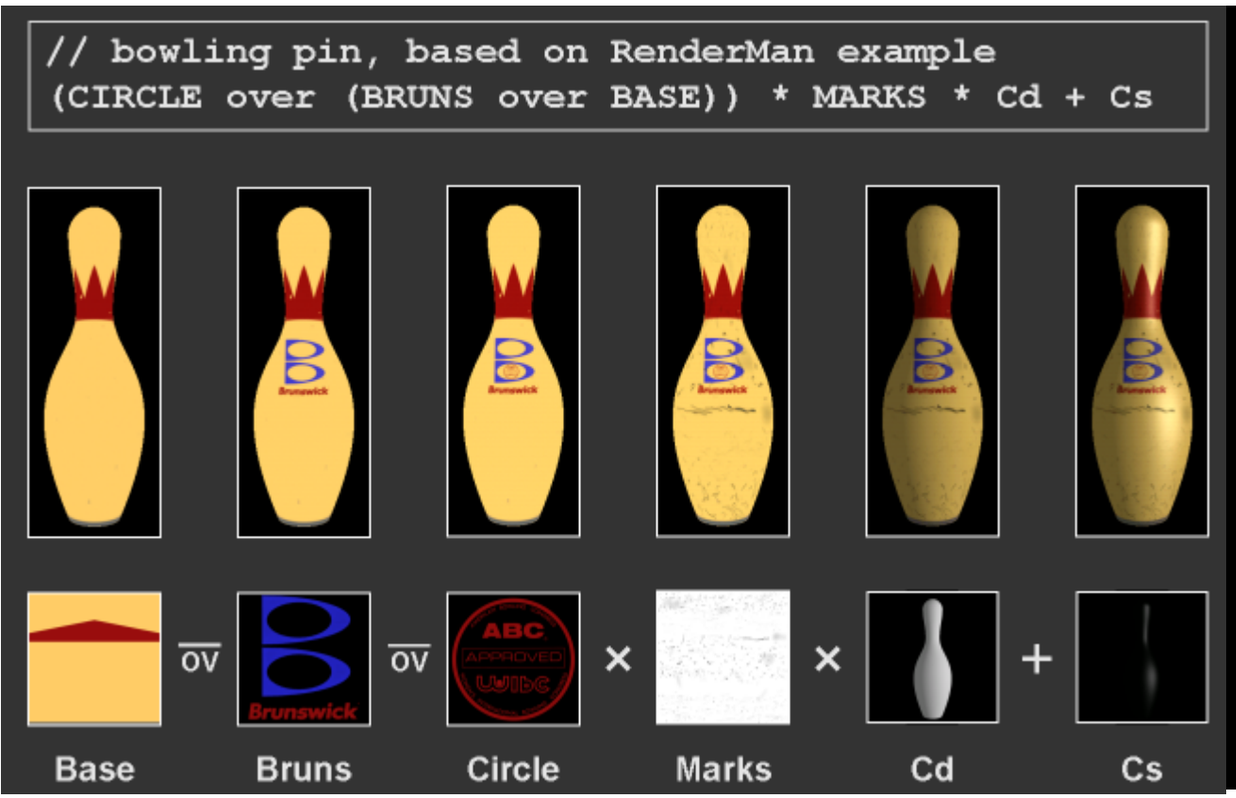
with slides from
Fredo Durand, Jonathan Ragan-Kelley, Hanrahan & Akeley, Gary McTaggart, NVIDIA, ATI

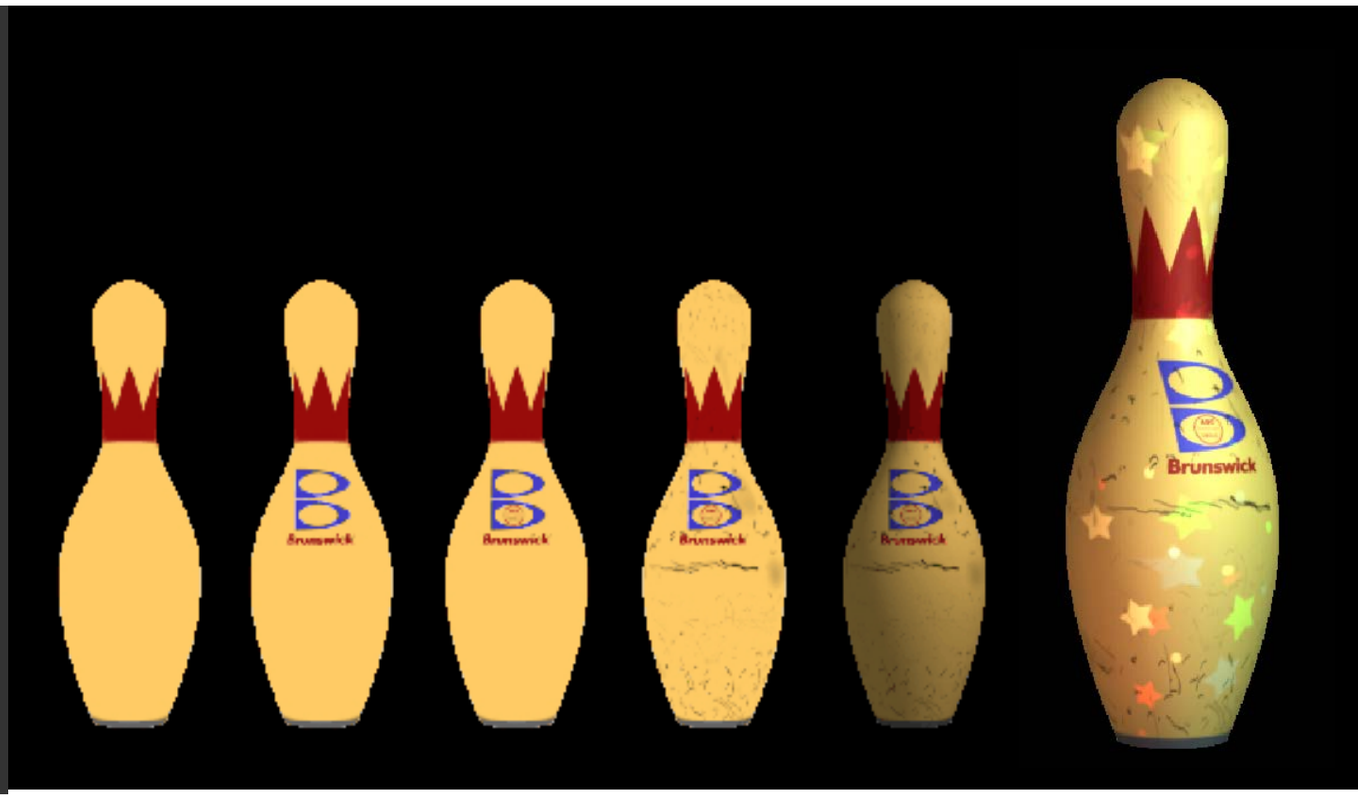
Final exam

- 4 pages (2 double-sided sheets) of notes
- Everything we saw this semester
  - With emphasis on 2nd part
- Similar to quiz 1
- Will target a 2-hour subject

Modern Graphics Hardware (“GPU”)

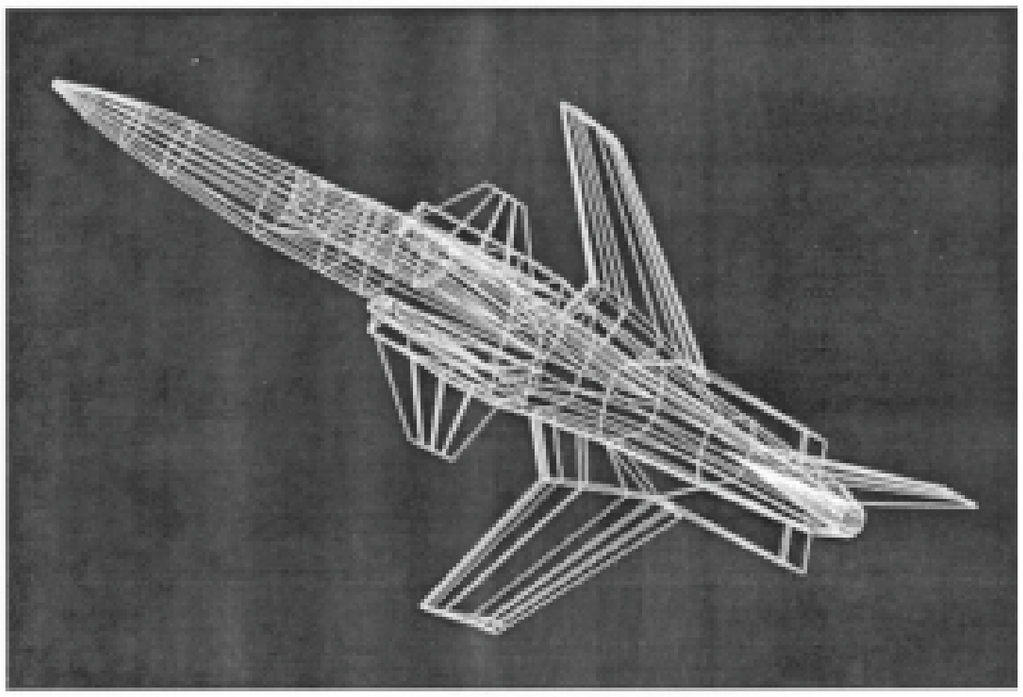
- Hardware implementation of the *rendering pipeline*
- Programmability (“shaders”) within specific stages of the pipeline
  - Recent, last 10 years
  - At the vertex and pixel level





First Generation - Wireframe

- Vertex: transform, clip, and project
- Rasterization: color interpolation (points, lines)
- Fragment: overwrite
- Dates: prior to 1987



Slide from Stanford CS448A, Akeley & Hanrahan, 2007

Second Generation - Shaded Solids

- Vertex: lighting calculation
- Rasterization: depth interpolation (triangles)
- Fragment: depth buffer, color blending
- Dates: 1987-1992

Slide from Stanford CS448A, Akeley & Hanrahan, 2007

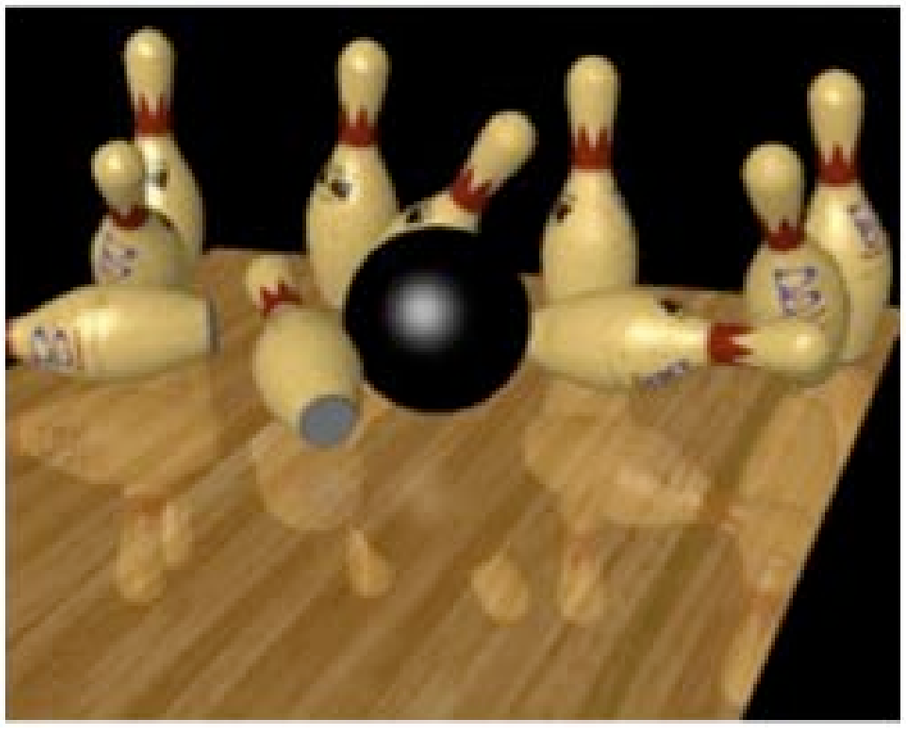
Third Generation - Texture Mapping

- Vertex: texture coordinate transformation
- Rasterization: texture coordinate interpolation
- Fragment: texture evaluation, antialiasing
- Dates: 1992-2000

Slide from Stanford CS448A, Akeley & Hanrahan, 2007

Fourth Generation - Programmable Shading

- Vertex: programmable transformation
- Rasterization: user attribute interpolation
- Fragment: programmable color computation
- Dates: 2000-2009

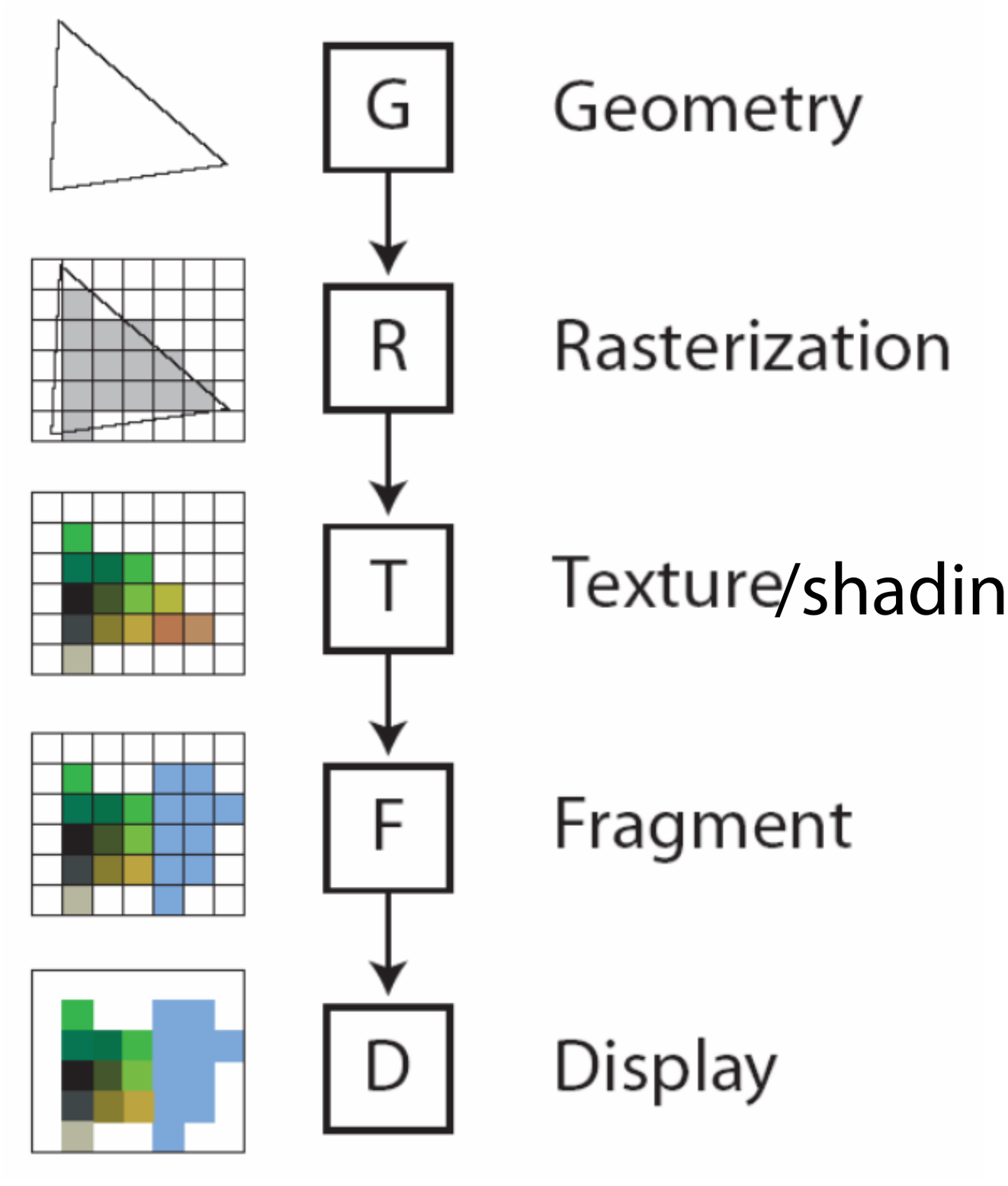


Fifth Generation - Programmable Pipeline

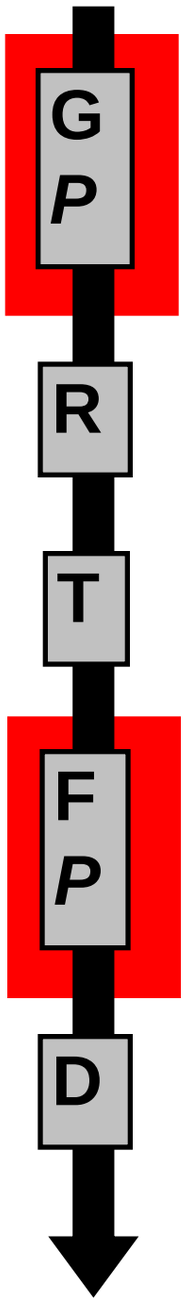
- Vertex,
  Rasterization,
  Fragment, ???: arbitrary parallel programs
- Dates: 2009+

Questions?

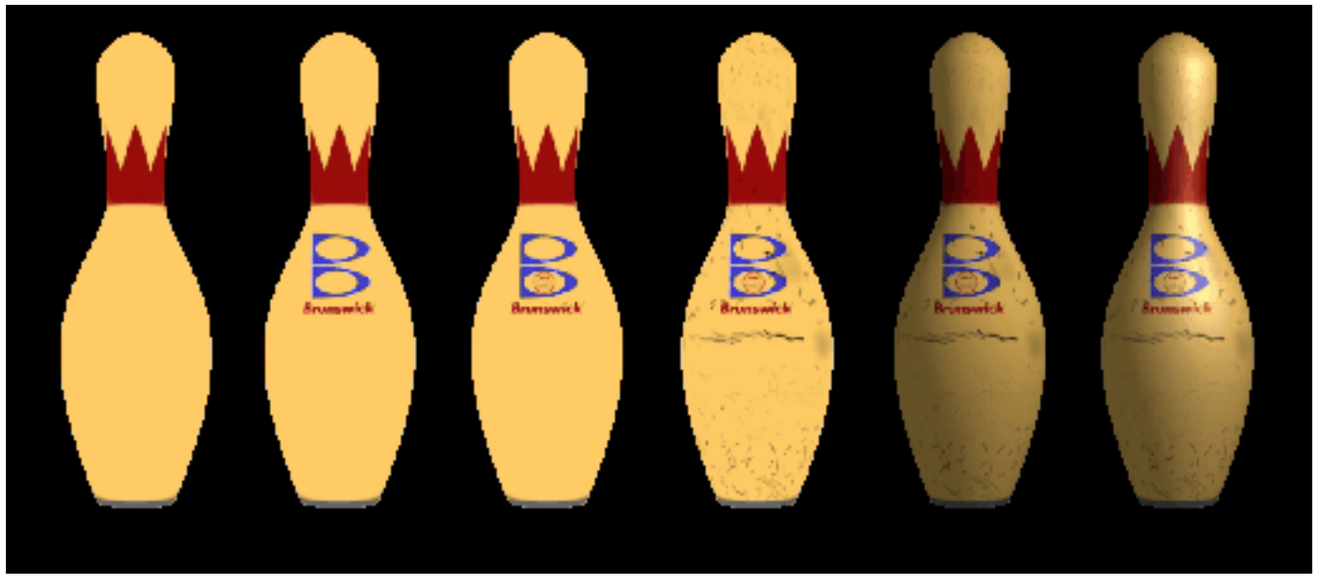
The Graphics Pipeline



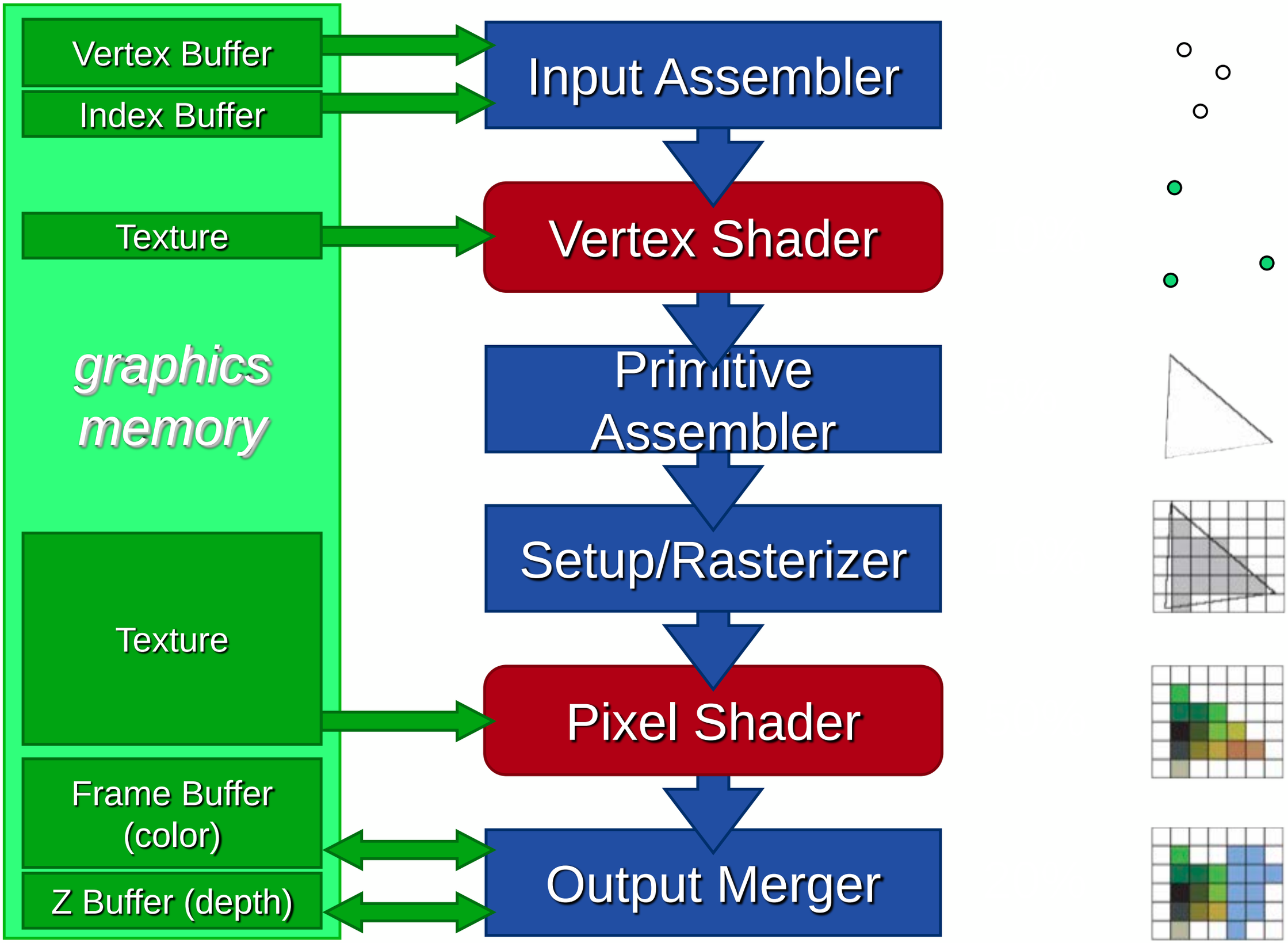
Programmable Graphics Hardware



- Geometry and Pixel (fragment) stage become programmable
  - Elaborate appearance
  - More and more general-purpose computation (GPGPU)



The Direct3D 9 Graphics Pipeline



Questions?

Vertex Shaders

f(position, attributes) ⟶ new position, attributes

purely functional (no side-effects)

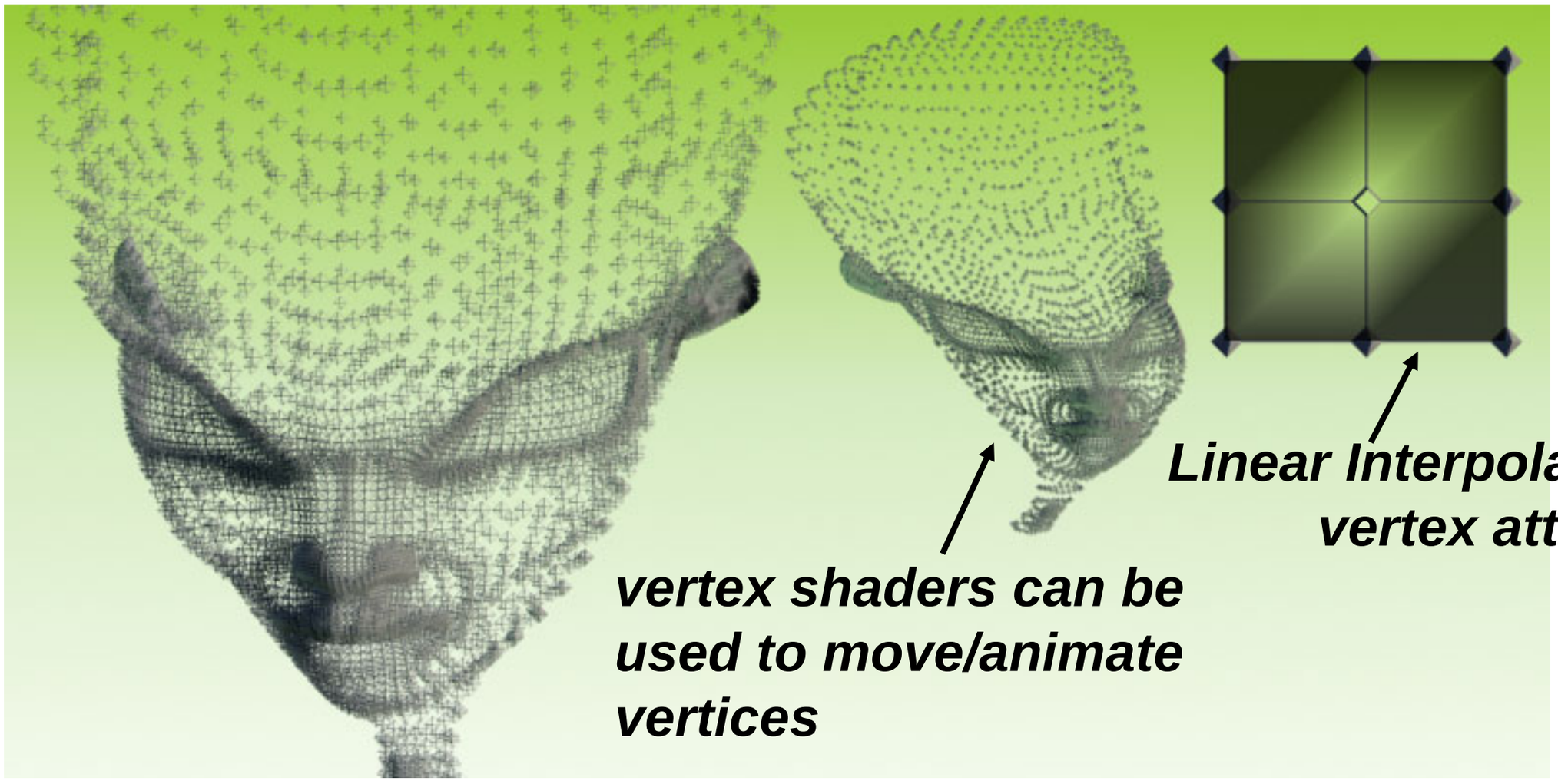
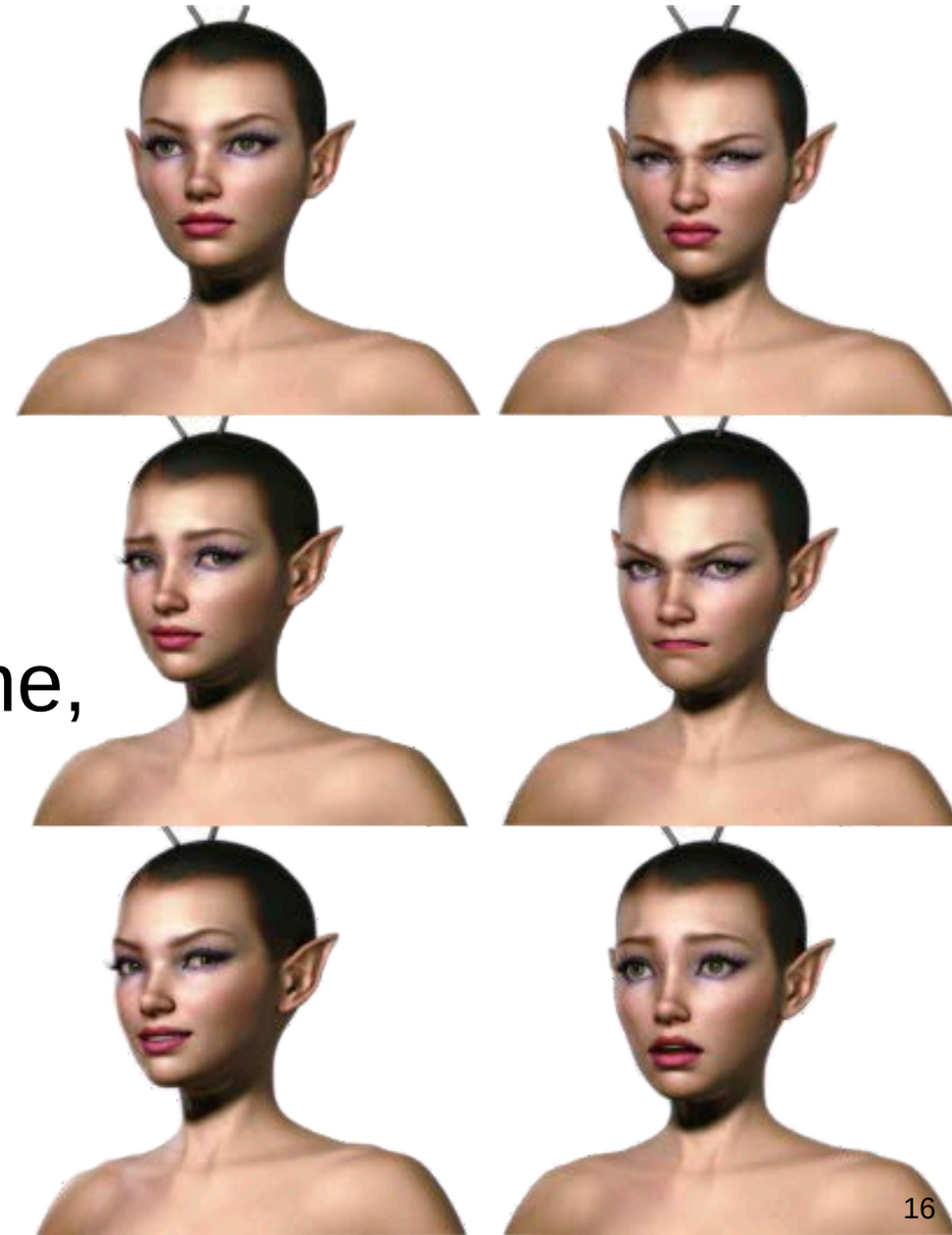


Figure from NVIDIA

Vertex Shader: Blendshapes

- 50 face geometries
  - angry, happy, sad, move eyebrow, …
- Each target stored as difference vector
  - For each vertex:
    average position + 50 differences
- Result is a weighted sum of all targets
  - Only transmit new weights each frame, the targets remain in GPU memory
  - Big multiply-add
    - Per active blend target
    - Per attribute



Vertex Shader: Skinning (SSD)

1. Transform each vertex $\mathbf{p}_i$ with each bone as if it was rigidly tied to it
2. Blend the results using bone weights

```
float4 skin(float4 restPos,
        uniform float4x4 xform[N_BONES],
        uniform float weight[N_BONES])
{
float4 outPos = float4(0,0,0,0)

for (int b = 0; b < N_BONES; b++) {
    outPos += weight[b]*mul(xform[b], restPos)
}

return outPos
}
```

Other jobs for vertex shaders

- **Prepare data** for **pixel shaders**
  (e.g. texture coordinates)
  - Stored/computed at vertices
  - Linearly interpolated per pixel
- Modern graphics hardware provides tons of ***interpolants*** (32x4 floats)

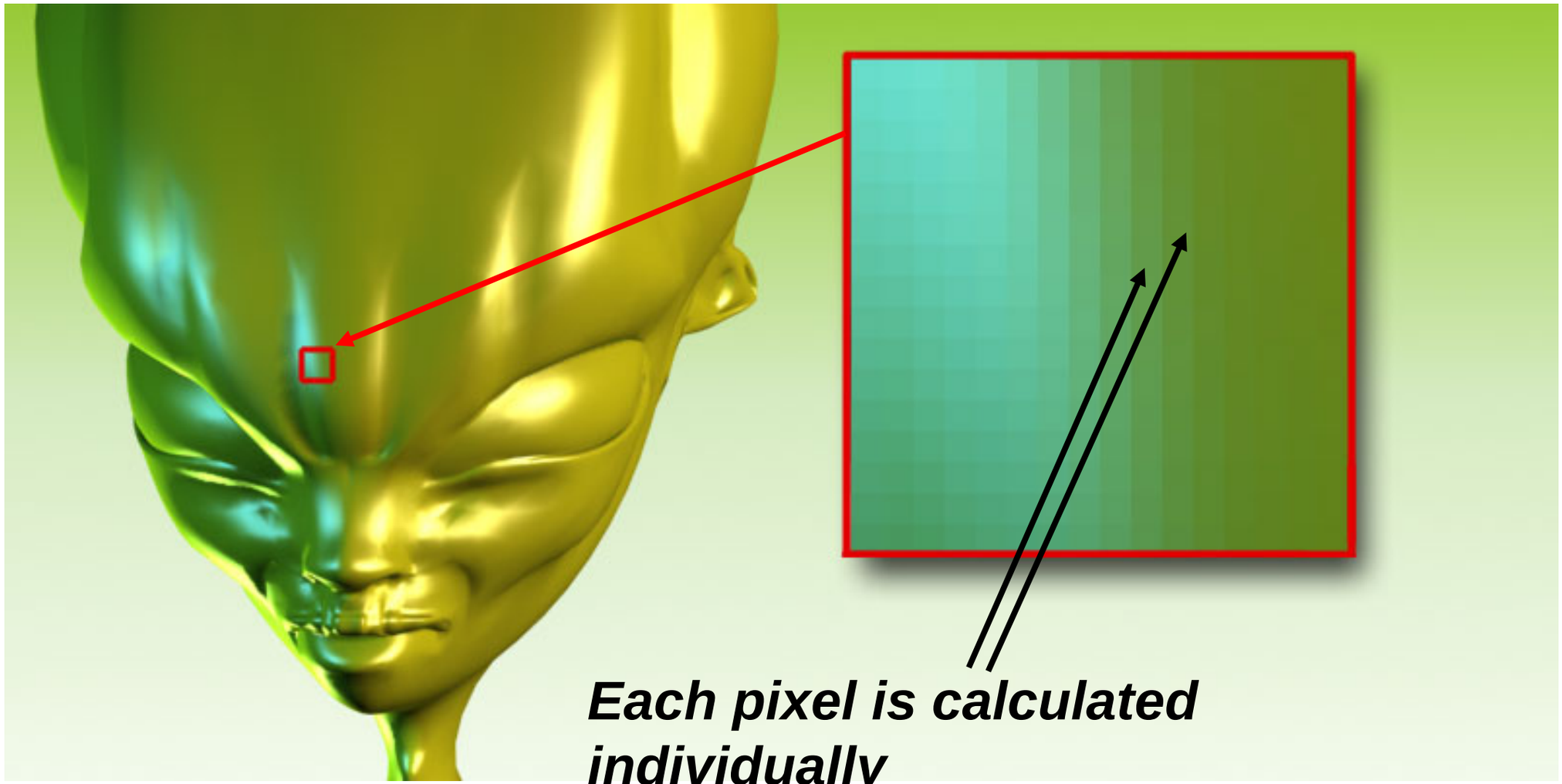
- **Project vertices** to the screen:

```
float4 transform(float4 worldPos,
                 uniform float4x4 modelViewProjection) {
return mul(modelViewProjection, worldPos)
}
```

Pixel Shaders

f(interpolants) ⟶ color, [depth]

purely functional (no side-effects)



Pixel shaders have little or no knowledge of neighboring pixels

Figure from NVIDIA

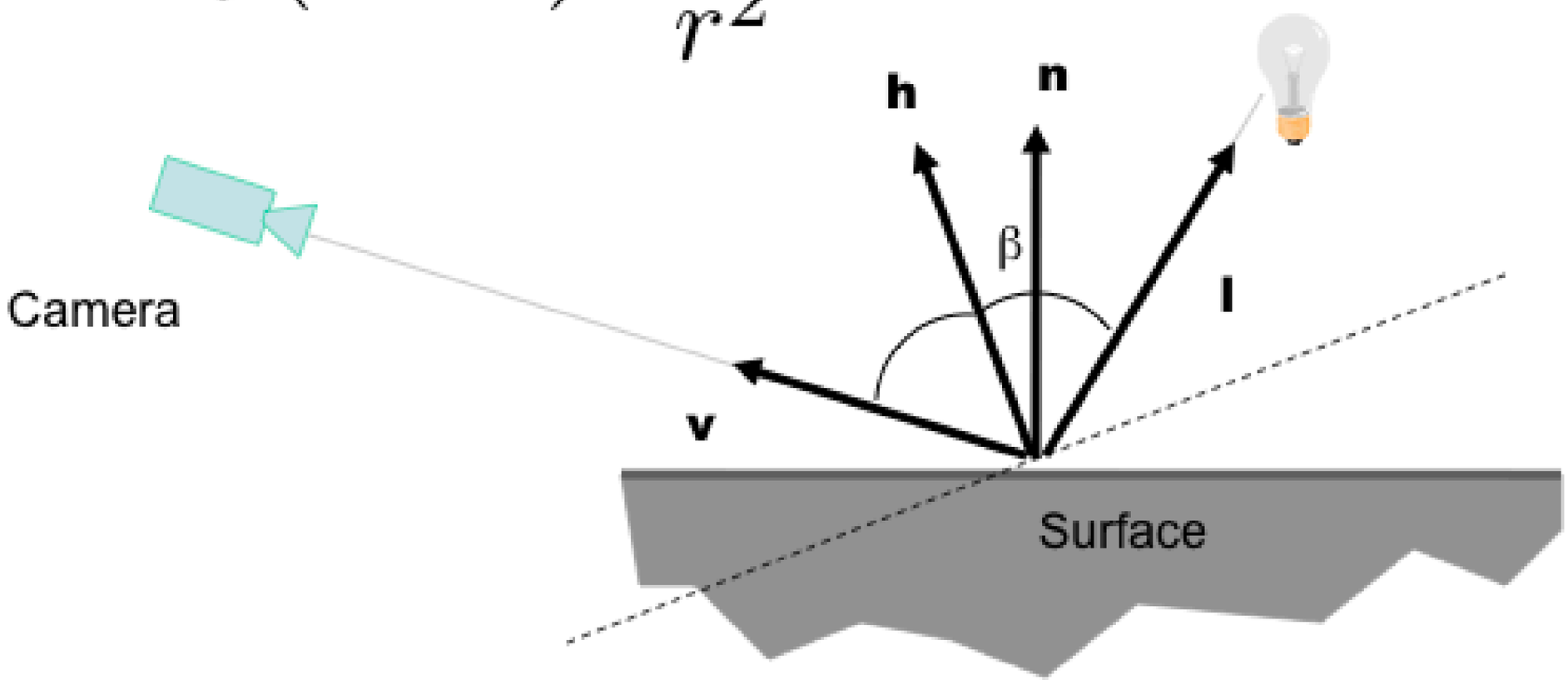
Pixel Shader: Blinn-Torrance Phong

Uses the "halfway vector" **h** between **l** and **v**.

$$L_o = k_s \cos(\beta)^q \frac{L_i}{r^2}$$

$$= k_s (\boldsymbol{n} \cdot \boldsymbol{h})^q \frac{L_i}{r^2}$$

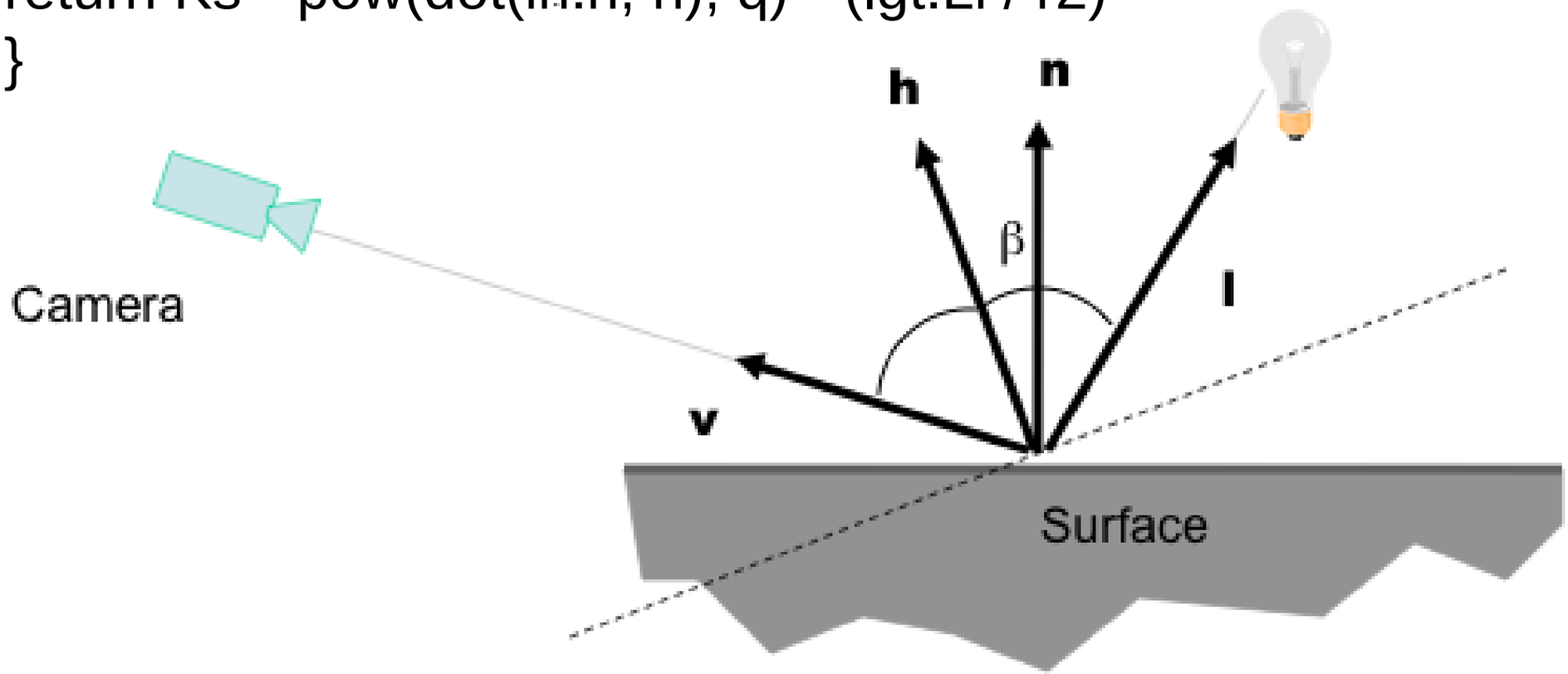
$$\boldsymbol{h} = \frac{\boldsymbol{l} + \boldsymbol{v}}{\|\boldsymbol{l} + \boldsymbol{v}\|}$$



Pixel Shader: Blinn-Torrance Phong

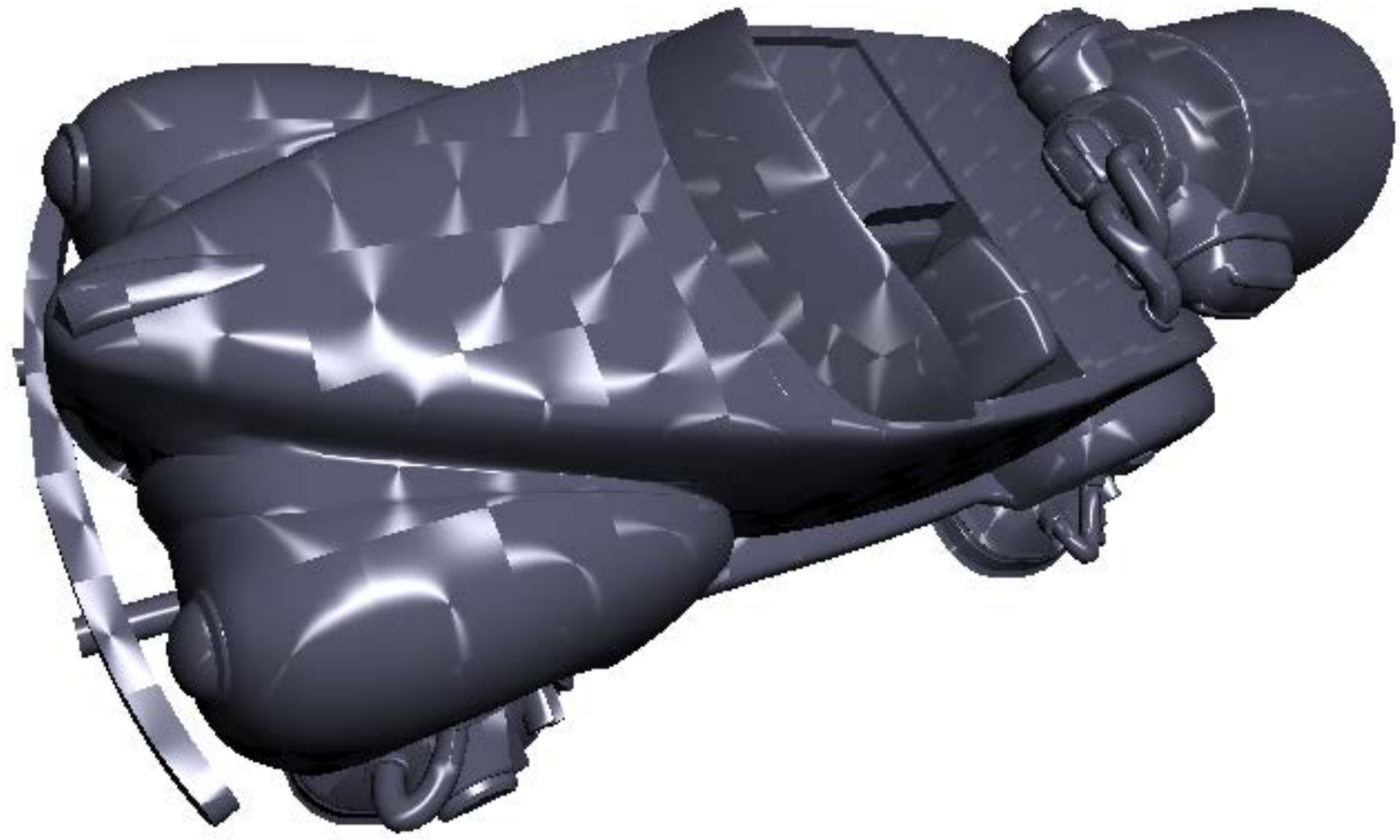
```
struct interpolants { float4 p, n, v }
struct light { float4 pos, float Li }

float4 phong(interpolants in, uniform light lgt,
             uniform float q, uniform float Ks)
{
float4 l = lgt.pos - in.p
float r2 = length(l)*length(l)
float4 h = normalize(l+in.v)    // useful built-ins
return Ks * pow(dot(in.n, h), q) * (lgt.Li / r2)
}
```



Brushed Metal

- Procedural texture
- Anisotropic lighting

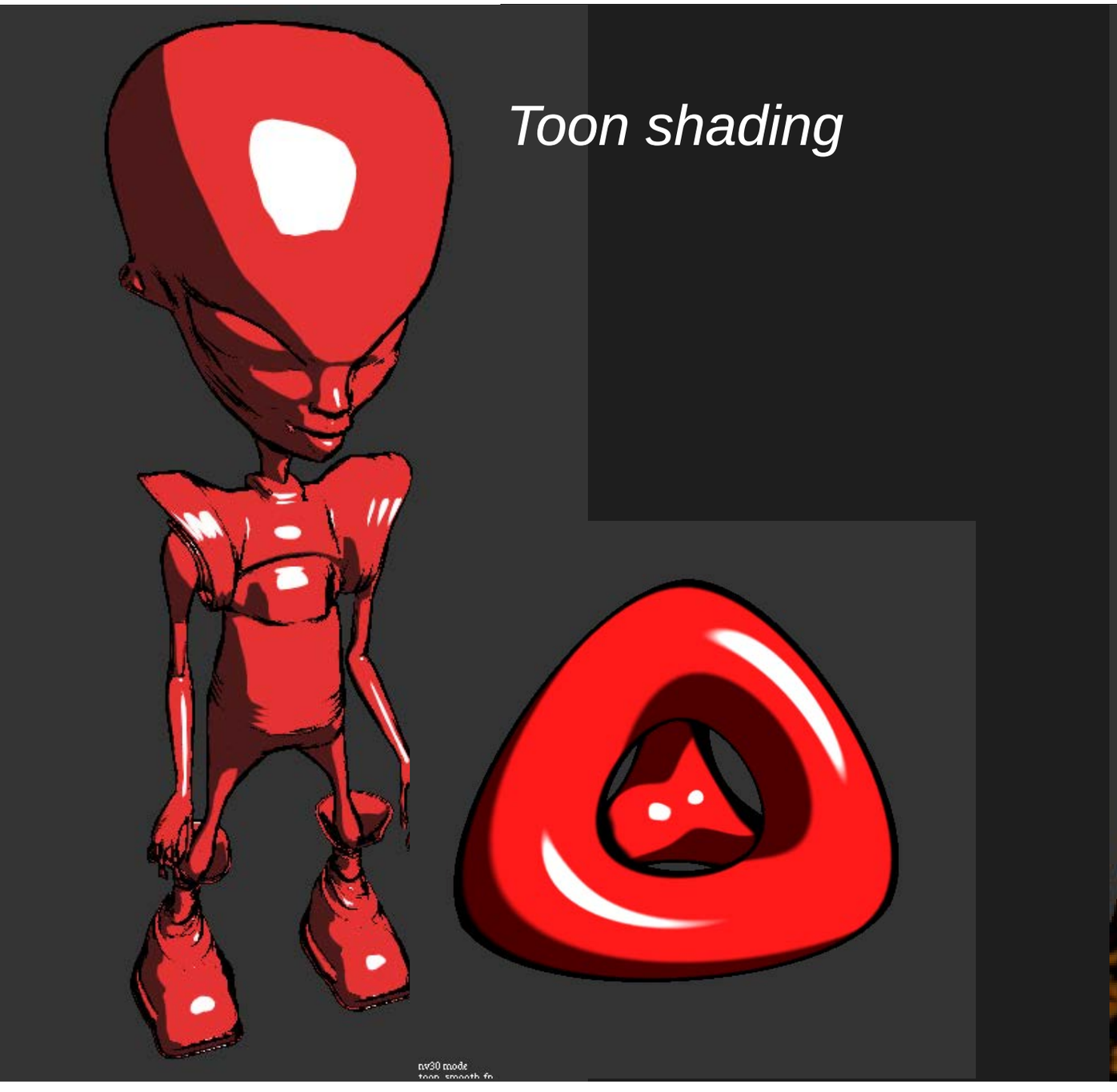


Melting Ice

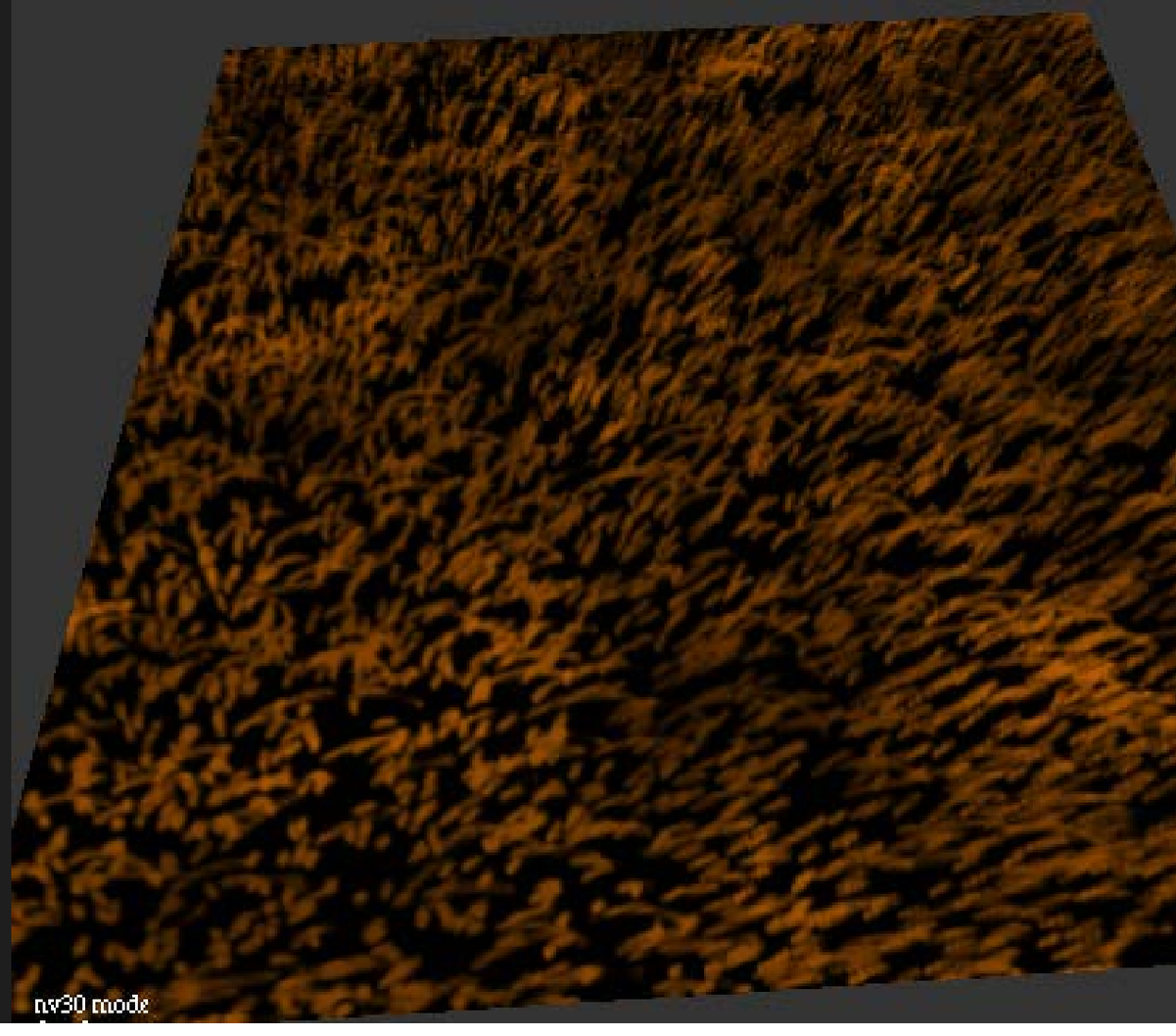
- Procedural, animating texture
- Bumped environment map

Toon & Fur

Toon shading



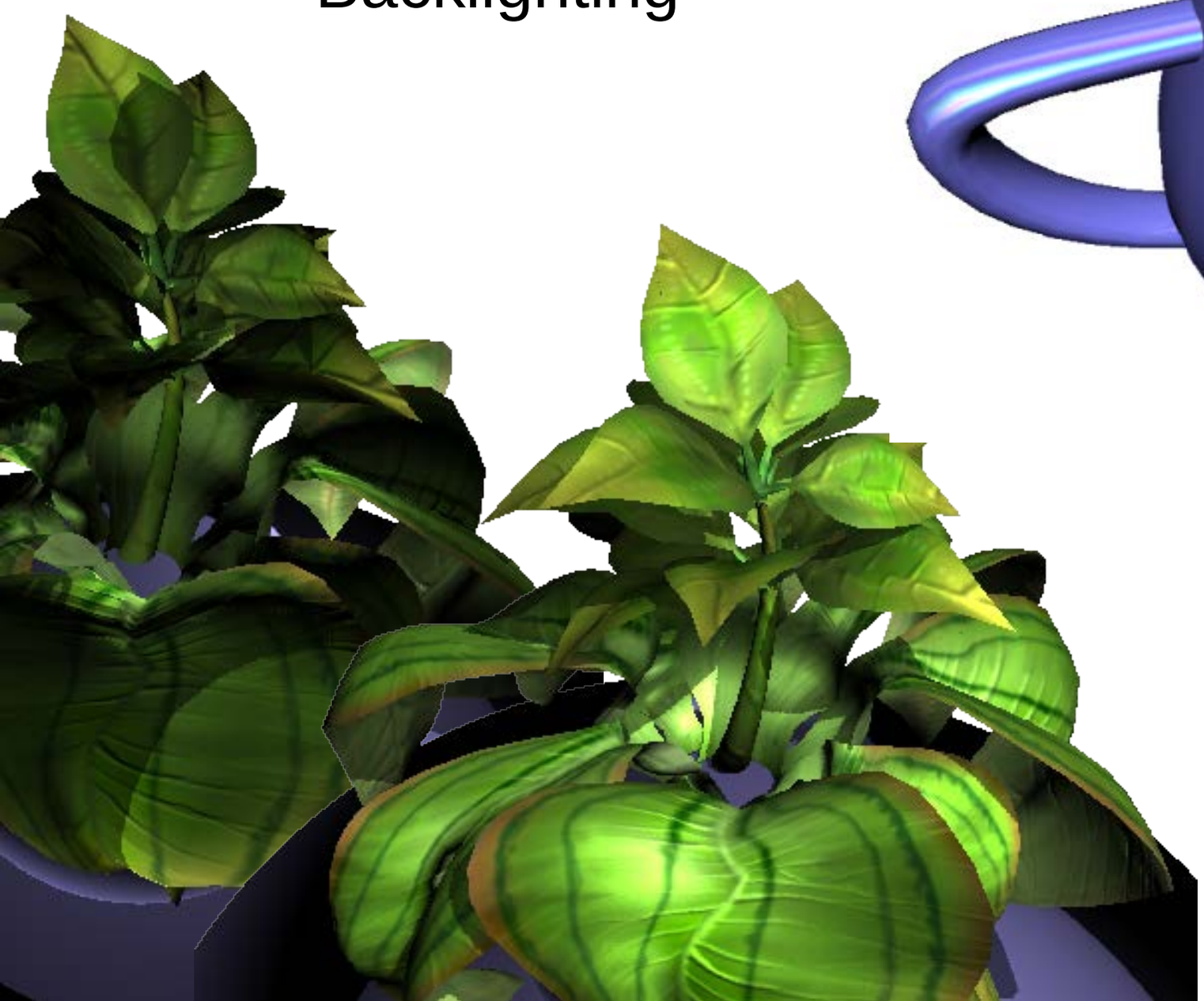
Volumetric fur



Vegetation & Thin Film

Translucence

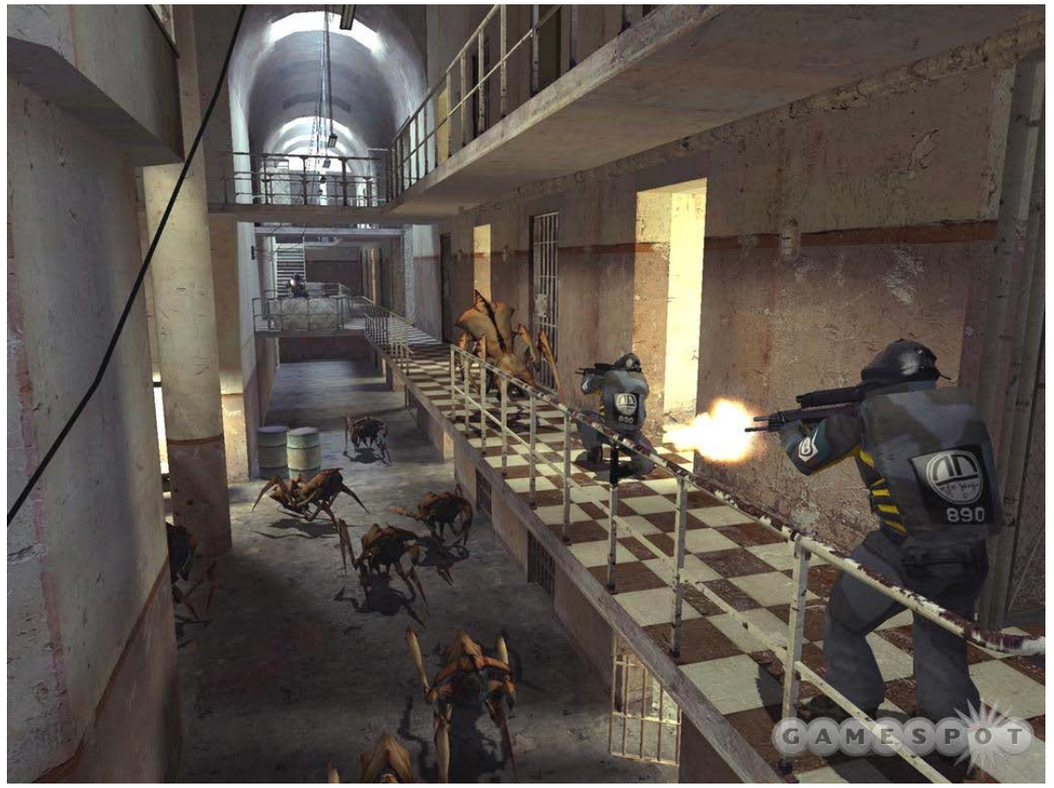
Backlighting



Custom lighting

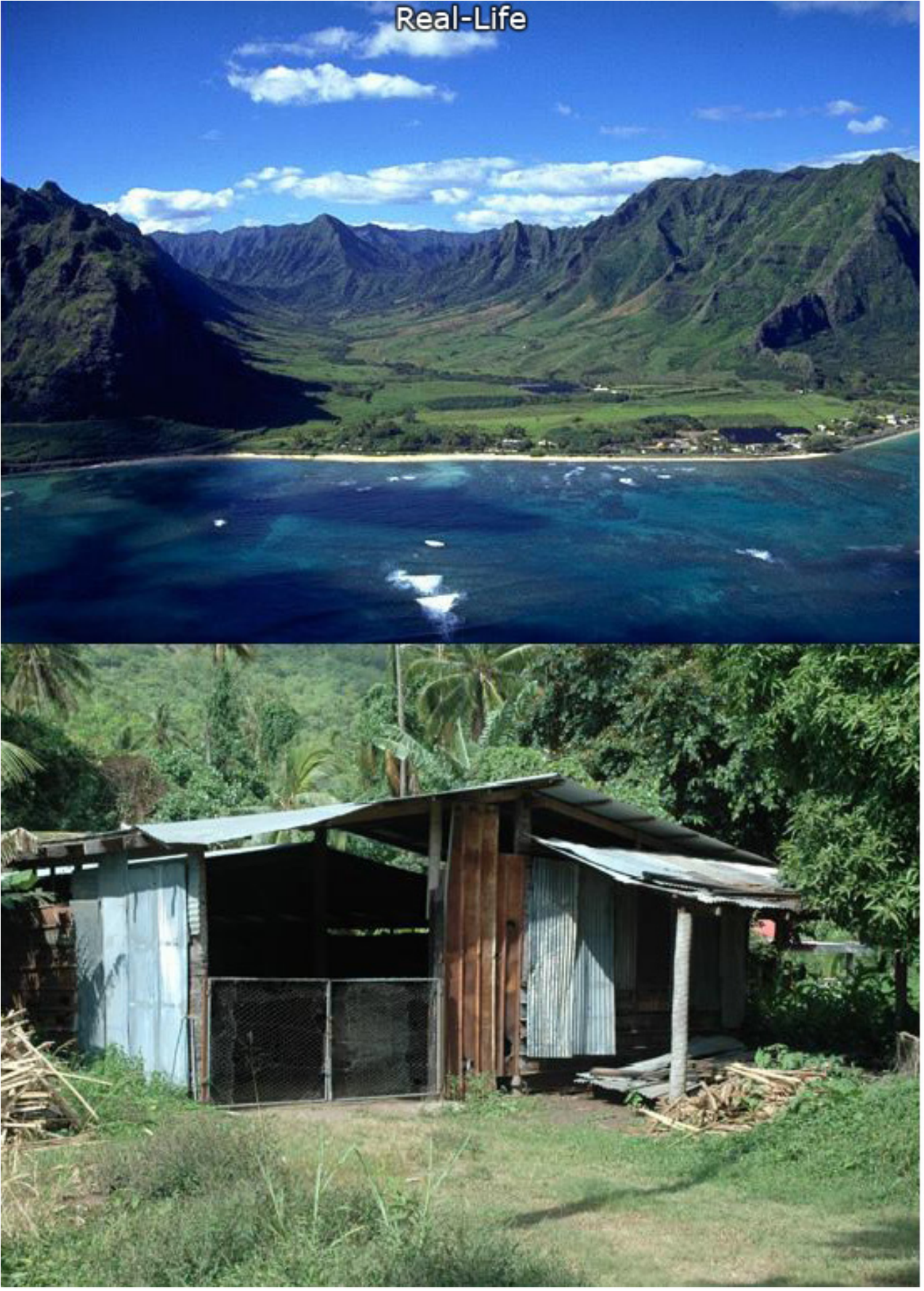
Simulates iridescence

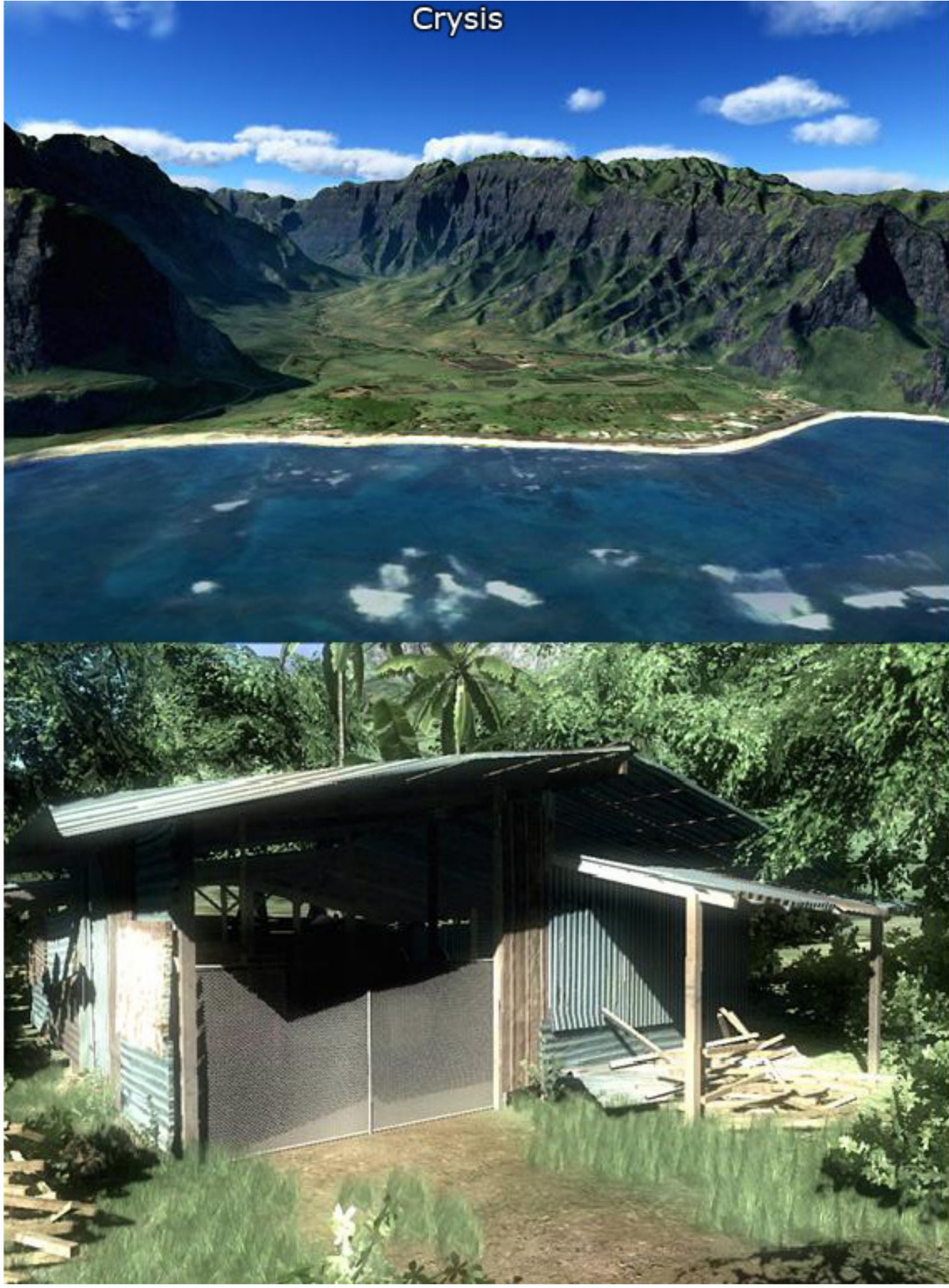
Allows for amazing quality



Real-Life

Crysis





Earth, 4.5x10^9 B.C.E.

Crysis, 2007

Rich scene appearance

- Vertex shader
  - Geometry (skinning, displacement)
  - Setup interpolants for pixel shaders
- Pixel shader
  - Visual appearance
  - Also used for image processing, GPGPU abuses
- Multi-pass
  - Render the scene or part of the geometry multiple times
  - Different viewpoints (e.g. shadow map, shadow volume)
  - More complex shading (e.g. pass per-light)
  - Image processing (e.g. HDR tone mapping, bloom)

Questions?

How to program shaders?

- Assembly code, or Higher-level language and compiler (e.g. HLSL, GLSL, Cg)

- Write separate **vertex** and **fragment** programs

- Send to the GPU via **graphics API** (Direct3D, OpenGL) (much like a texture or other piece of state)

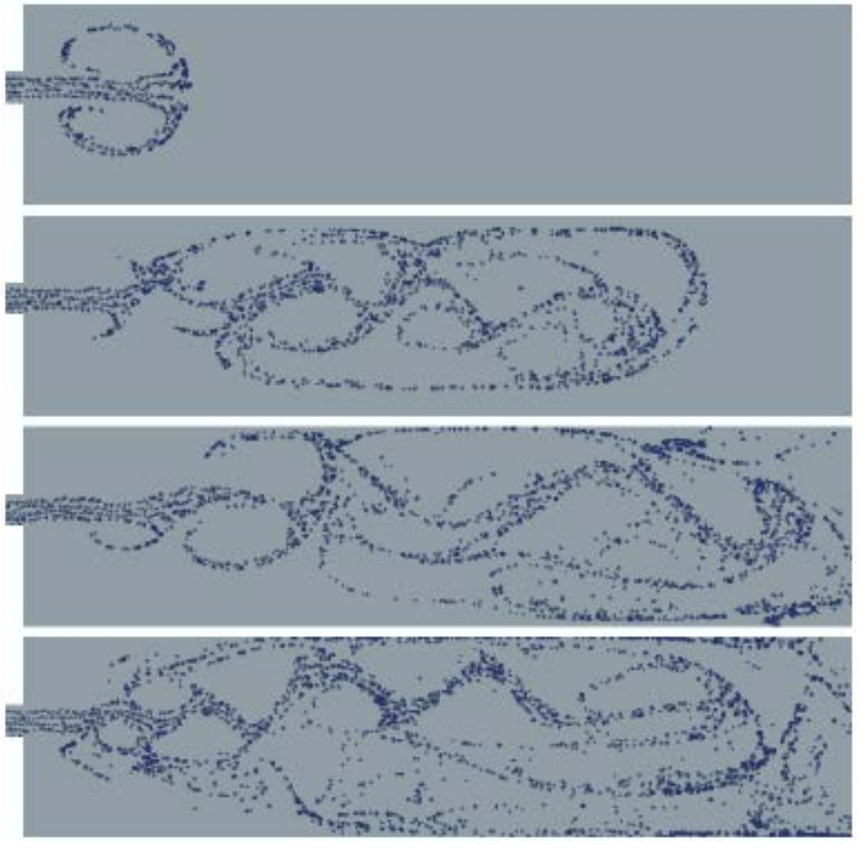
- (Heavily) optimized at runtime by the driver’s **JIT compiler**

General-purpose computation on GPUs

- Dense data-parallel processor (TeraFLOPS)
- Increasingly programmable (Code executed for each vertex or each pixel)
- C & C-like languages

- Use “shaders” for general-purpose computation

- Peak performance difficult to reach
- Parallelism = hard



Navier-Stokes on GPU [Bolz et al.]

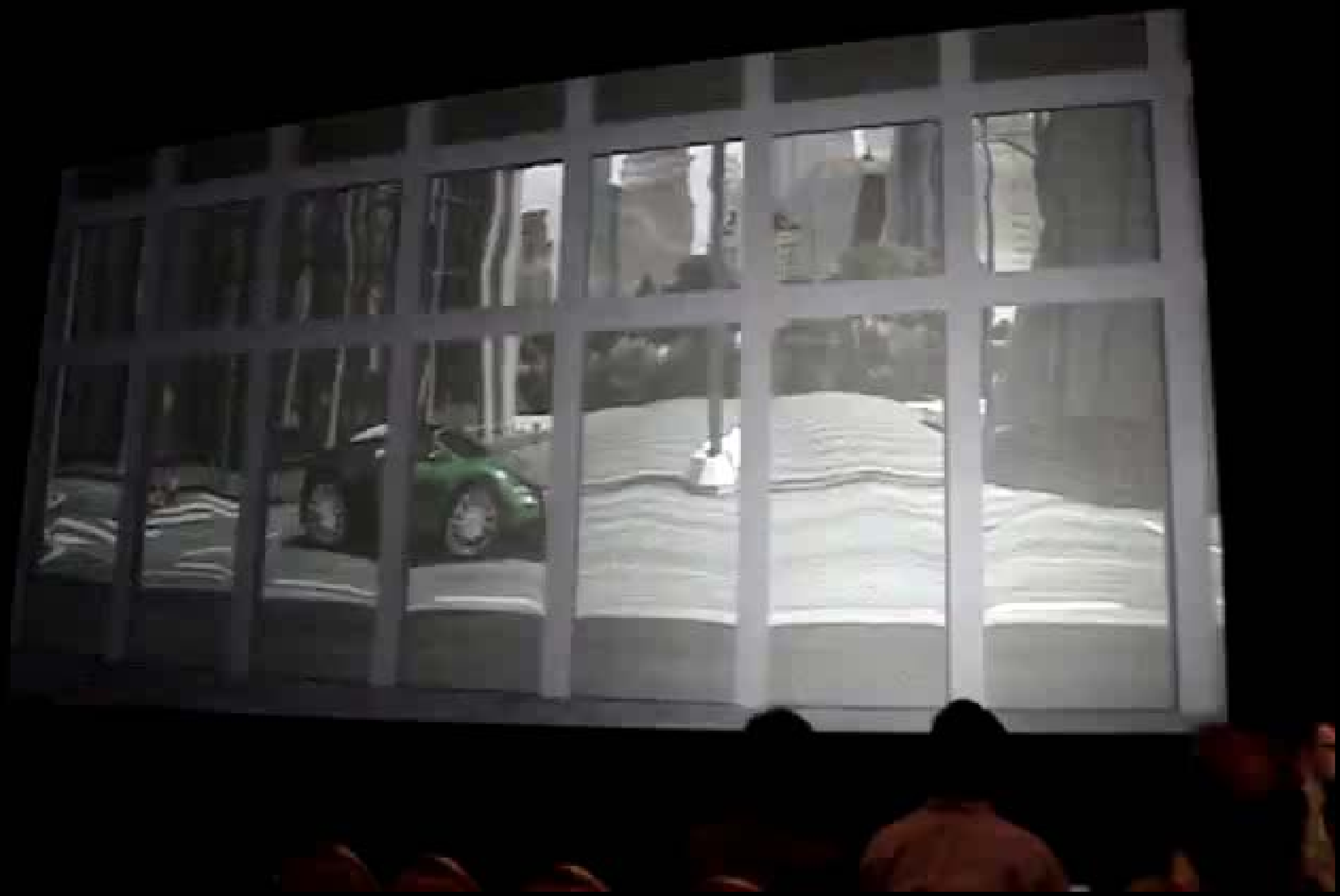
Real-Time Gravity Demo (NVIDIA)

Ray Tracing as GPGPU

- cool video!

NVIDIA

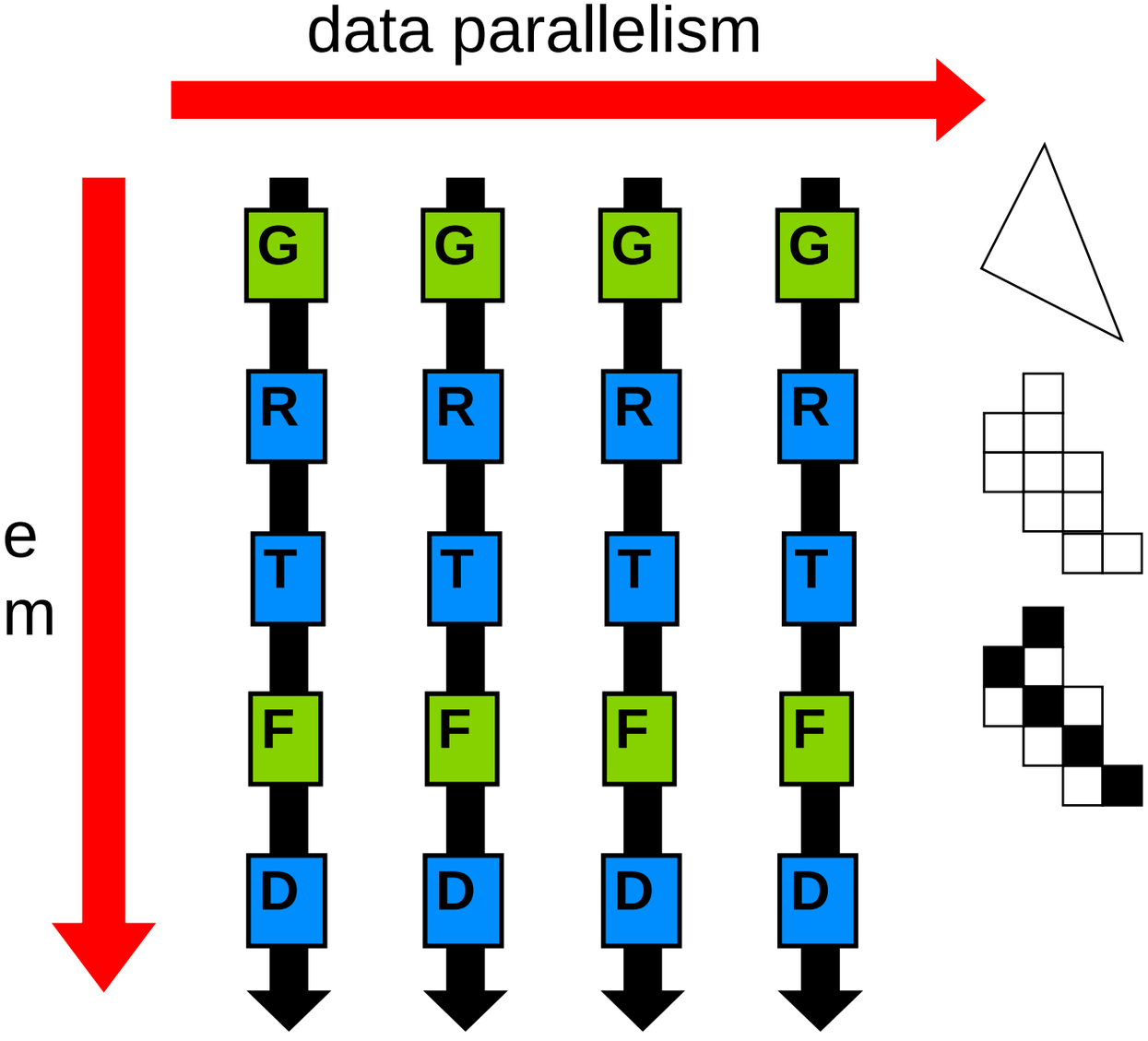
Ray Tracing as GPGPU



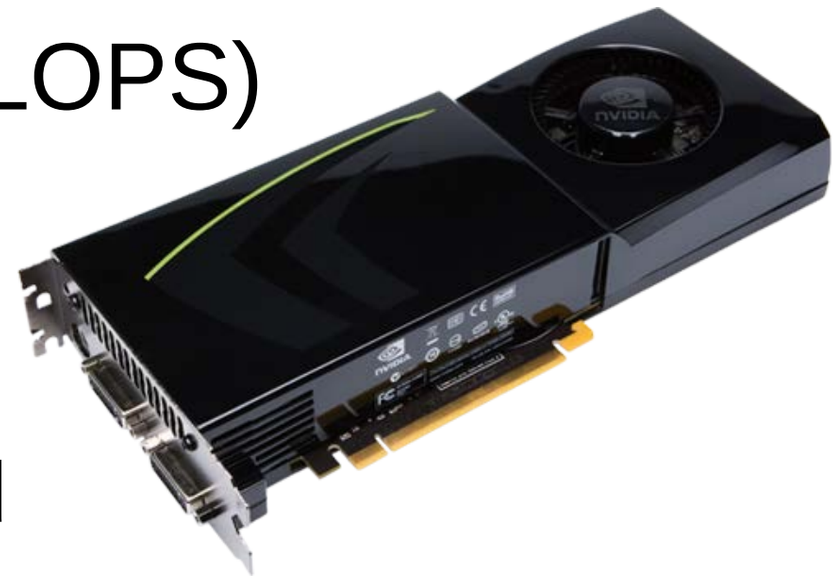
Questions?

Graphics Hardware

- High performance through:
  - Parallelism
  - Specialization
  - Limited synchronization (minimal data dependency)
  - Efficient latency hiding (pre-fetching, threading)



Modern Graphics Hardware



- 100s of parallel shader pipelines (TFLOPS)
- Deep pipeline (~1k stages)
- Hierarchical tiling of screen (~4x4)
  - Early Z rejection if entire tile is occluded
- Pixels processed as quads (2x2 pixel neighborhoods)
  - Allows *derivatives* in pixel shader (for filtering)
- Massive multithreading to hide texture latency
  - And smart memory layout
  - Throughput (bandwidth)-optimized memory controllers (at the expense of latency)

Why is graphics hardware so fast?

GPU: throughput

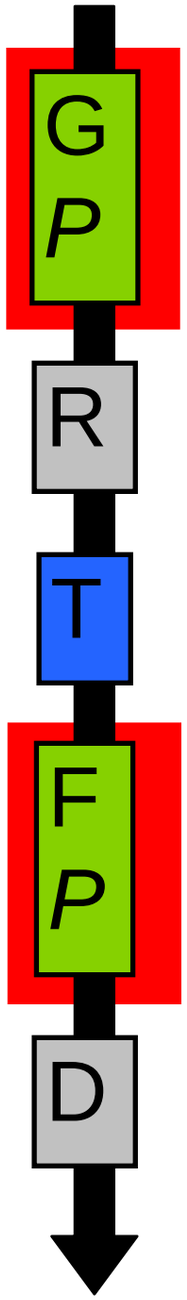
- Massive, explicit **parallelism**
- More transistors do **compute**, less cache, bookkeeping
- **Specialization** (rasterizer, texture filtering)
- **High** arithmetic intensity (math ops » memory ops)
- Deep pipeline, latency hiding, prefetching
- Little data dependence
- Controlled memory access patterns

CPU: latency

- Predominantly **scalar**
- Most transistors are **bookkeeping, cache**
- Highly **general-purpose** (x86 ISA)
- **Low** arithmetic intensity (math ops ≈ memory ops)
- Short pipeline, latency intolerance, out-of-order exec
- Arbitrary data dependence
- Arbitrary, dynamic memory access patterns

Questions?

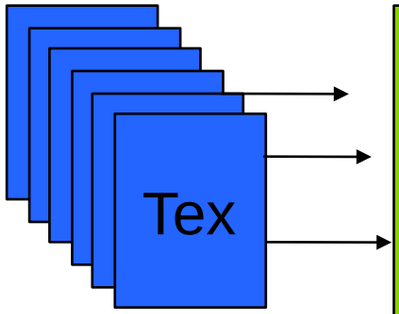
Traditional hardware architecture



vertex units

one big parallel rasterizer

mipmap filtering

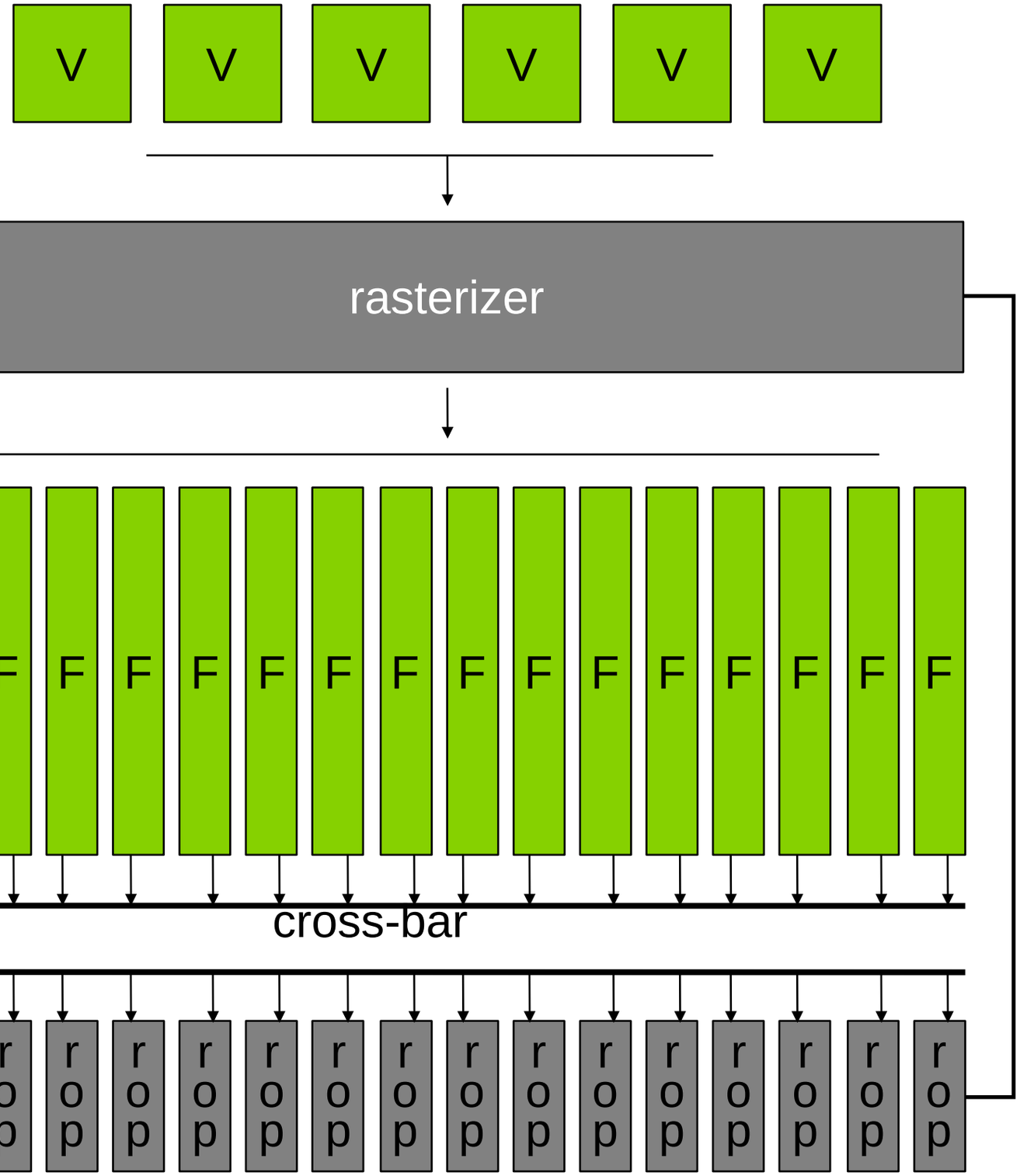


fragment units

raster operation units

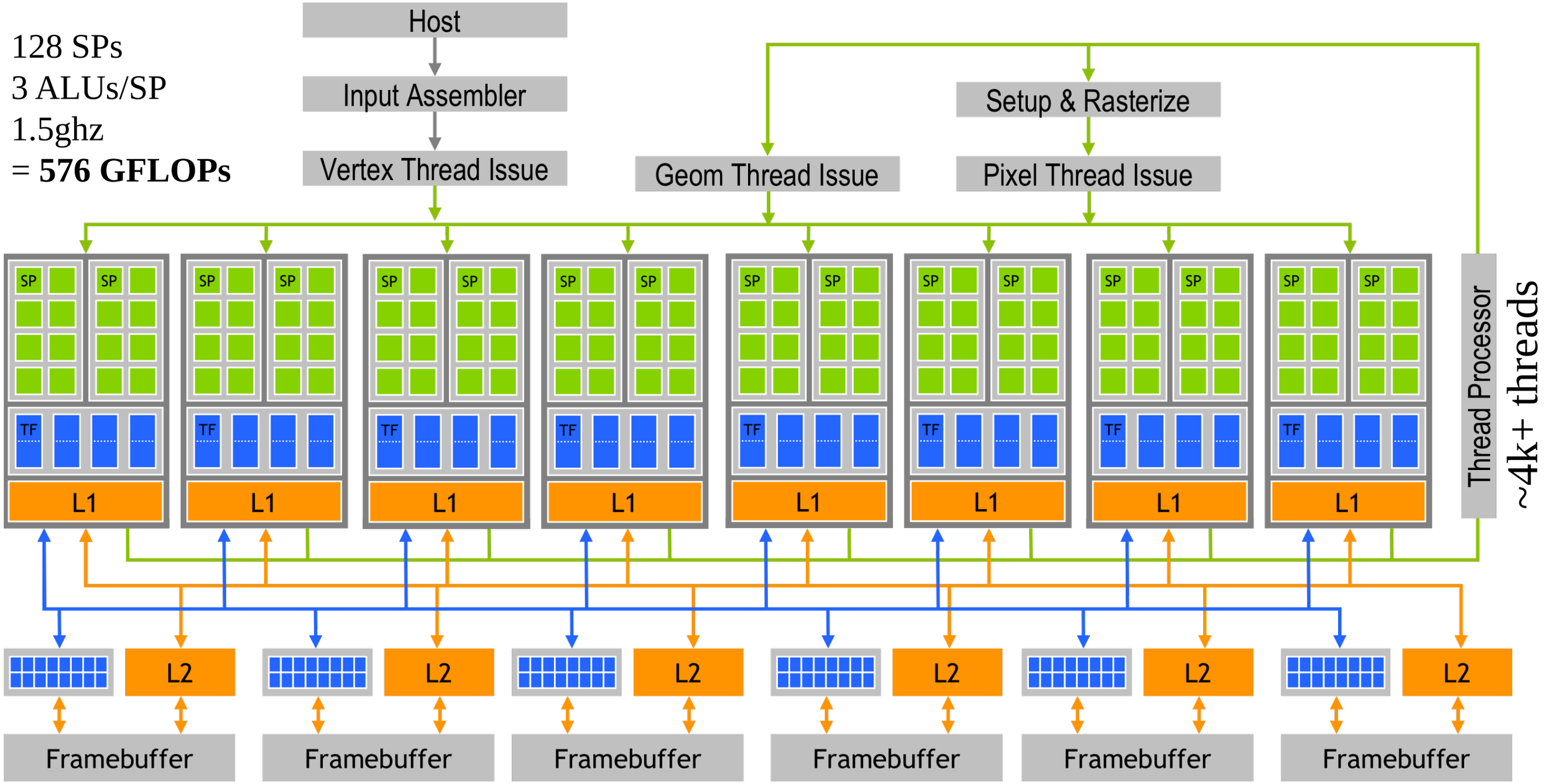
z-buffer, framebuffer

screen-locked



Unified shading architecture (NVIDIA G80)

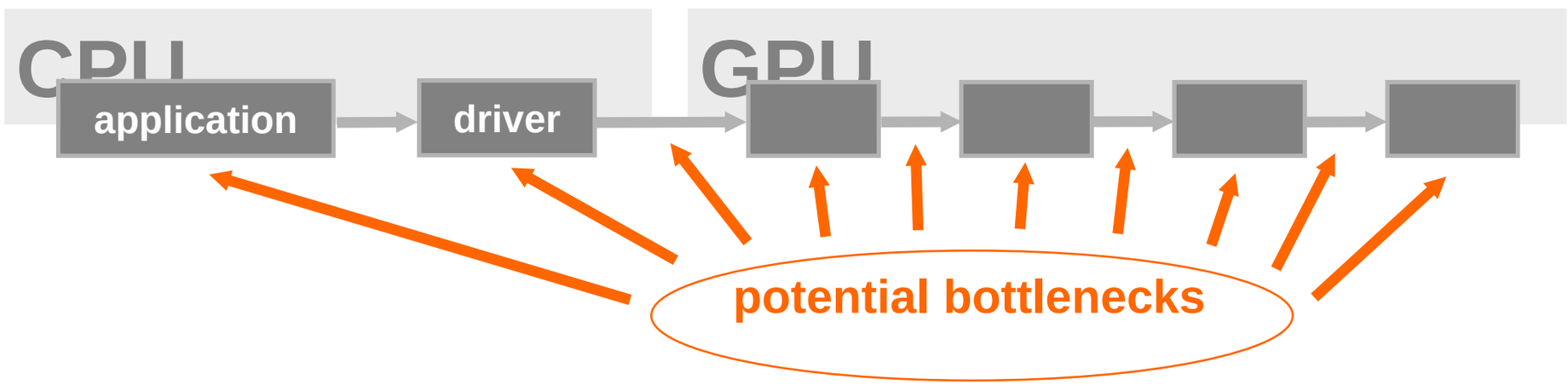
128 SPs
3 ALUs/SP
1.5ghz
= **576 GFLOPs**



85 GB/sec

Questions?

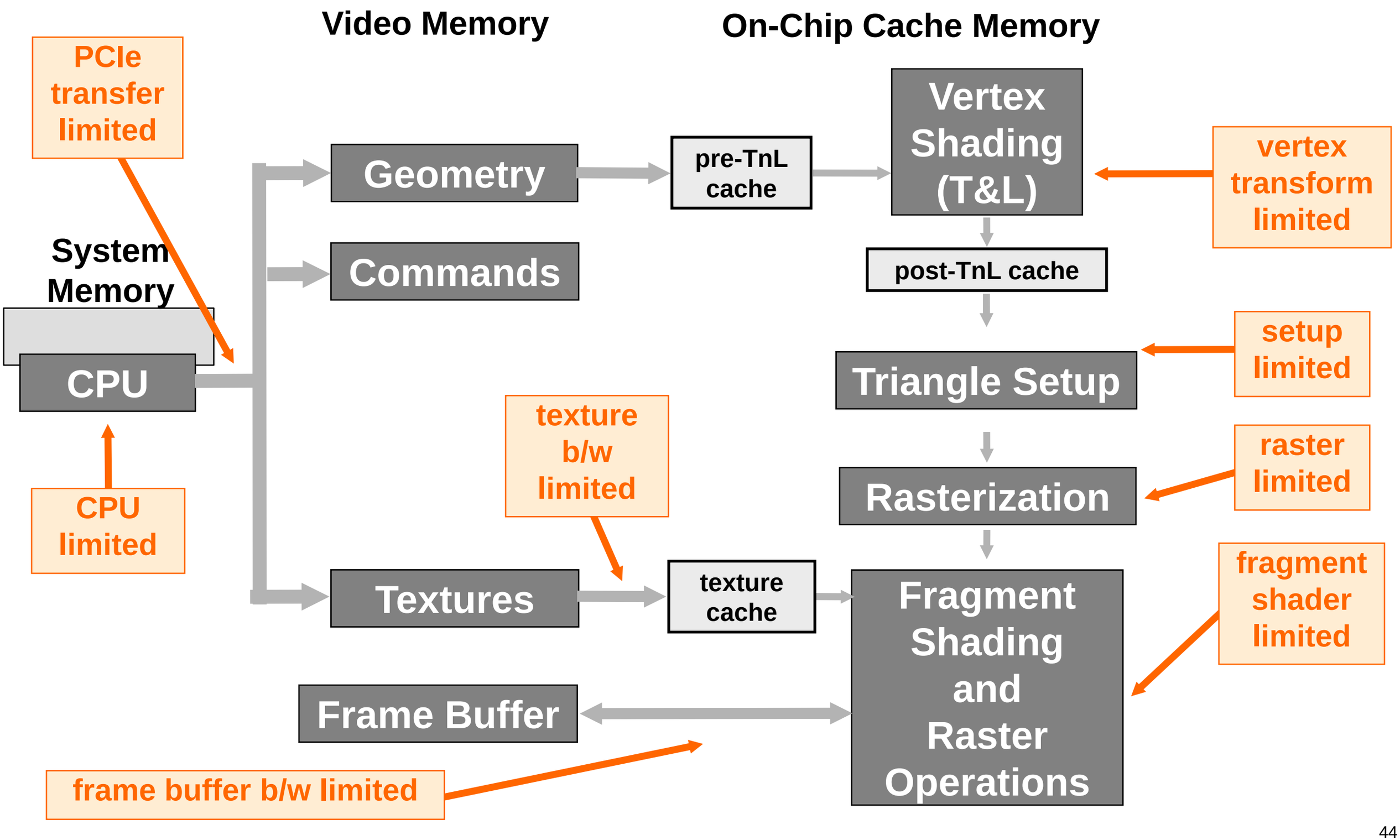
Bottlenecks?



- The bottleneck determines overall throughput
- In general, the bottleneck varies over the course of an application and even over a frame
- Getting good performance is all about finding and eliminating bottlenecks in the pipeline

Slide from NVIDIA

Potential Bottlenecks



Rendering pipeline bottlenecks

- The term “transform/vertex/geometry bound” often means the bottleneck is “anywhere before the rasterizer”

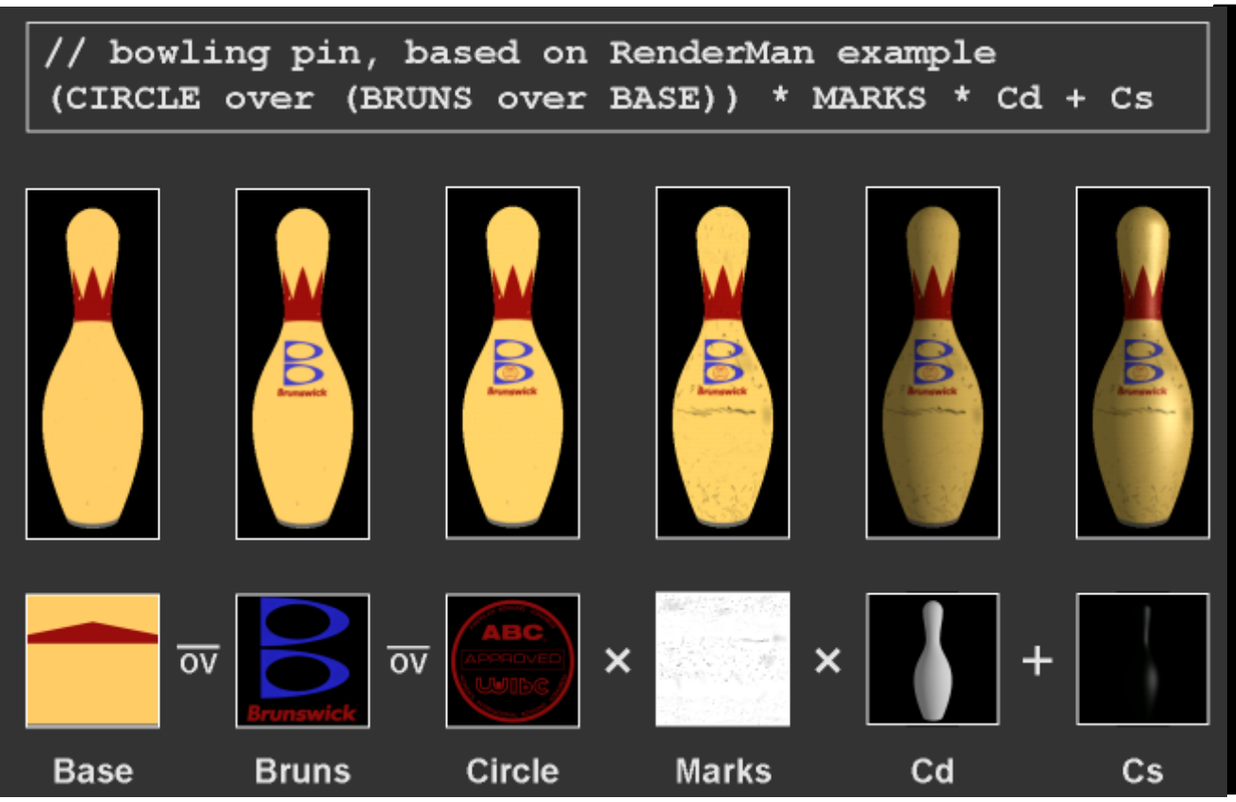
- The term “fill/raster bound” often means the bottleneck is “anywhere after setup for rasterization” (computation of edge equations)

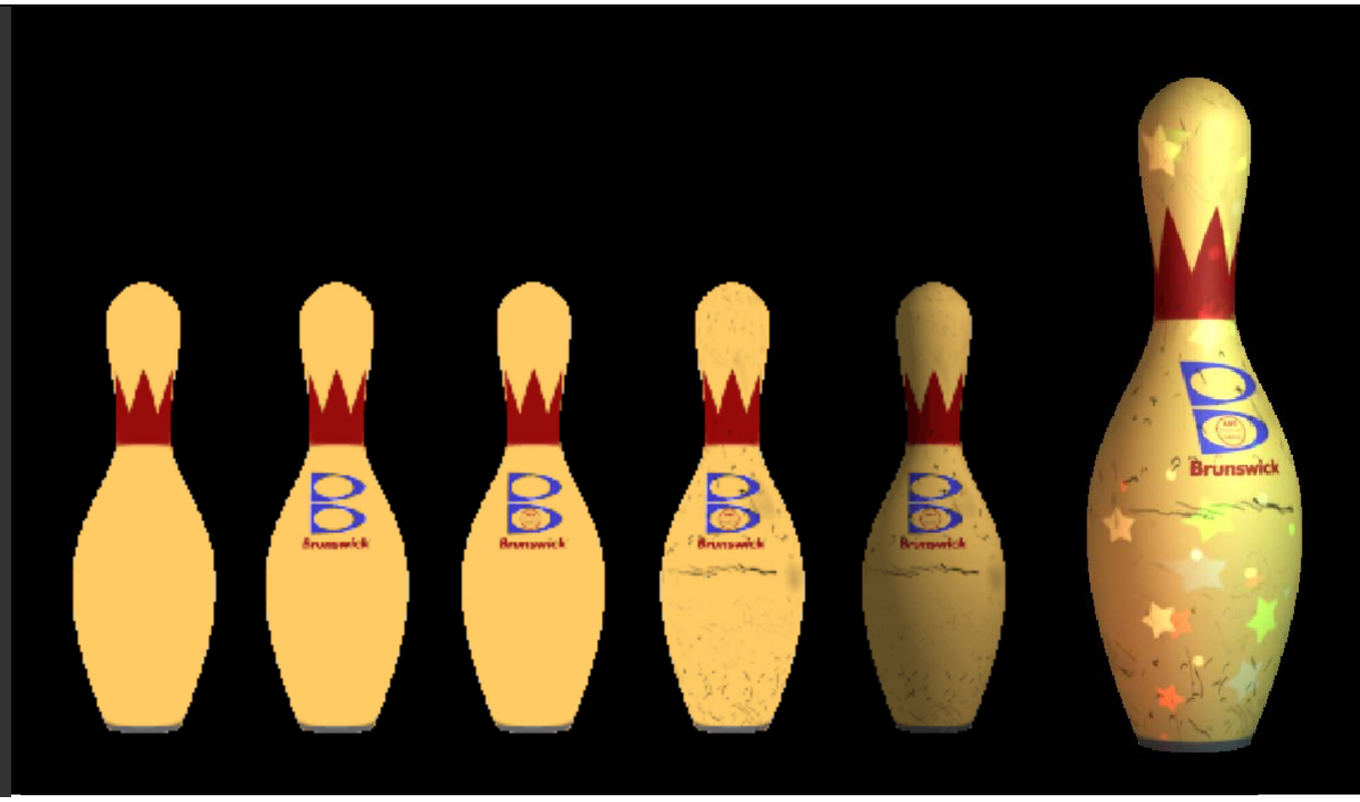
- Can be both transform and fill bound over the course of a single frame!

Questions?

Shader zoo

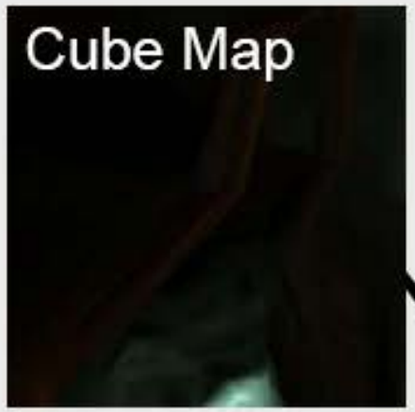
Layering

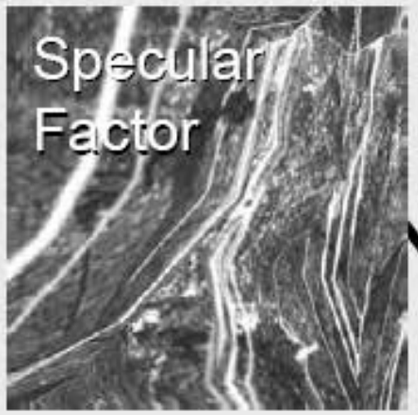

// bowling pin, based on RenderMan example
(CIRCLE over (BRUNS over BASE)) * MARKS * Cd + Cs
over
over
×
×
+
Base
Bruns
Circle
Marks
Cd
Cs

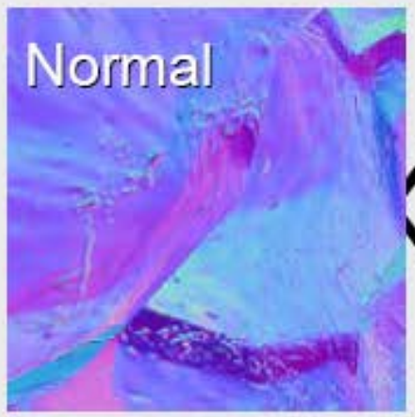


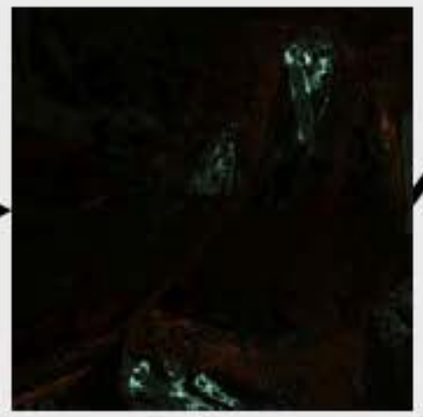
From Half Life 2 (Valve)

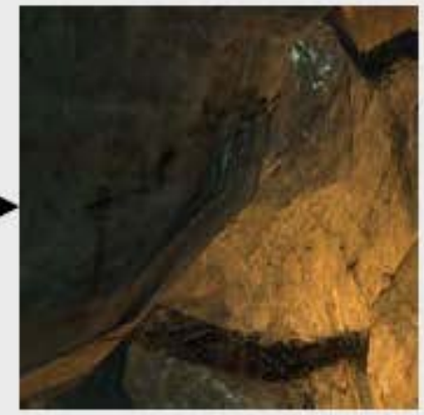
Radiosity Normal Mapping Shade Tree



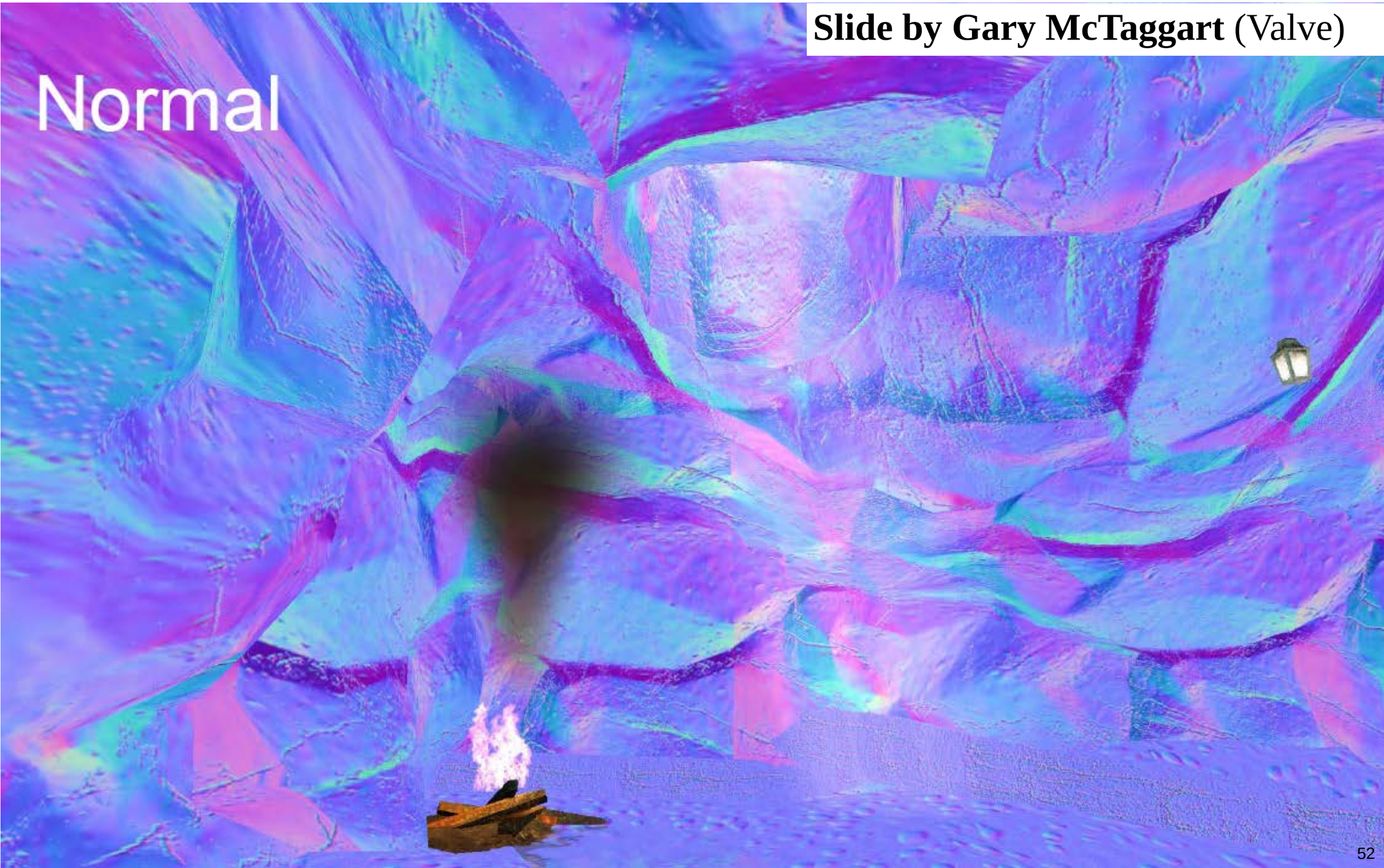






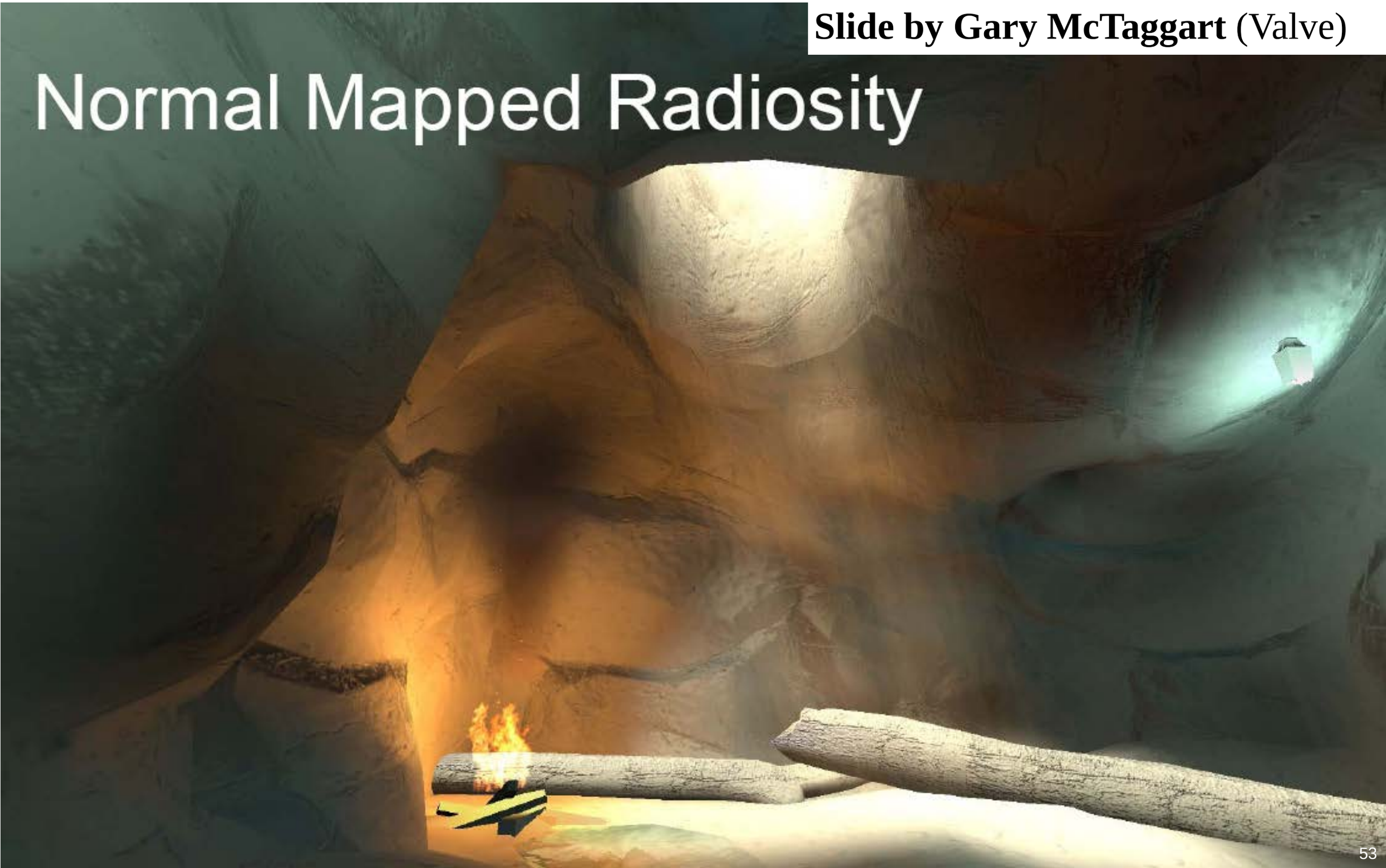


Slide by Gary McTaggart (Valve)



Slide by Gary McTaggart (Valve)

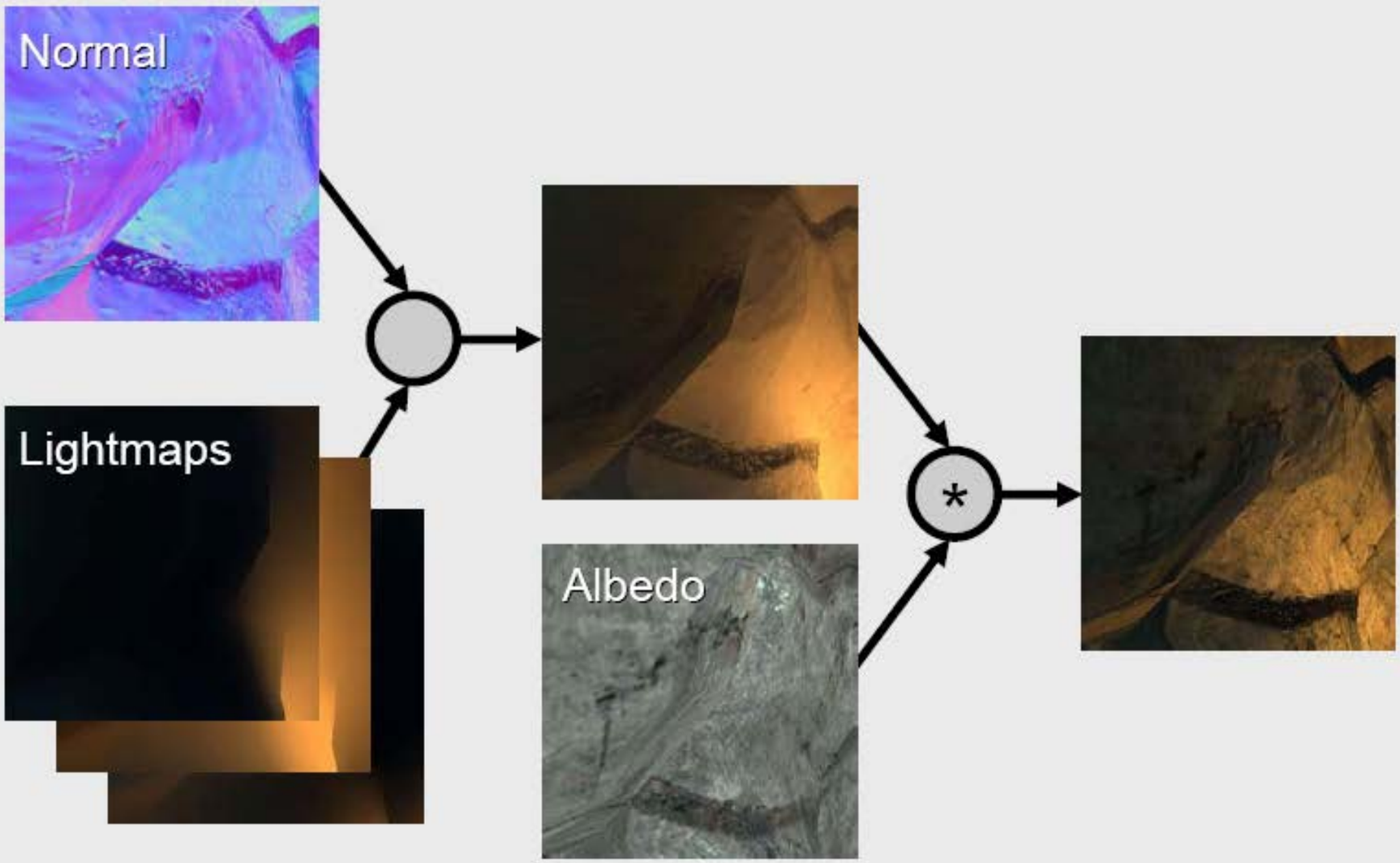
Slide by Gary McTaggart (Valve)



Slide by Gary McTaggart (Valve)

Slide by Gary McTaggart (Valve)

Radiosity Normal Mapping Shade Tree



Slide by Gary McTaggart (Valve)

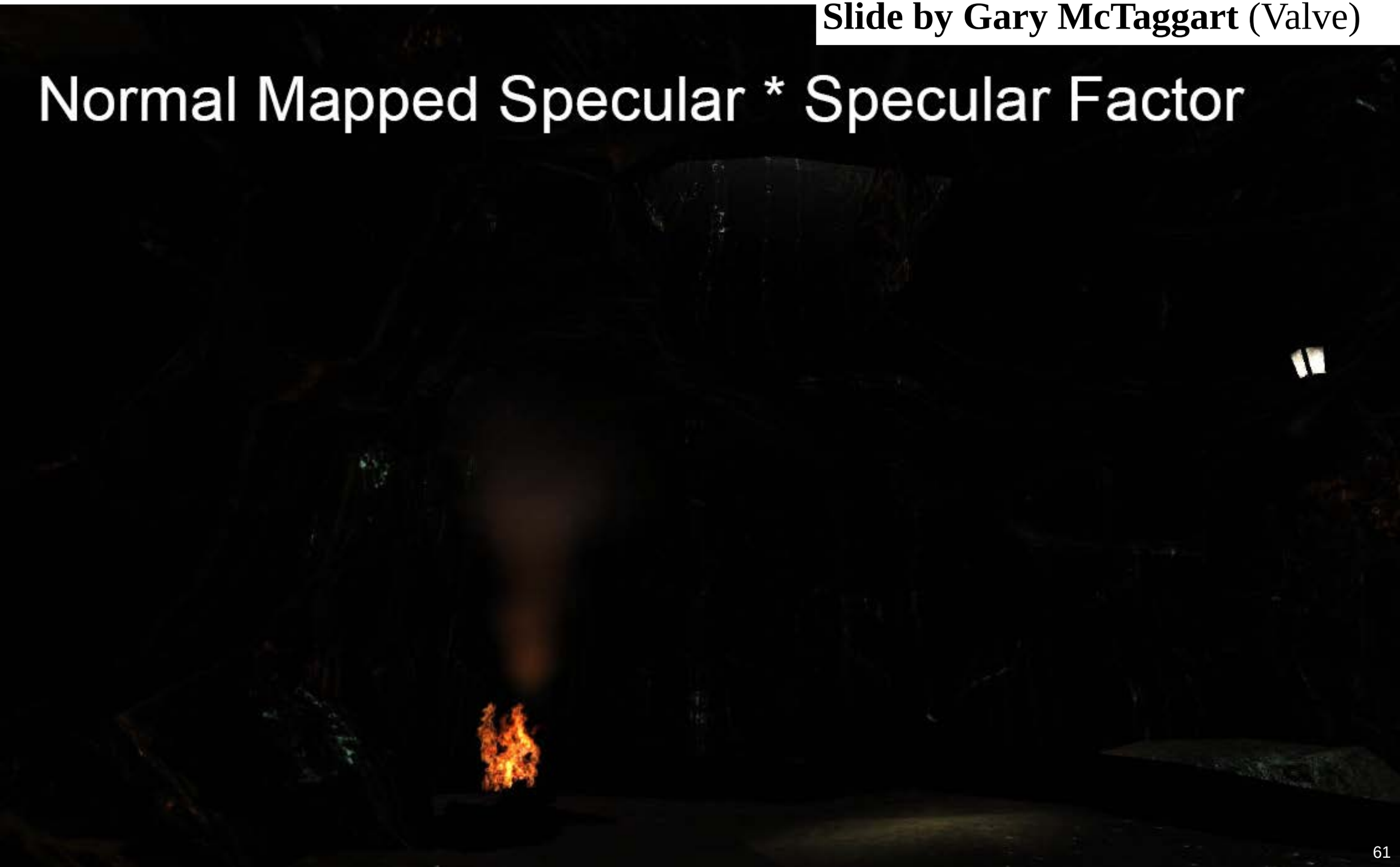
Cube Map Specular

Slide by Gary McTaggart (Valve)

Slide by Gary McTaggart (Valve)

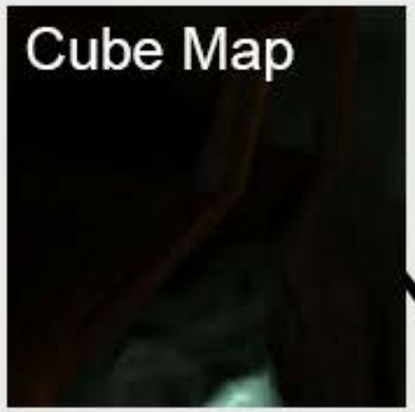
Slide by Gary McTaggart (Valve)

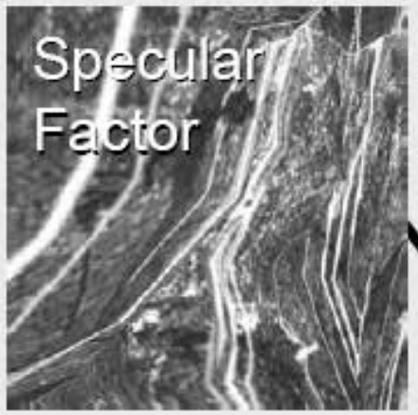
Slide by Gary McTaggart (Valve)

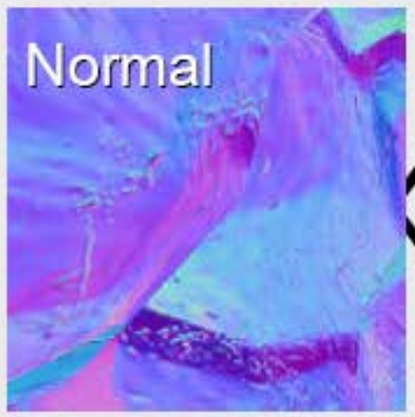


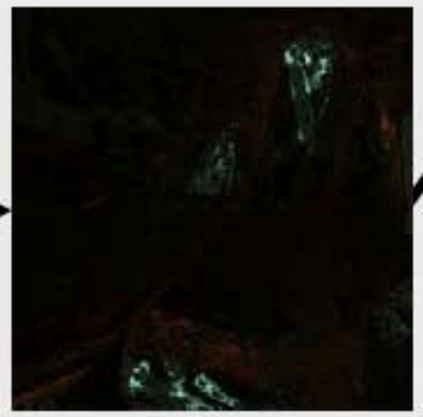
Slide by Gary McTaggart (Valve)

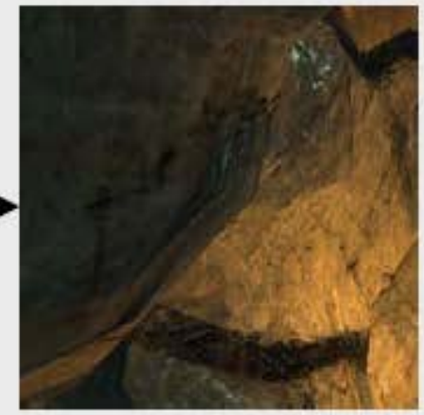
Radiosity Normal Mapping Shade Tree

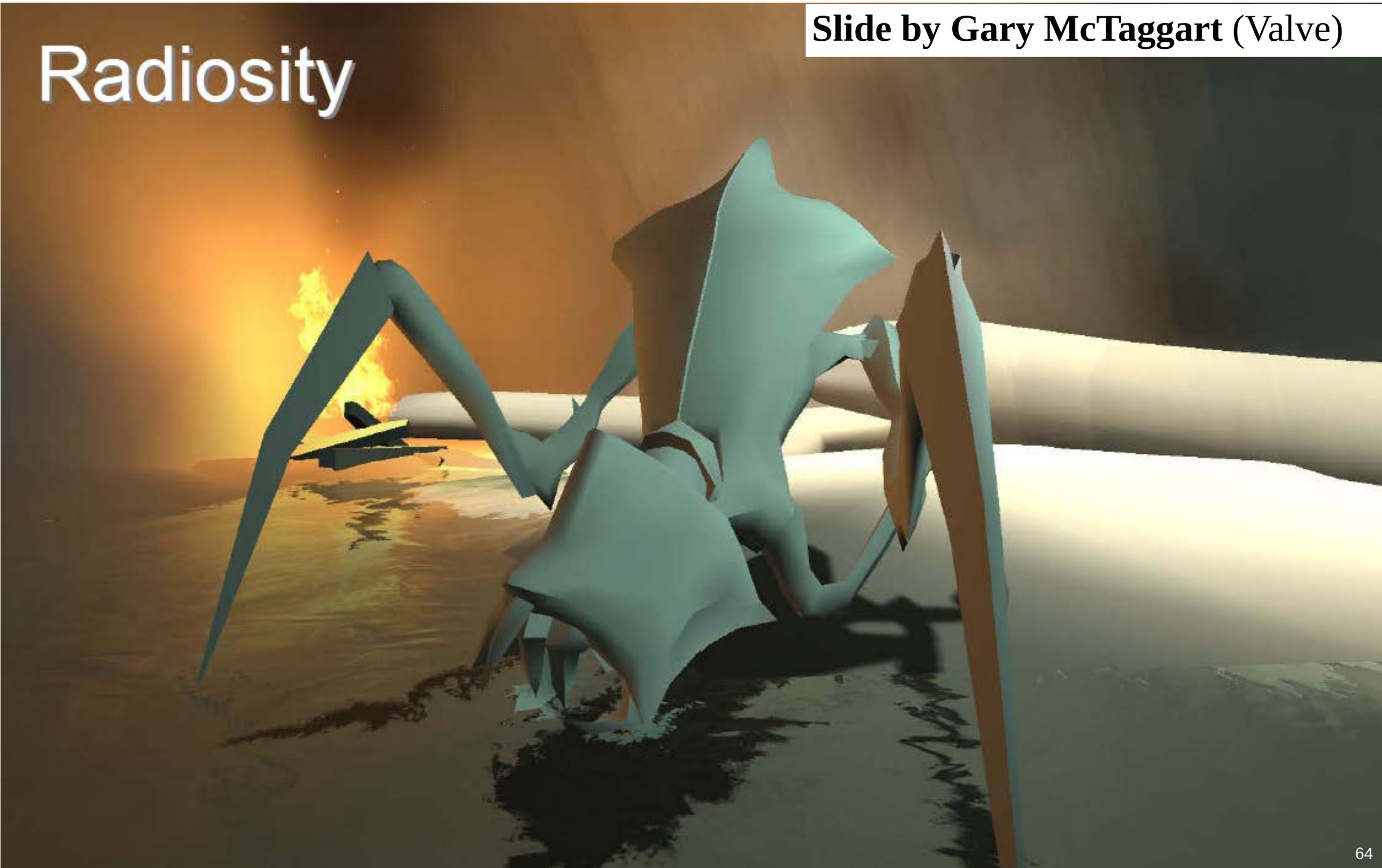




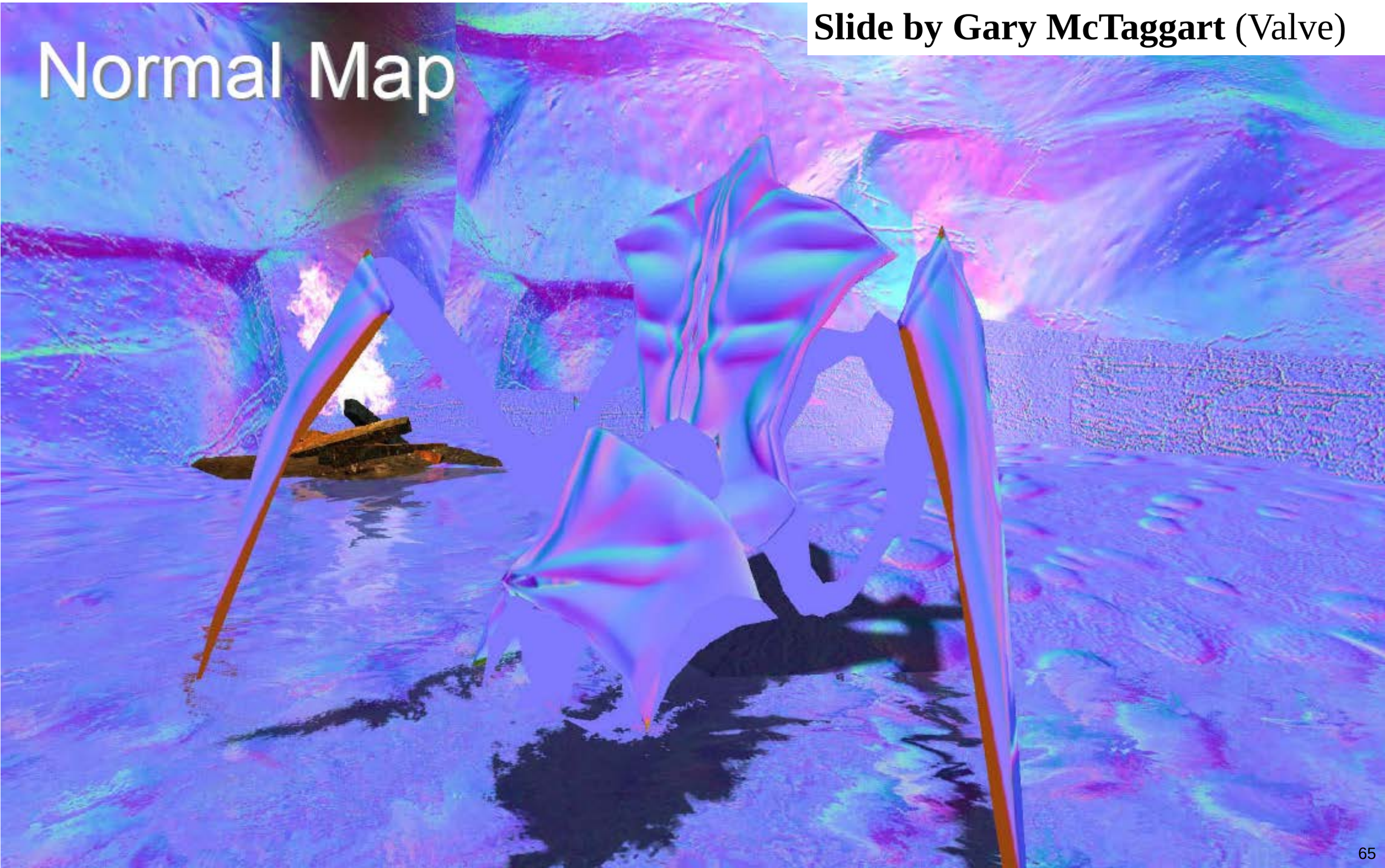








Slide by Gary McTaggart (Valve)



Slide by Gary McTaggart (Valve)

Slide by Gary McTaggart (Valve)

Slide by Gary McTaggart (Valve)

Slide by Gary McTaggart (Valve)

Slide by Gary McTaggart (Valve)

Slide by Gary McTaggart (Valve)

Slide by Gary McTaggart (Valve)

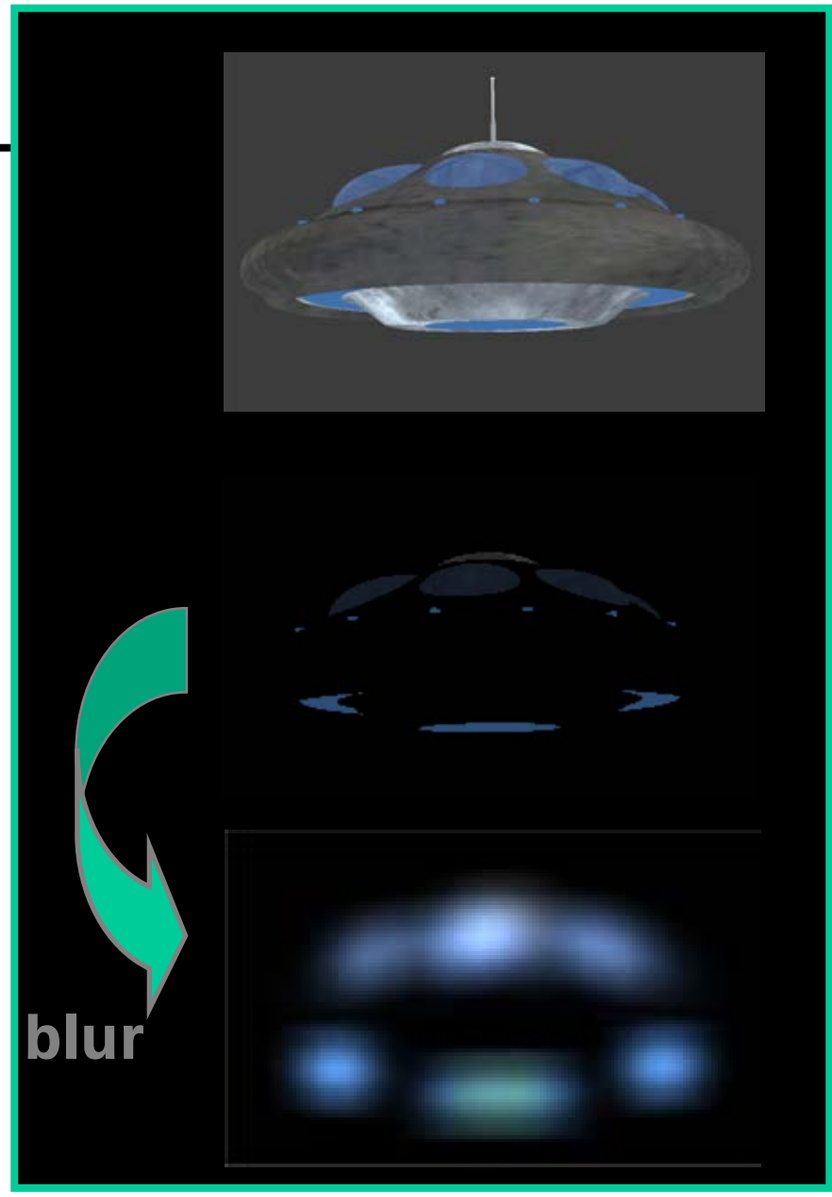
Slide by Gary McTaggart (Valve)

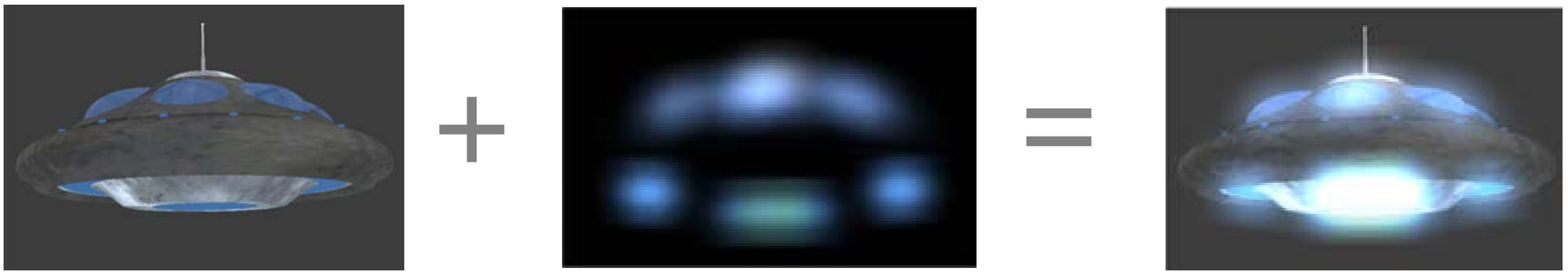
Refraction mapping (multipass)

Slide by Gary McTaggart (Valve)

Image processing

- Start with ordinary model
  - Render to backbuffer
- Render parts that are the sources of glow
  - Render to offscreen texture
- Blur the texture
- Add blur to the scene





More glow

Assets courtesy of Monolith & Disney Interactive

From *“Tron”*

To learn more…

- http://www.graphics.stanford.edu/courses/cs448a-01-fall/
  https://graphics.stanford.edu/wikis/cs448-07-spring
- http://s09.idav.ucdavis.edu
  http://s08.idav.ucdavis.edu
- http://www.beyond3d.com
- http://ati.amd.com/developer/techpapers.html
- http://developer.nvidia.com
  http://download.nvidia.com/developer/SDK/Individual_Samples/samples.html
  http://download.nvidia.com/developer/SDK/Individual_Samples/effects.html
  http://developer.nvidia.com/page/tools.html

Game Development Outline

- Game Development
  - Typical Process
- What's in a game?
  - Game Simulation
  - Numeric Computation
  - Shading

Tim Sweeney

http://www.cs.princeton.edu/~dpw/popl/06/Tim-POPL.ppt

Game Development: Gears of War

- Resources
  - ~10 programmers
  - ~20 artists
  - ~24 month development cycle
  - ~$10M budget
- Software Dependencies
  - 1 middleware game engine
  - ~20 middleware libraries
  - OS graphics APIs, sound, input, etc

Software Dependencies

Gears of War
Gameplay Code
~250,000 lines C++, script code

Unreal Engine 3
Middleware Game Engine
~250,000 lines C++ code

DirectX Graphics

OpenAL Audio

Ogg Vorbis Music Codec

Speex Speech Codec

wx Widgets Window Library

ZLib Data Compr-ession

...

Game Development – Platforms

- The typical Unreal Engine 3 game will ship on:
  - Xbox 360
  - PlayStation 3
  - Windows
- Some will also ship on:
  - Linux
  - MacOS

What is in a game?

The obvious:

- Rendering
- Pixel shading
- Physics simulation, collision detection
- Game world simulation
- Artificial intelligence, path finding

But it’s not just fun and games:

- Data persistence with versioning, streaming
- Distributed Computing (multiplayer game simulation)
- Visual content authoring tools
- Scripting and compiler technology
- User interfaces

Three Kinds of Code

- Gameplay Simulation
- Numeric Computation
- Shading

Gameplay Simulation

- Models the state of the game world as interacting objects evolve over time
- High-level, object-oriented code
- Written in C++ or scripting language
- Usually garbage-collected

Gameplay Simulation – The Numbers

- 30-60 updates (frames) per second
- ~1000 distinct gameplay classes
  - Contain imperative state
  - Contain member functions
  - Highly dynamic
- ~10,000 active gameplay objects
- Each time a gameplay object is updated, it typically touches 5-10 other objects

Numeric Computation

- Algorithms:
  - Scene graph traversal
  - Physics simulation
  - Collision Detection
  - Path Finding
  - Sound Propagation
- Low-level, high-performance code
- Written in C++ with SIMD intrinsics
- Essentially functional
  - Transforms a small input data set to a small output data set, making use of large constant data structures.

Shading

- Generates pixel and vertex attributes
- Written in HLSL/CG shading language
- Runs on the GPU
- Inherently data-parallel
  - Control flow is statically known
  - “Embarassingly Parallel”

Shading – The Numbers

- Game runs at 30 FPS @ 1280x720p
- ~5,000 visible objects
- ~10M pixels rendered per frame
  - Per-pixel lighting and shadowing requires multiple rendering passes per object and per-light
- Typical pixel shader is ~100 instructions long
- ~500 GFLOPS compute power

Three Kinds of Code

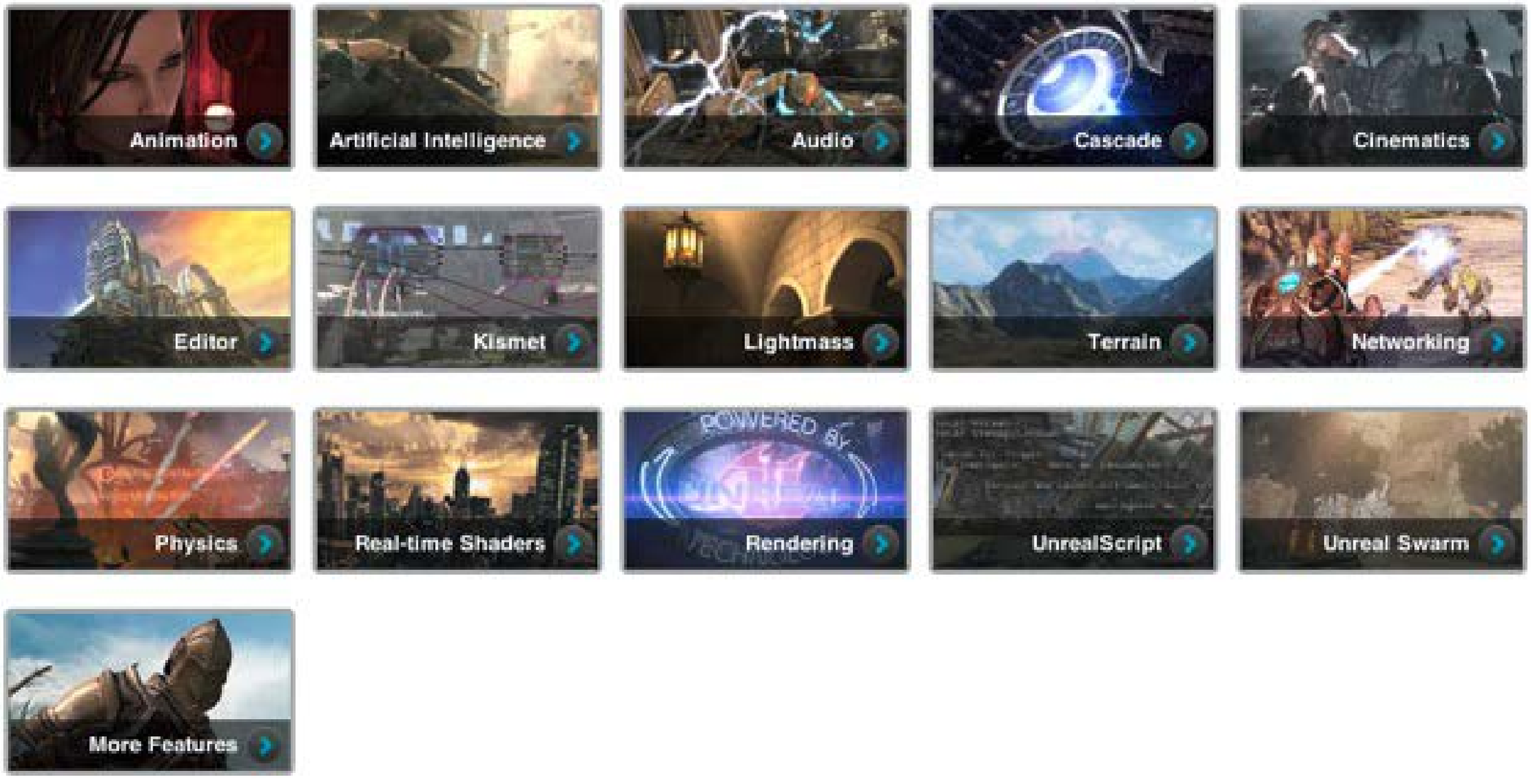
	Game Simulation	Numeric Computation	Shading
Languages	C++, Scripting	C++	CG, HLSL
CPU Budget	10%	90%	n/a
Lines of Code	250,000	250,000	10,000
FPU Usage	0.5 GFLOPS	5 GFLOPS	500 GFLOPS

Gears of war 3

- (Unreal engine)
- Smart conservative collision detection
- Blades of grass using vertical textures
- Anisotropic mipmapping not bad
- Physics
- Visibility culling
- Ambient occlusion
- Softbody dynamic, fracture

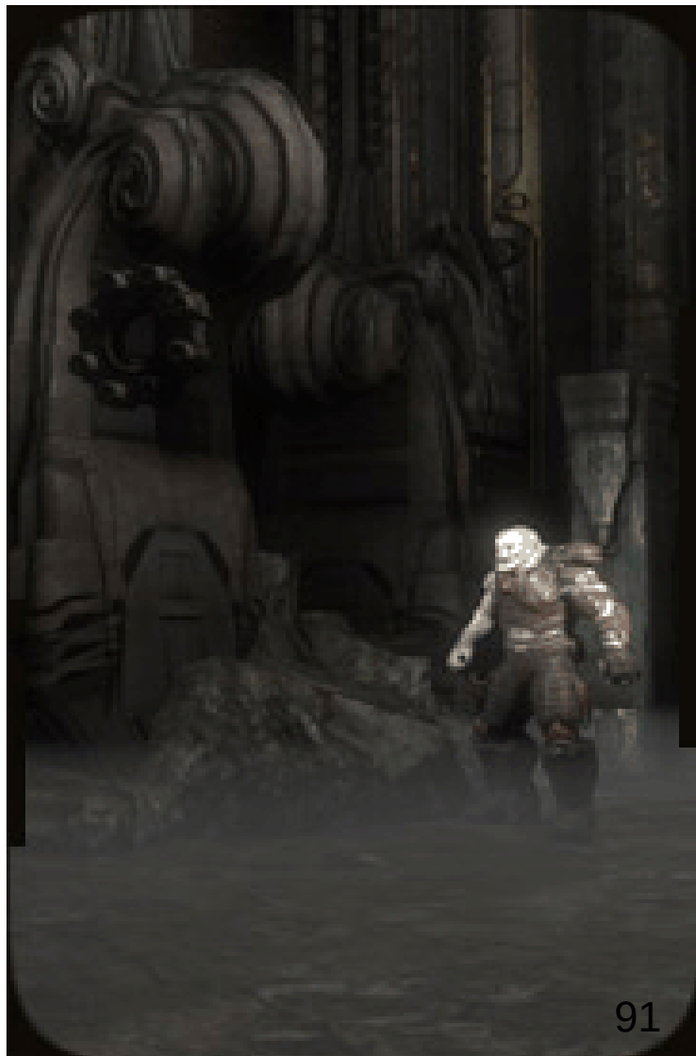
Unreal Engine

- http://www.unrealengine.com
- Mostly for First Person Shooters
- Animation, rendering, sound, etc.



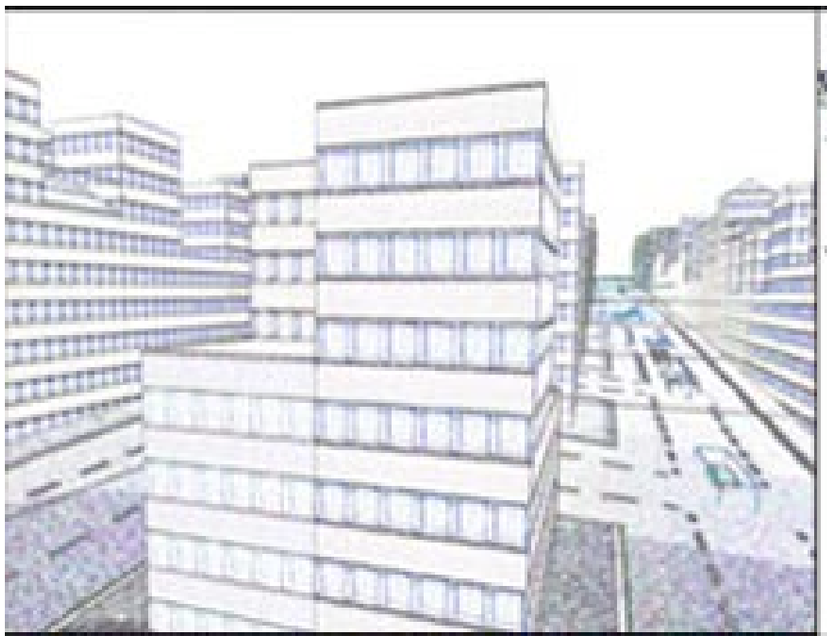
Rendering in Unreal Engine 3

- http://www.unrealtechnology.com/features.php?ref=rendering
- Multithreading
- Visibility culling
- High Dynamic Range (more than 0-255)
- Pixel shaders: normal map, Phong; anisotropic effects; displacement mapping; etc.
- Platform abstraction layer
- Lighting with spherical harmonics
- Shadow volume, shadow map, PCF
- Volumetric effects (fog)
- texture streaming
- reflection with dynamic environment maps
- Image post-processing: bloom, tone mapping, etc.
- Lots of shader support

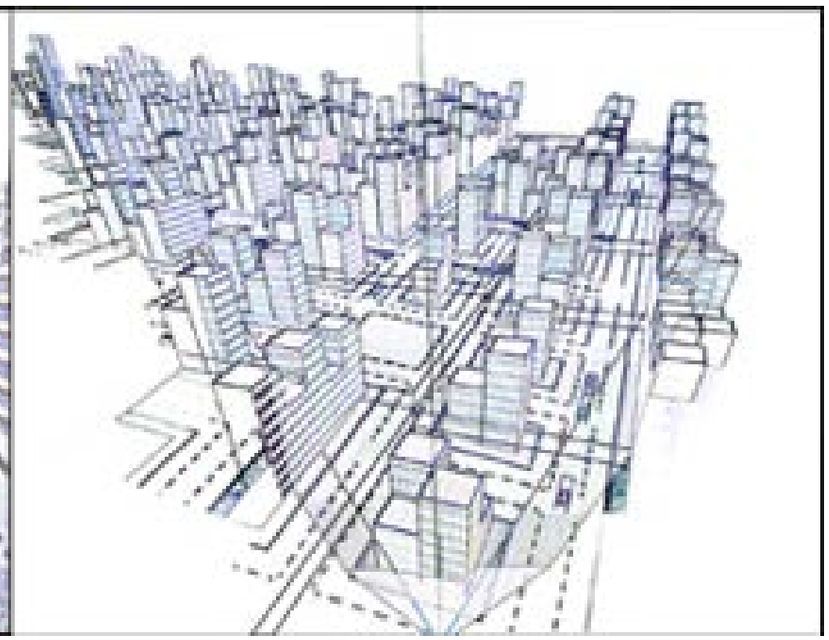


Umbra’s occlusion culling

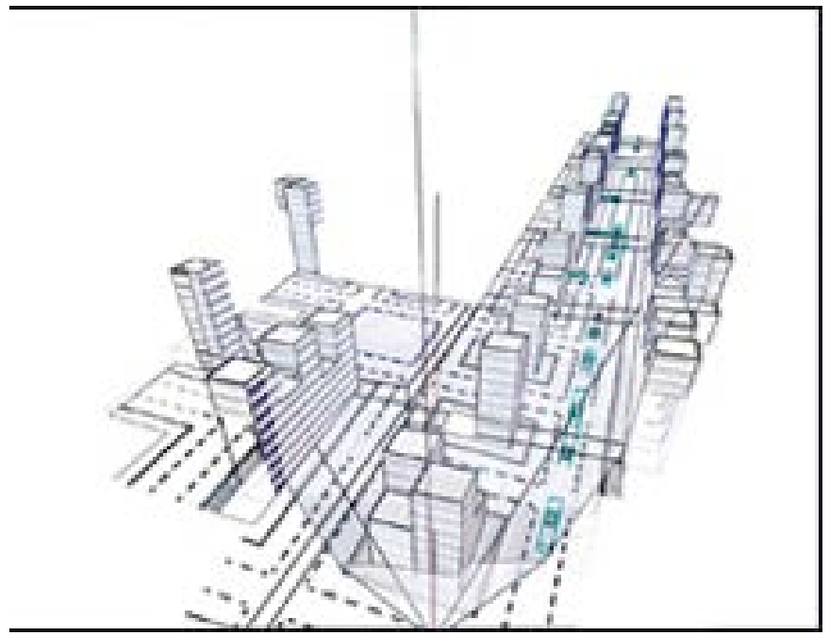
- http://www.umbrasoftware.com/index.php?products&umbra



A view of a 3d-scene



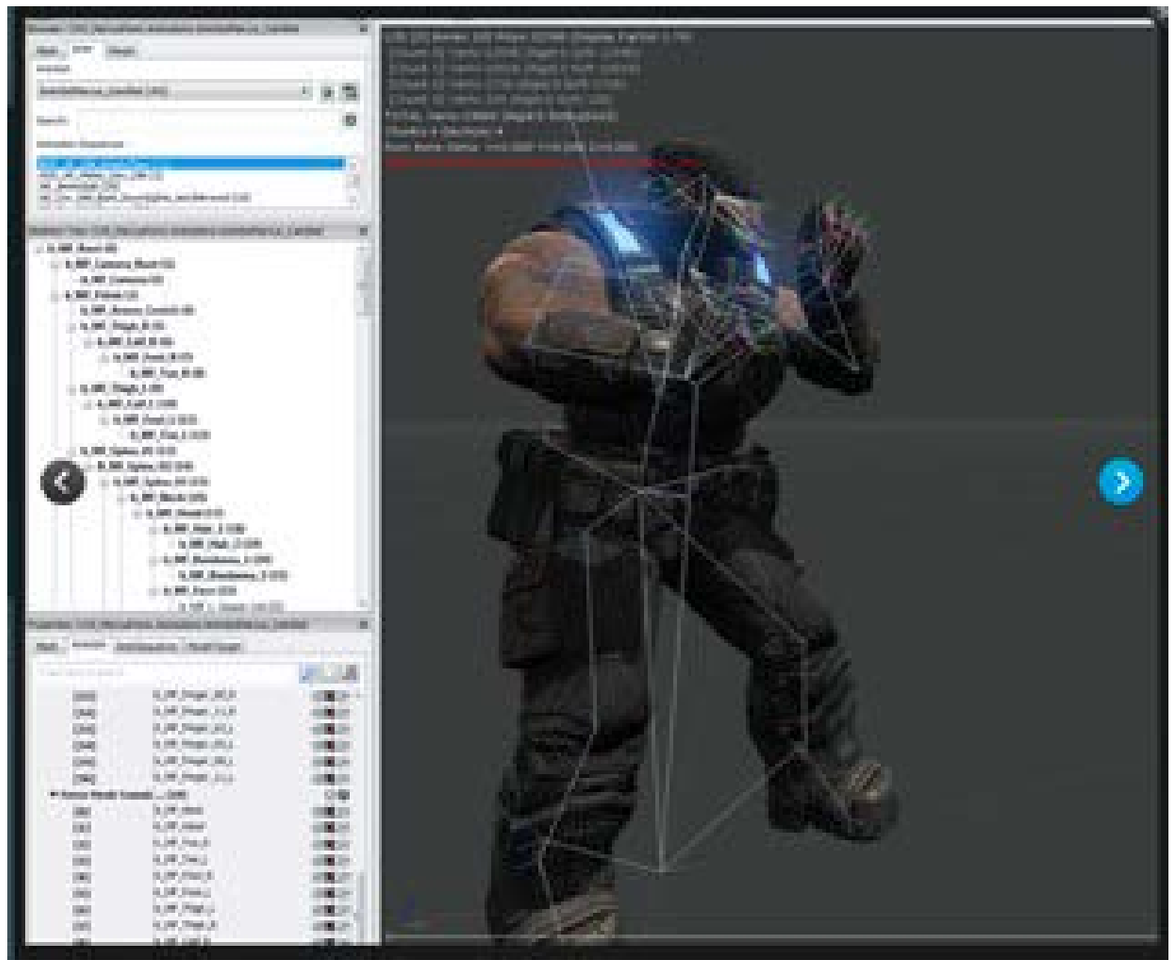
The same view seen from higher perspective.



The same scene with Umbra culling off the nonvisible objects. And it works real time!

Animation

- http://www.unrealengine.com/features/animation/
- Skinned mesh rendering (up to 4 bones / vertex)
- Mesh and bone Level of Detail
- Blend shapes
- Keyframed animation
- Particle system
- Rigid-body physics
- Cloth simulation
- Soft-body physics
- Fracture
- Collision detection



Unreal AnimSet Editor: COG_MarcusFenix_Animations.AnimSetMarcus_CamSkel

File View Mesh AnimSet AnimSequence Notifies MorphSet MorphTarget Animation Compression Window

Browser: COG_MarcusFenix_Animations.AnimSet_CameraMovement

Mesh | Anim | Morph

AnimSet

AnimSetMarcus_CamSkel [246]

Animation Sequences

AR_Cov_Std_Idle_Ready2Blind [11]
AR_Cov_Std_Idle_Ready2Lean_Aim [9]
AR_Cov_Std_Idle_Ready2Mirrored [6]
AR_Cov_Std_Lean_Aim2Idle_Ready [9]
AR_Cov_Std_Lean_DownSights_Aim [30]
AR_Cov_Std_Nudge_Stop_F10 [25]
AR_Cov_Std_Nudge_Stop_F30 [25]
AR_Cov_Std_Peek_Side [50]
AR_Cov_Std_Reload [75]
AR_Cov_Std_Reload_Fail [120]
AR_Cov_Std_Run [20]
AR_Cov_Std_Run2Cov_Std_Idle [31]
AR_Cov_Std_Slip [21]
AR_Cov_Std_Slip_Mirrored [21]
AR_Cov_Std_SwatTurn_End [25]

Skeleton Tree: COG_MarcusFenix_Animations.AnimSet_CameraMovement

b_MF_Pelvis (1)
b_MF_Armor_Crotch (2)
b_MF_Spine_01 (3)
b_MF_Spine_02 (4)
b_MF_Spine_03 (5)
b_MF_Clavicle_L (6)
b_MF_Armor_Sho_L (7)
b_MF_UpperArm_L (8)
b_MF_Armroll_L (9)
b_MF_Forearm_L (10)
b_MF_Hand_L (11)
b_MF_Finger_01_L (12)
b_MF_Finger_02_L (13)
b_MF_Finger_03_L (14)
b_MF_Finger_04_L (15)
b_MF_Finger_05_L (16)
b_MF_Finger_06_L (17)
b_MF_Finger_07_L (18)
b_MF_Finger_08_L (19)
b_MF_Finger_09_L (20)
b_MF_Finger_10_L (21)
b_MF_Finger_11_L (22)
b_MF_Finger_12_L (23)
b_MF_Thumb_01_L (24)
b_MF_Thumb_02_L (25)
b_MF_Thumb_03_L (26)
b_MF_Weapon_L (27)
b_MF_Weapon_L_End (28)
b_MF_ForeTwist2_L (29)

Properties: COG_MarcusFenix_Animations.AnimSet_CameraMovement

Mesh | AnimSet | AnimSequence

AnimSet

bAnimRotationOnly

UseTranslationBoneNam...

[0] b_MF_Weapon_L
[1] b_MF_Pelvis
[2] b_MF_Weapon_R

LOD [0] Bones:110 Polys:11255 (Display Factor:1.57)
Verts:7566 (Rigid:2494 Soft:5072)
Chunks:2 Sections:2

20: b_MF_Weapon_L_End
21: b_MF_Weapon_L

Pos: 24.2 pct, Time: 0.4037s, Len: 1.6667s

AnimSet: AnimSetMarcus_CamSkel Mem: 3.48MB/9.30MB

AnimSeq: AR_Cov_Std_Lean_DownSights_Aim Len: 1.67s Mem: 7.32KB/31.72KB

Unreal Tournament 3 [EFX]

Cascade: P_EFX_Explosion_ThickSmoke

Window

FireBase

Flames

Sparks

Required

Spawn

Lifetime

Initial Size

Initial Velocity

SubImage Index

Initial Rotation

Size By Life

Initial RotRate

Initial Color

Scale Color/Life

Acceleration

Cylinder

Initial Location

Color Over Life

Accelerate

Unreal Curve Editor: P_Robot_Gib_Spark

ColorOverLife

LifeMultiplier

AlphaOverLife

SizeScale

StartLocation

Generic

Actor Classes

Groups

Level

Referenced Assets

Show All Resource Types

FaceFX Asset

Font

Fractured Static Mesh

Lens Flare

Material

Material Instance Constant

Material Instance Time Varying

MaterialLackingPhysMat

MorphTargetSet

MorphWeights

Particle System

Physical Material

Physics Asset

PhysX Destructible

PhysX Particle System

PostProcess

Envy_Effects*

Bubbles

Dirt

Energy

Explosion

Fire

Flames

Level

Materials

Particles

ScreenShot_Effects

Smoke

Snow

Tests

Vehicle_Damage

VH_Deaths

Envy_Effects2*

Particles

P_EFX_Explosion_ThickSmoke

1 Selected, 1 Total

See also Valve source engine

- Slides from Jason Mitchell
- http://www.pixelmaven.com/jason/