Graphics

Introduction

- A glimpse into what game graphics programmers do
- System level view of graphics architectures & pipelines
- Intro to commonly used rendering techniques in games

Game Graphics Programmers

- Create infrastructure to realize artistic vision of product
  - Work closely with Art Director
  - Work closely with artists to define art production requirements
- Utilize the hardware and APIs of the target platforms
- Integrate renderer with other game components
  - Art pipeline, AI, front-end etc.
- Achieve interactive performance
- Work with memory constraints

What do you see?

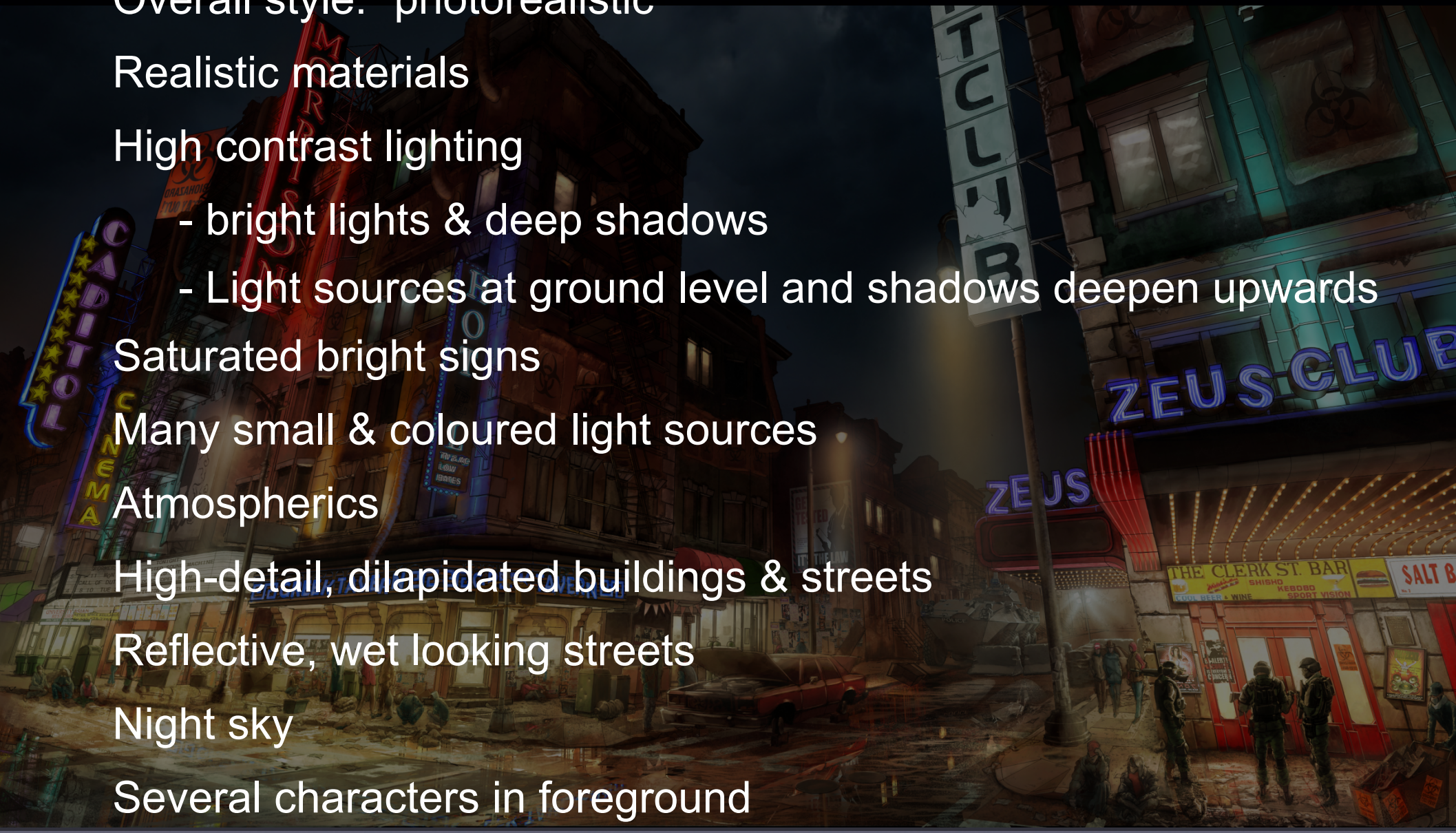
How would you render this?

Lots going on here:

- Objects (car, road, ground, bushes, driver)
- Sky
- Lighting (direct and indirect)
- Material properties – transparency, specular
- Reflections
- Shadows
- Motion blur
- Color correction

Deconstruct the Art Direction: What do you see?

Deconstruct the Art Direction: What do you see?



Overall style: “photorealistic”

Realistic materials

High contrast lighting

- bright lights & deep shadows
- Light sources at ground level and shadows deepen upwards

Saturated bright signs

Many small & coloured light sources

Atmospherics

High-detail, dilapidated buildings & streets

Reflective, wet looking streets

Night sky

Several characters in foreground

Art Production Requirements

- Efficiently generate buildings & varied store fronts
- Generating reusable materials – shader + textures
- Decide geometry vs texture detail
- Create and place signage
- Create and place reusable street props
- Workflow for placing lights
- Specifying light volumes / atmospherics
- Specify secondary textures (grime, puddles)
- Build art for multiple level of details
- Character production pipeline
  - Which has a whole list of their own

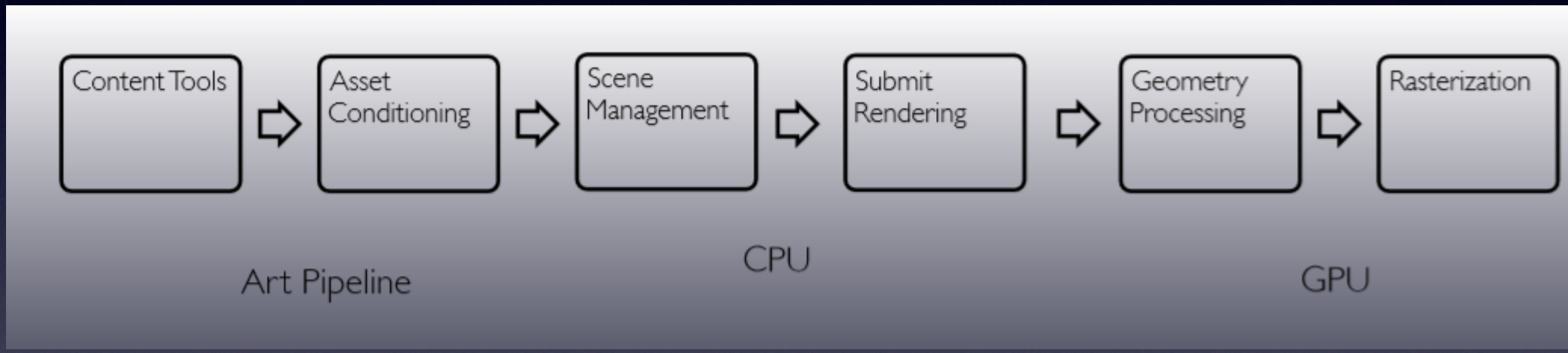
Rendering Requirements

- Direct lighting – spot lights, point lights, light cards
- Indirect lighting / ambient occlusion
- Static light sources
- Shaders (Asphalt, wet asphalt, brick, windows)
- Emissive (animated?) shader for neon signs
- High dynamic range
- Billboard for light volumes
- Skybox
- Reflection maps for puddles
- Etc ...

Rendering Pipeline

Quick overview of getting from the art tools to pixels on screen

Elements of a Renderer



Art Creation

- Artists create resources
  - Models
  - Textures
  - Shaders?
- Off the shelf vs custom tools
  - Off-the-shelf : Maya / Photoshop / etc.
  - In house: Shader editor, level editor
- Exporter plugins

Art Pipeline

- Assets generally need some offline processing to be ready for in-game use
  - Export of geometry
  - Optimization of geometry
  - Texture compression
  - Shader compilation
- Some elements of rendering may be generated here
  - Baking lights
  - Generating world representation
  - Geometry -> bump maps
  - And many more…
- Some portions may be hardware dependant
  - Consoles can often compile to final shader code, PCs can’t

Scene Management

- Visibility determination
  - Frustum culling – quickly reject objects that lie outside the view frustum
  - Occlusion culling – quickly reject objects that are in the frustrum but covered by other objects
  - Linear sort probably good enough, but there are some more exotic data structures (octree, KD-tree, etc.)
- Level of detail (LOD) management
  - Replace an object with simpler form when object is small on the screen
  - May involve simplifying shaders, animation, mesh
- Translucency sorting
  - z-buffer will handle out-of-order draws of opaque stuff
  - Translucent stuff needs to be manually sorted

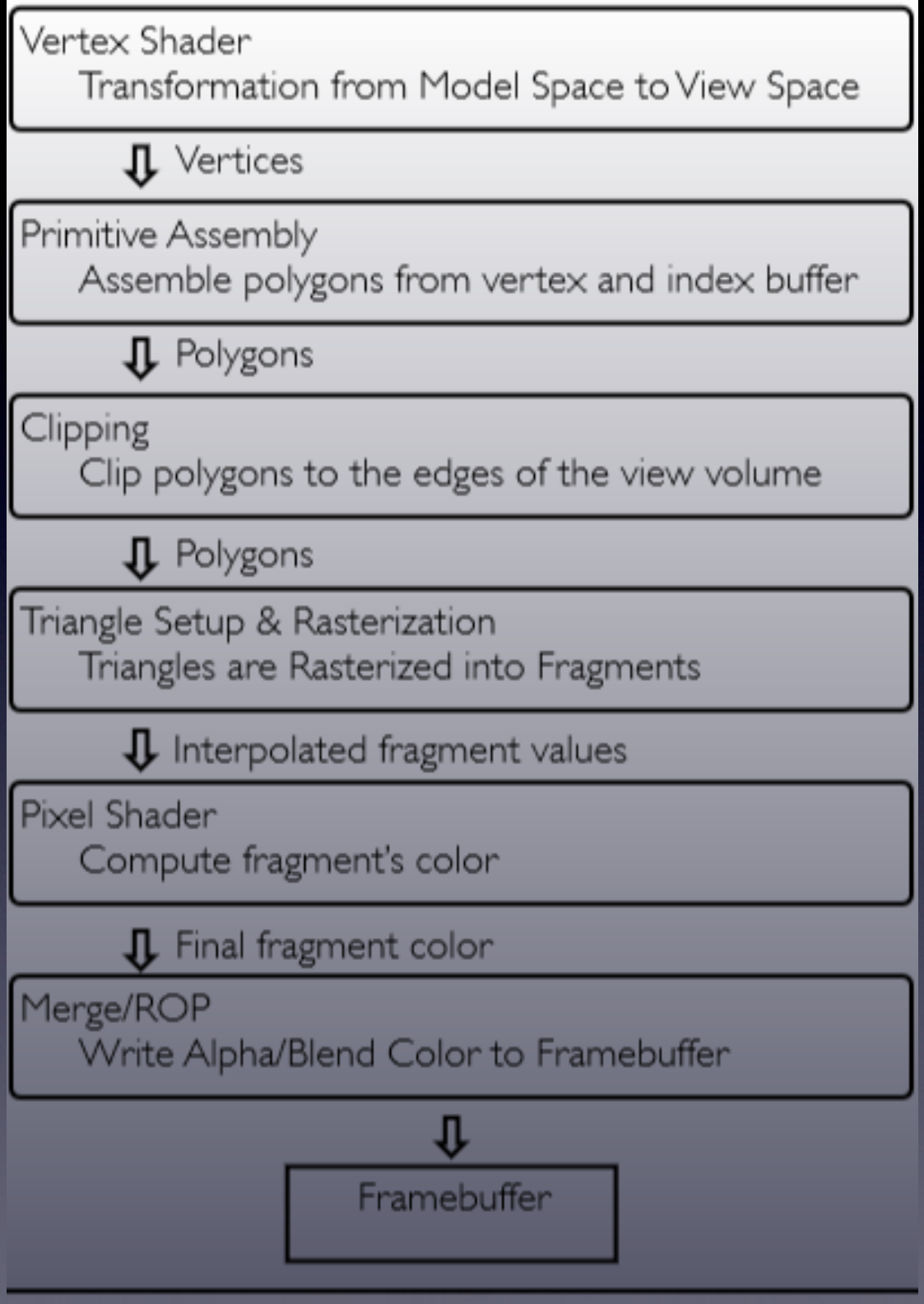
Submit geometry to GPU for rendering

- Iterate through list of visible objects
- Iterate over each submesh-material pair
- Set render state based on material's specification
  - Set vertex and pixel shaders
  - Set uniforms (parameters to shaders)
  - Set a few other state elements (clipping, alpha blending)
  - Set vertex and index buffers
- Call DrawIndexPrimitive() or glDrawArray()

Performance

- Graphics hardware is deeply pipelined & highly parallelized
  - Submit large vertex buffers
  - Avoid state switching
  - Reading from render targets causes stalls
- Rendering engine design considerations:
  - Don’t chop the world up too finely
    - Reduce draw calls
    - Even at expense of drawing more off-screen
  - Batching by state:
    - Sort primitives by material type
  - Minimize data access by CPU:
    - Geometry, textures, display lists, positions are uploaded to VRAM once per frame
  - Minimize reading back from render targets
    - Organize processing into multiple passes

GPU Pipeline



Programmable vs. Fixed Function

- DX7/OpenGL 1.4/OpenGL ES 1.x and earlier hardware used fixed function vertex processing
  - Select from a limited set of states to configure processing of components
  - PS2, Xbox era consoles
- All modern hardware is fully programmable at the vertex & fragment level
- Why do we care?
  - Need to be aware of what portions of API are legacy, particularly with OpenGL.

Vertex Components

- Position, colour, texture, normals, etc.
  - For programmable processing, components of a vertex are defined by the engine

```
// simple vertex format
struct Vertex
{
  float3  pos;
  float3  normalWS;
  float3  tangentWS;
  float3 color;// vertex color
  float2 uv0   // texcoords for diffuse, specular & normal map
  float2 uv1   // secondary texcoords e.g. for grime map lookup
}
```

Geometry Primitives: Triangle Lists

- Simply lists the vertices in groups of 3
- Lots of duplicate vertices

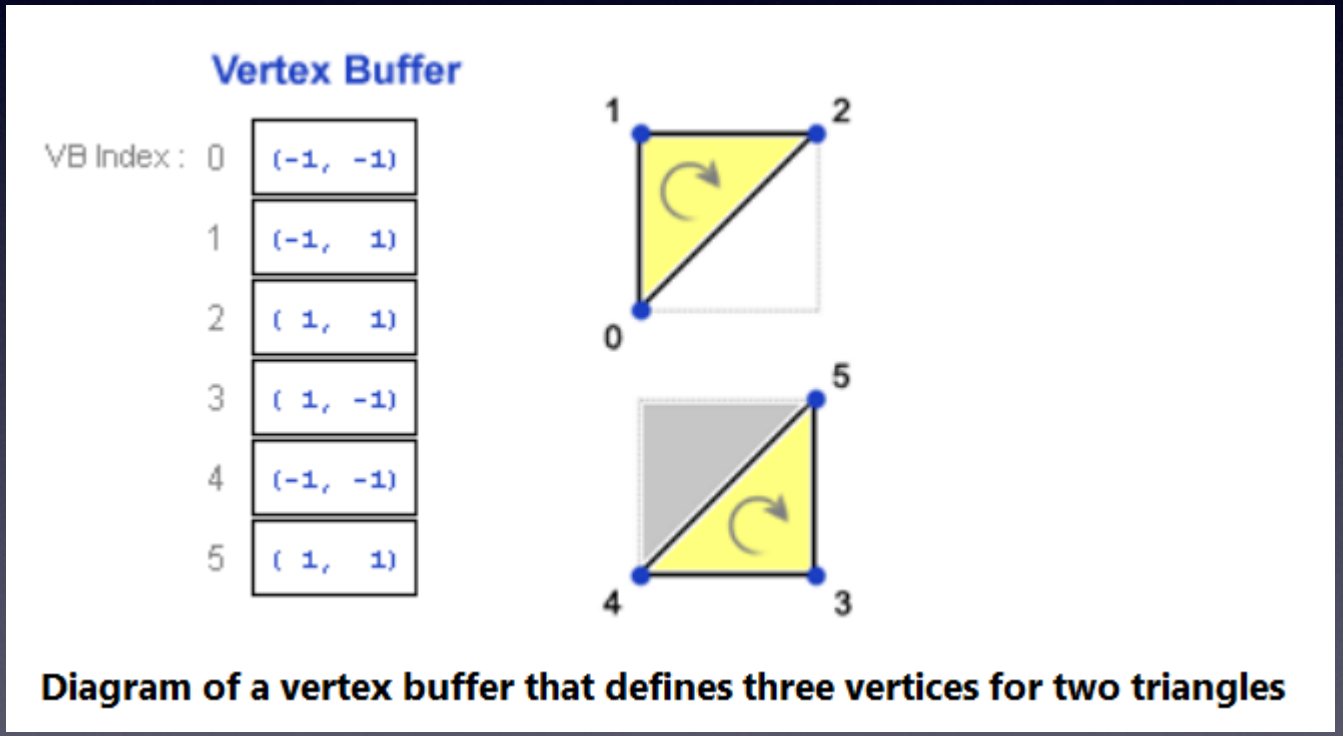


Diagram of a vertex buffer that defines three vertices for two triangles

MSDN Direct3D Reference

Geometry Primitives: Strips & Fans

- Triangle strips & fans
  - Generated by offline tools
  - Reduce vertex duplication
  - Predefined vertex orders

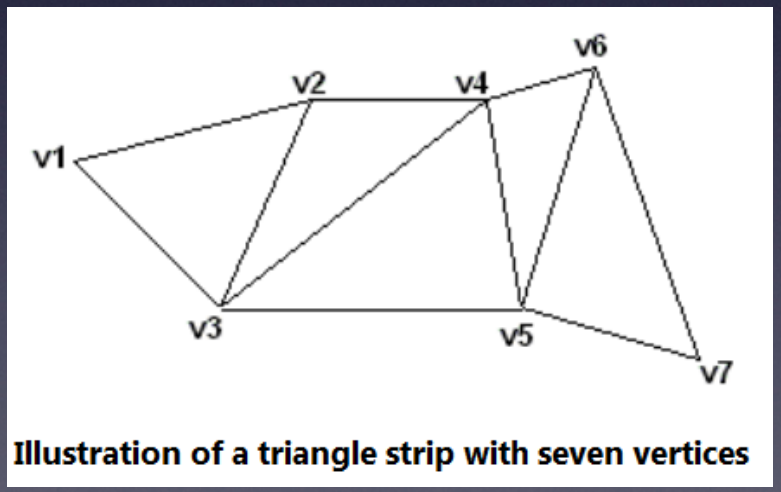


Illustration of a triangle strip with seven vertices

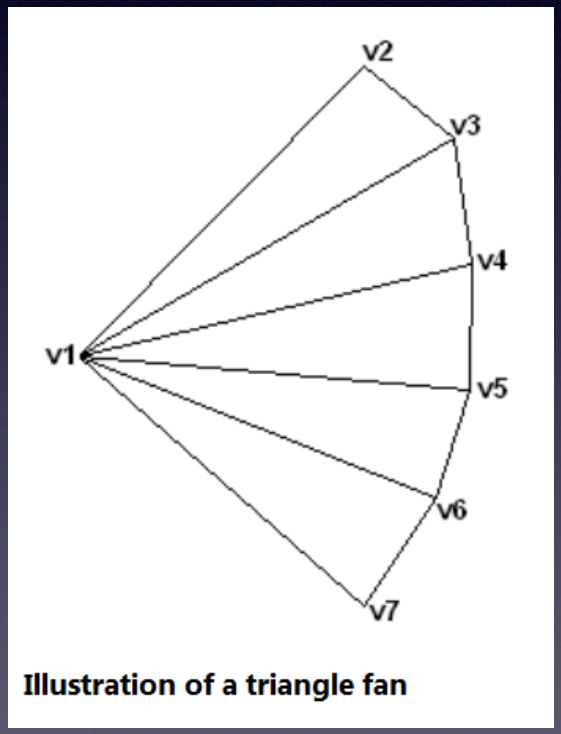


Illustration of a triangle fan

MSDN Direct3D Reference

Geometry Primitives: Index Triangle Lists

- Commonly used particularly with Programmable pipelines
- Required to take advantage of GPU vertex caching

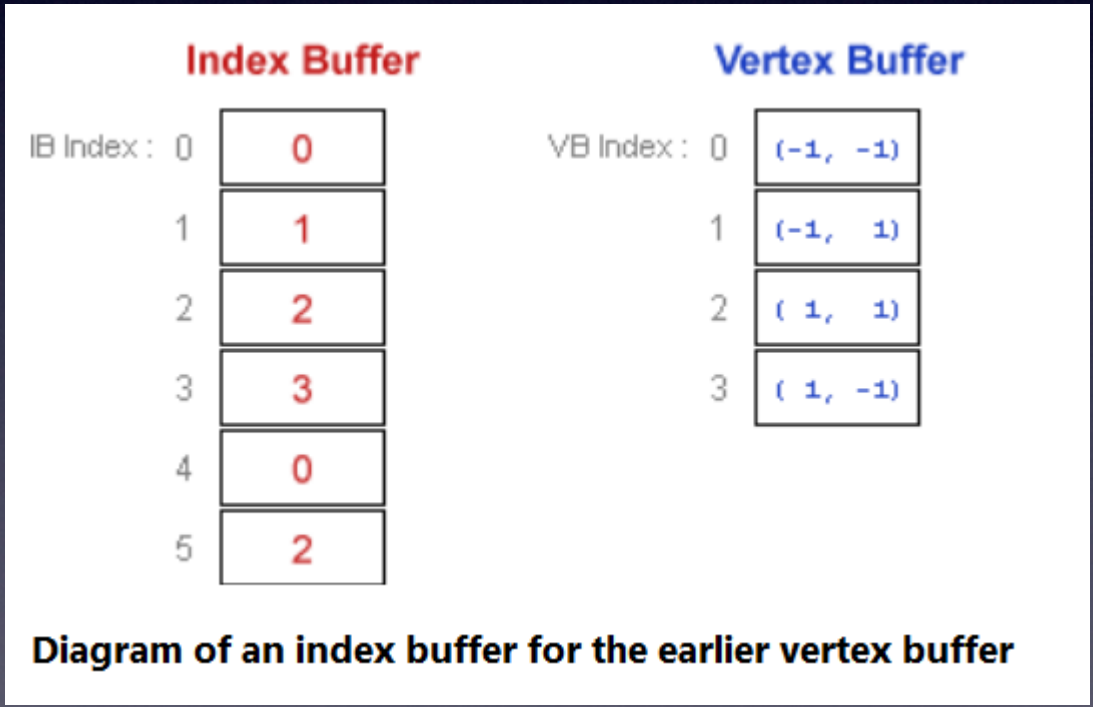
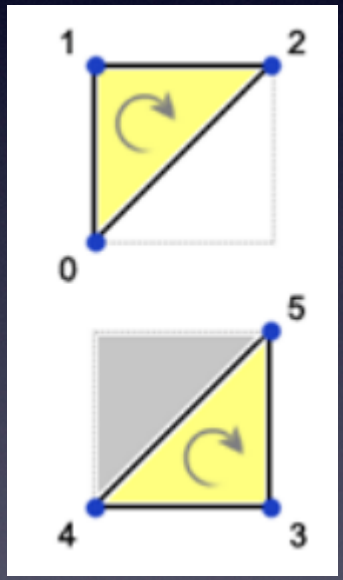


Diagram of an index buffer for the earlier vertex buffer



MSDN Direct3D Reference

Programmable Shaders

- Shader program accesses data by two methods:
  - Registers – Input & Constant
    - Input come from vertex stream, constants (uniforms) are set from calling code
  - Texture maps
  - Input registers can be passed between vertex and pixel shaders (‘varying’)
- Supports vector and matrix types intrinsically e.g. float3, float4
  - Float x[4] is not same as float4 x
  - ALU performs math operations on 3 or 4 components in a single instruction
    - Registers are 128 bit wide
- Avoid use of conditional logic

Example vertex shader

```
varying vec3 normal;
varying vec3 vertex_to_light_vector;

void main()
{
    // Transforming The Vertex
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

    // Transforming The Normal To ModelView-Space
    normal = gl_NormalMatrix * gl_Normal;

    // Transforming The Vertex Position To ModelView-Space
    vec4 vertex_in_modelview_space = gl_ModelViewMatrx * gl_Vertex;

    // Calculating The Vector From The Vertex Position To The Light Position
    vertex_to_light_vector = vec3(gl_LightSource[0].position - vertex_in_modelview_space);
}
```

Example fragment shader

```
varying vec3 normal;
varying vec3 vertex_to_light_vector;

void main()
{
    // Defining The Material Colors
    const vec4 AmbientColor = vec4(0.1, 0.0, 0.0, 1.0);
    const vec4 DiffuseColor = vec4(1.0, 0.0, 0.0, 1.0);

    // Scaling The Input Vector To Length 1
    vec3 normalized_normal = normalize(normal);
    vec3 normalized_vertex_to_light_vector = normalize(vertex_to_light_vector);

    // Calculating The Diffuse Term And Clamping It To [0;1]
    float DiffuseTerm = clamp(dot(normal, vertex_to_light_vector), 0.0, 1.0);

    // Calculating The Final Color
    gl_FragColor = AmbientColor + DiffuseColor * DiffuseTerm;
}
```

Graphics Hardware Performance

- Vertex Processing Rate
  - Depends on GPU clock rate & number of vertex ALUs available
  - No. of vertex shader instructions
- Fragment processing rate
  - Depends on GPU clock rate + number of ALUs available
  - No. of pixel shader instructions
- Pixel processing, texture fetch rate
  - Depends on GPU/VRAM bandwidth
- Modern graphics hardware is massively parallel
  - PS4 : 18 cores, 64 shader units per core = 1152 shader units

Break

Rendering techniques

- How do we render individual types of things that model some of the phenomena we want to capture?
- Huge number of techniques available
  - Lighting
  - Shadows
  - Framebuffer Effects
  - Billboards & Particles

Lighting

Defines the look of the environment

Many techniques – as much art as science

Uses of Lighting

- Can’t see anything without lights
- Directs the viewer’s eye
- Creates depth
- Conveys time-of-day and season
- Conveys mood, atmosphere and drama
- Express character’s inner state

Basic Lighting Calculation

- Lighting algorithms are concerned with:
  - Properties of the lights:
    - Color, intensity, shape, fall-off
  - Properties of the surface:
    - Shininess, roughness, color, transparency
    - Refraction index
- Blinn/Phong lighting model
  - Intensity = Ambient + Diffuse + Specular
- Implemented in pixel shader
  - Allows lighting interaction to be specified per material
  - Can model more advanced interactions by having a “depth map” e.g. to model subsurface scattering

Light Sources

- Dynamic Lights
  - Direct lighting is computed per frame at runtime
  - Dynamic game objects will receive light
  - Points, directional, spot
- Static Lights
  - Used to light environment
  - Captures direct and/or indirect lighting
  - Compute offline & store e.g. vertex colors, light maps, spherical harmonics coefficients
- Use a combination of techniques
  - E.g. emissive objects
  - May have special lights for characters and worlds, and small subset that affect both

Image Based Lighting Approaches

- Modern games make use of image data for Lighting
  - Specular Maps
  - Normal Maps
  - Environment Cube Maps
  - Ambient occlusion maps

Normal Maps

- Normal vectors are encoded in RGB color channels
- Transformed from tangent to world space for lighting computation
- More surface detail with less geo

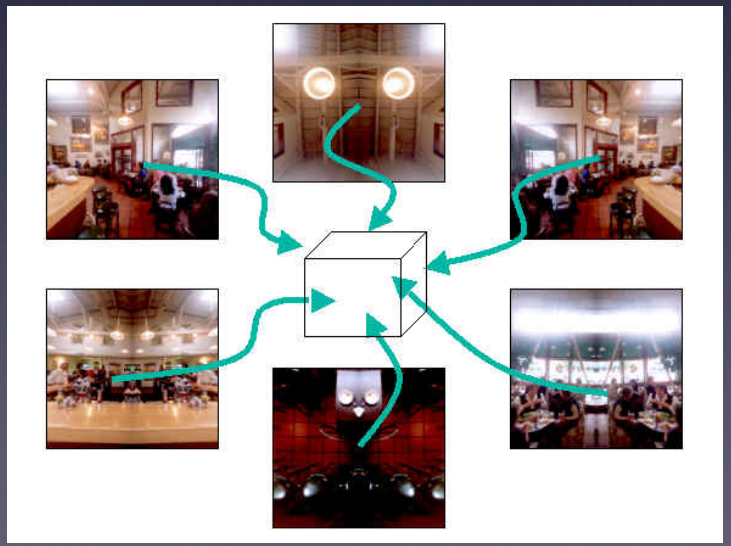
Without Normal Map

With Normal Map

created by David Maas

Environment Maps

- A cube map generated from six directions
- Map on to the six inner surfaces of a box at infinity
- Use to render reflections on reflective surfaces, e.g. window panes
- Can be used for:
  - Specular color: approximate roughness by sampling in lower mips
  - Luminance: if cube map is HDR



Ambient Occlusion Maps

- Describes how much light each point of a surface receives when uniformly lit
- Computed with offline tools
  - Construct a hemisphere with large radius centered on the point
  - Determine what percentage of the hemisphere's area is visible from the point (i.e. how much of the hemisphere is occluded by other surfaces)

Lots of dynamic lights

- Gamer: Oh, crap! Look at all those bad guys!
- Rendering Programmer: Oh, crap! Look at all those dynamic lights!

Lots of dynamic lights

- Real world has lots of lights in it
  - Light maps let you have as many static lights as you want
  - But what we really want is lots of dynamic lights
- Hard to get lots of dynamic lighting with conventional techniques
  - You tend to end up paying for it even when not benefiting
- But, when there are lots of lights they tend to be small
- If we only render pixels touched by given light, cost is manageable

Deferred Rendering

- The way to do this is to separate surface property calculation and lighting
- Components needed in the lighting calculation are stored in intermediate buffers (per pixel), e.g.
  - Diffuse color
  - Specular power, intensity
  - Normals
  - Depth
- Lighting computation is done in screen space – once per pixel

- Geometry Pass:
  - Render scene geometry, write lighting components to G-Buffers

- Light Accumulation Pass:
  - Initialize light accumulation buffer with “baked” lighting components
  - Determine lit pixels
  - Render pixels affected by each light, and accumulate

Shadows

Commonly used techniques

Where did this guy jump from and how far is he off the ground?



A shadow...

- Grounds objects in the scene
- Generally it is better to have an inaccurate shadow than none at all
- Gives hint of how high the object is from the ground
- Most commonly used techniques:
  - Projected blob
  - Projected texture
  - Shadow volumes
  - Shadow maps

Technique #1: Blobs

- Raycast to ground
- Draw textured quad
- Scale quad based on object’s height
- Multiple blobs for articulated objects
  - E.g. One per leg
- Stretch and squash shadow based on object speed


HELTER SHELTER

Technique #2: Projected Texture

- Render shadow caster from perspective of light sources into a temporary buffer
  - Often use lower-complexity model
- Project result onto shadow-receiving surfaces in the world
- Similar to blob but requires an extra rendering pass
- Watch for aliasing

Technique #3: Stencil Volumes

- Compute silhouette edges of geometry with respect to light
- Extrude edges in the direction of the light rays
- Draw scene to framebuffer updating z-buffer
- Clear stencil buffer (zero out)

Stencil Volume (cont'd)

- Turn off z-writes
- Draw front-facing polys of shadow volume (from camera view point)
  - front face of volume: increment stencil buffer
- Draw back-facing polys of shadow volume (from camera view point)
  - back face of volume: decrement stencil buffer
- If the backface of the shadow volume is not drawn (fragment fails z-test) then stencil buffer value = +1
- Darken fragments where stencil is not zero.

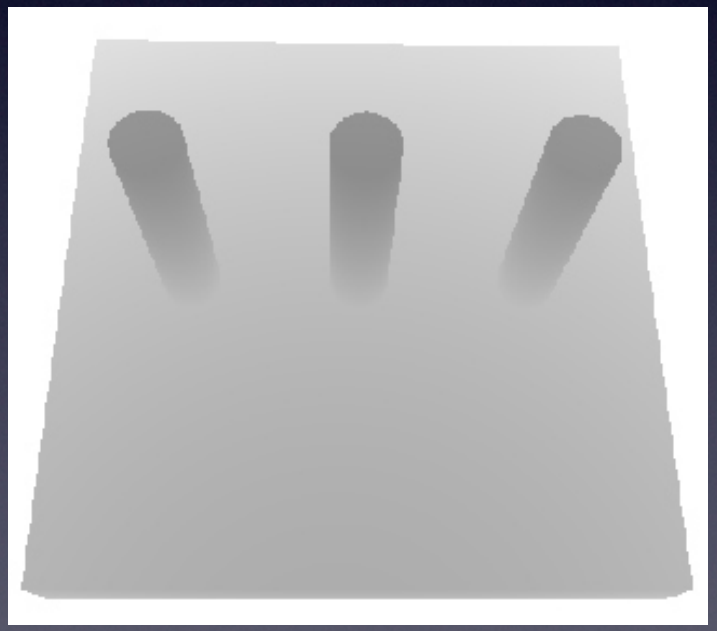

MARS DELTA LABS
LEADING THE WAY

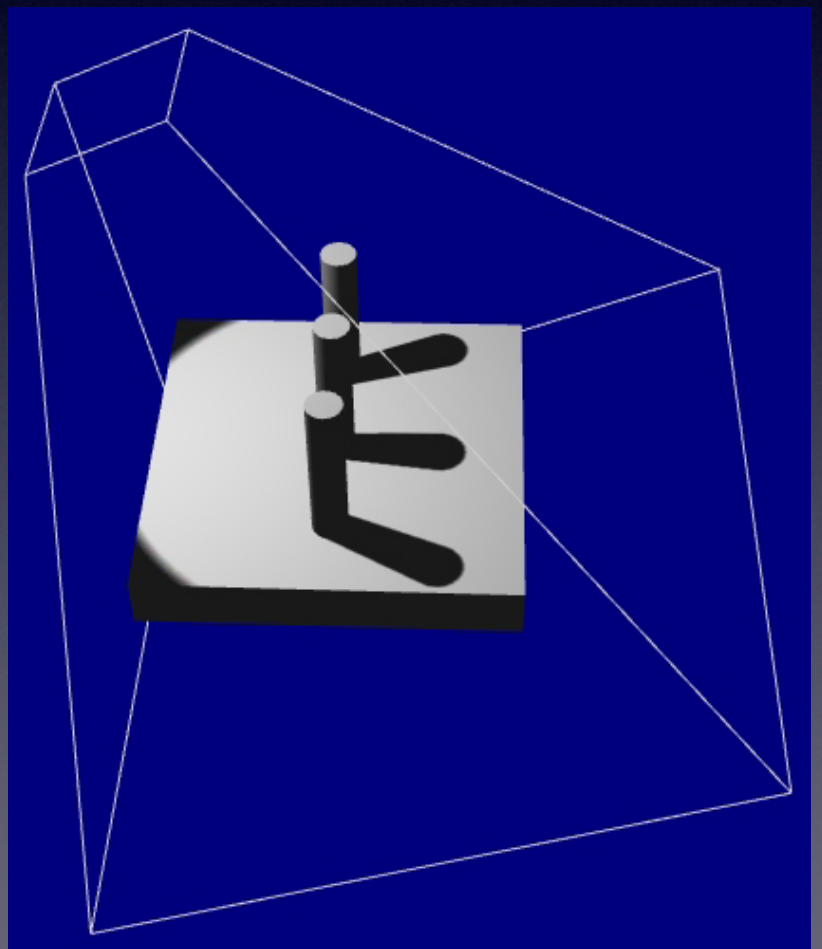
Stencil Volume: Pros & Cons

- Pros
  - Works for arbitrary surfaces
  - Handles self-shadowing nicely
- Cons
  - Hard edges
  - Silhouette edge calculation is expensive
  - GPU intensive (vertex processing, render target writes & reads)

Technique #4: Shadow Maps

- Render scene from point of view of the light
- Save the depth buffer as a texture
  - Shadow map texture contains z-depth objects closest to it





Shadow Map (cont'd)

- Render the scene from point of view of camera. For each vertex:
  - Transform vertex position to light space.
  - Interpolate light space coordinates between vertices
  - Fragment position in light space
  - Convert light space x,y coordinates to u,v
  - Compare light space z-depth with depth stored in corresponding texel of the shadow map

Shadow Maps: Challenges

- Can suffer from severe aliasing problems
  - Need large shadow maps
  - Pre- or post-filter shadow to reduce aliasing (also gives soft shadows)
- Expensive: multiple passes required

Frame buffer effects

FrameBuffer Effects

- Utilize rendered data as a texture for subsequent operations
  - Render-to-texture, frame-buffer-as-texture
- Examples:
  - Motion blur
  - Depth-of-field
    - Copy rendered frame to off-screen buffer, and blur
    - Using z-buffer as a mask, copy blurred frame to rendered frame
  - Refraction / reflection (Predator effect, heat shimmer, water surface)
    - Render or copy to offscreen buffer
    - Composite with perturbed texture co-ordinates
  - Colour correction


Deferred composition


Image with post-processing (depth of field, bloom, motion blur, colorize, ILR)

Color correction

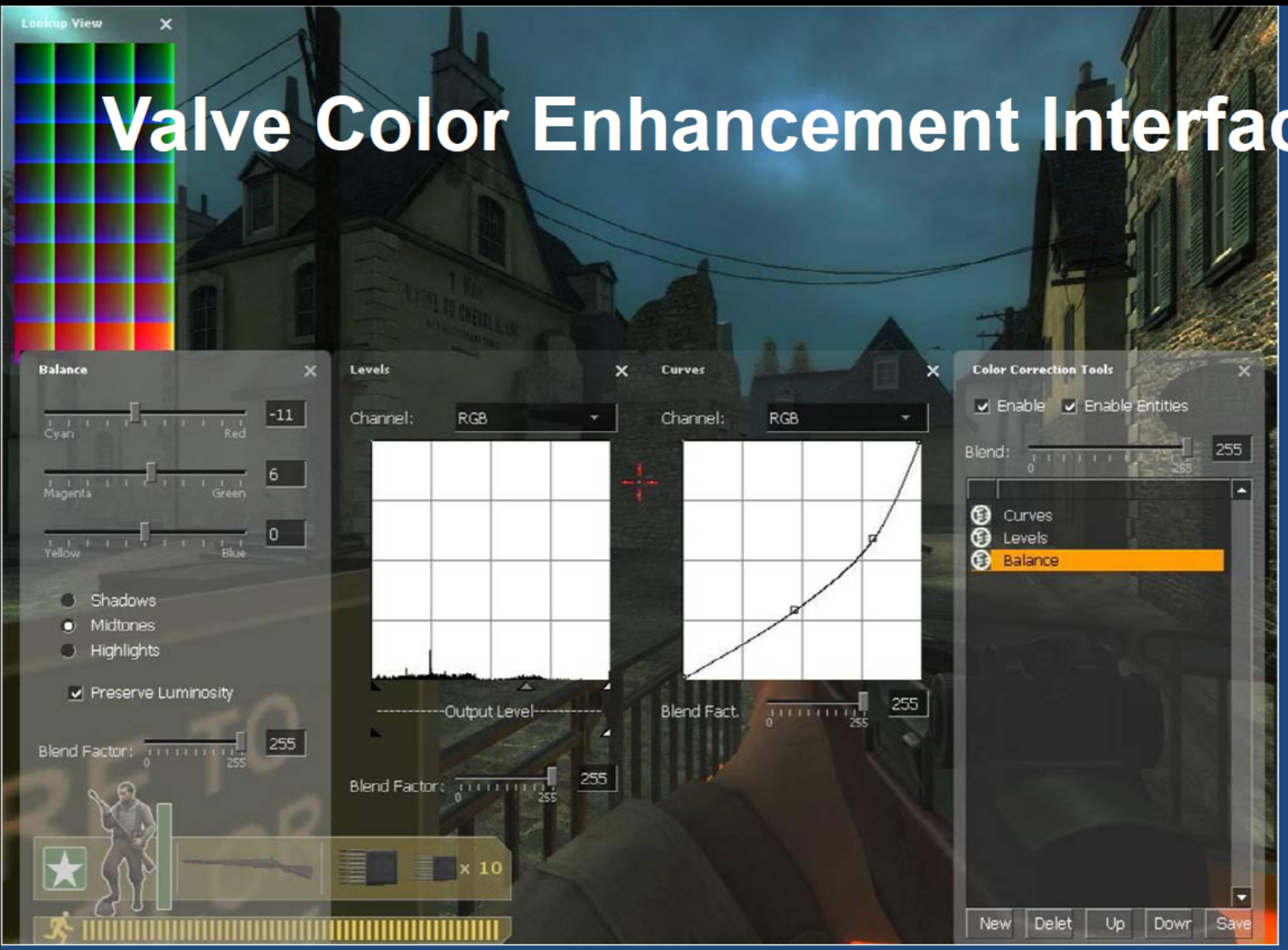
- Technique commonly used in film but gaining popularity in games
- Primary goal same as in film:
  - Creative control over look to manipulate the mood of the viewer
  - Call attention to important visual elements
- Easy to art-direct & allows for changes late in production
- In games:
  - can't tweak color with same precision since scene content changes in unpredictable ways from frame to frame
  - Can be used to implement dynamic game state (such as player's health)

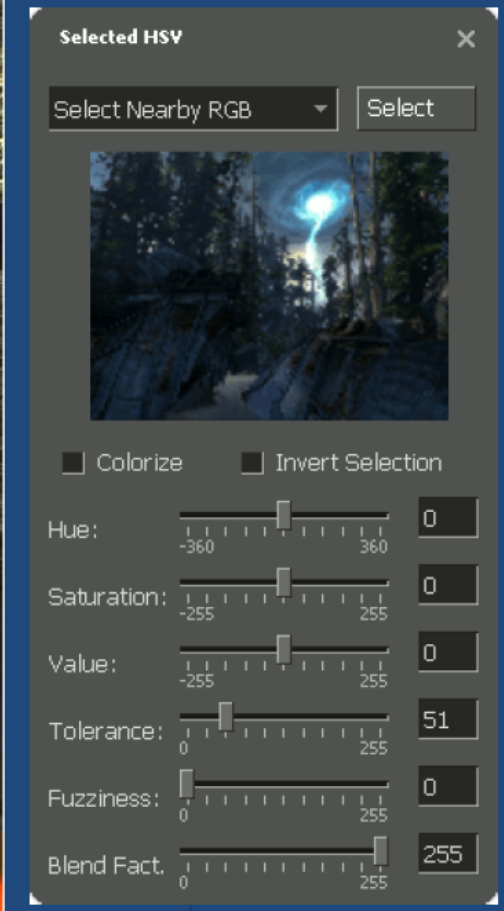
Color LUTs

- Color Look Up Table
  - Represent the RGB color space as a 3D texture
  - 32 x 32 x 32 pixels (8 bit color)
- Use input RGB (from framebuffer), look new color in LUT
- Can be authored:
  - Offline
  - Using in-game Photoshop-like GUI e.g. Valve's Source engine

Valve Color Enhancement Interface

Images used with Valve's permission

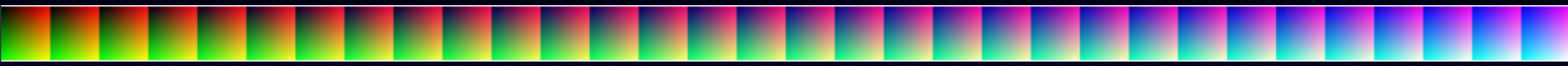




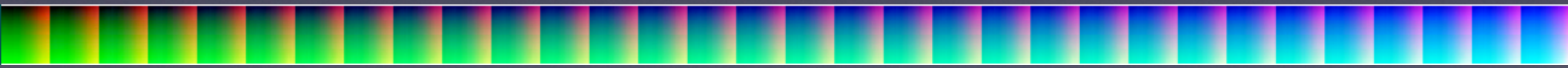
Color Enhancement for Video Games – Siggraph '09
(Used with permission fr author)

Authoring LUTs Out-of-Engine

- Create an “identity LUT”



- Flatten (e.g.) 32 x 32 x 32 cube into 1024 x 32 strip
- Grab (uncorrected) screenshot from the game and paste “identity LUT strip” on it
- Give screenshot + LUT strip to artist to perform color manipulations in external app
- Convert strip back to 3D LUT, import into engine



Example: Left 4 Dead (No Effects)

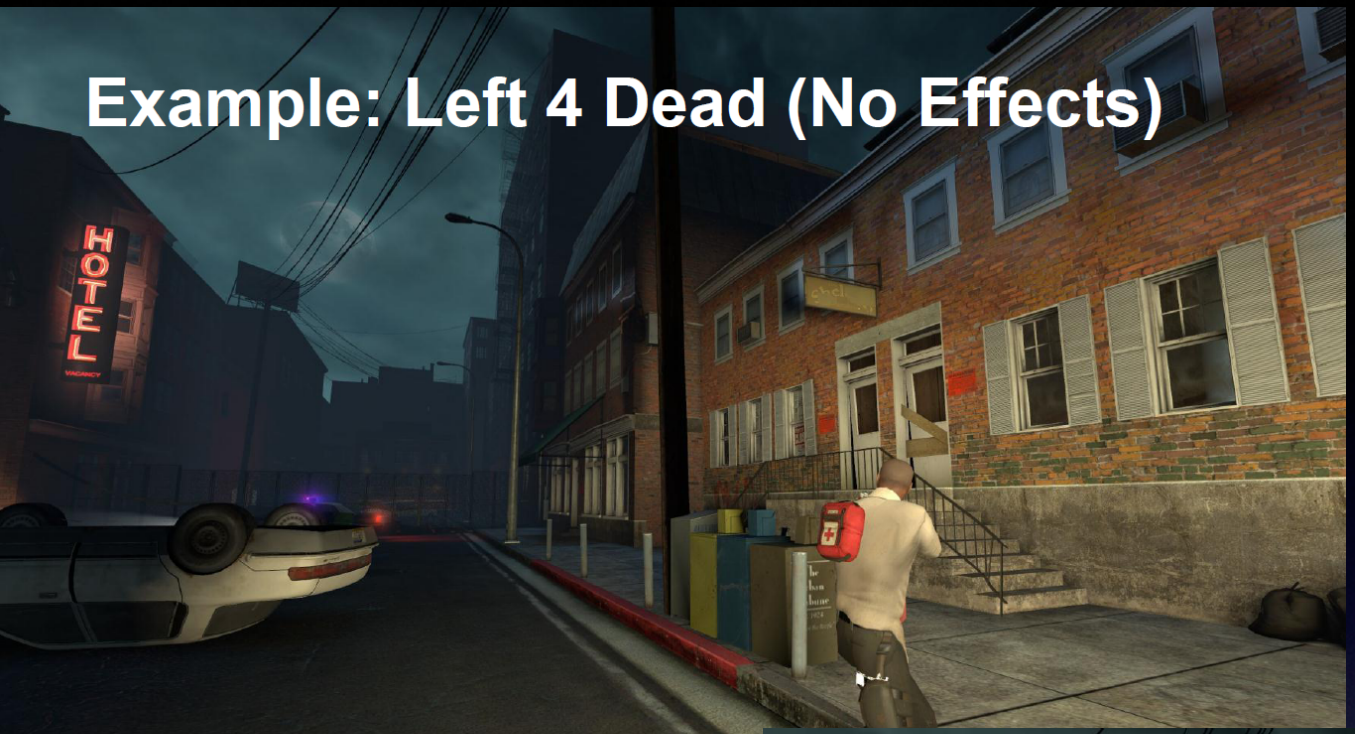


Image used with Valve's permission

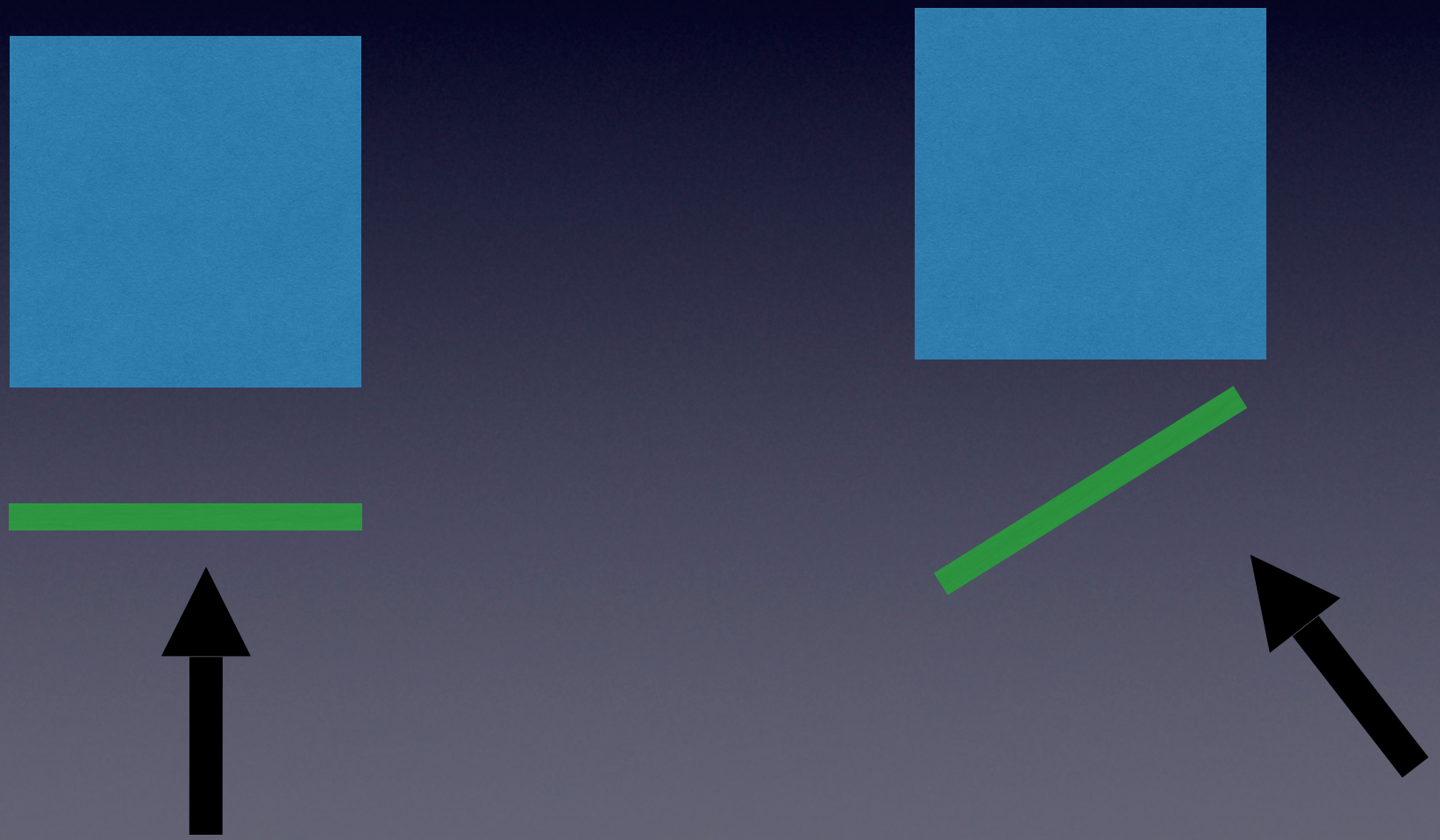
Example: Left 4 Dead (Color LUT)

Image used with Valve's permission

Color Enhancement for Video Games – Siggraph '09
(Used with permission from author)

Billboards and particles

- Quads that are aligned to the camera
  - Maybe only along one axis



Billboards and particles

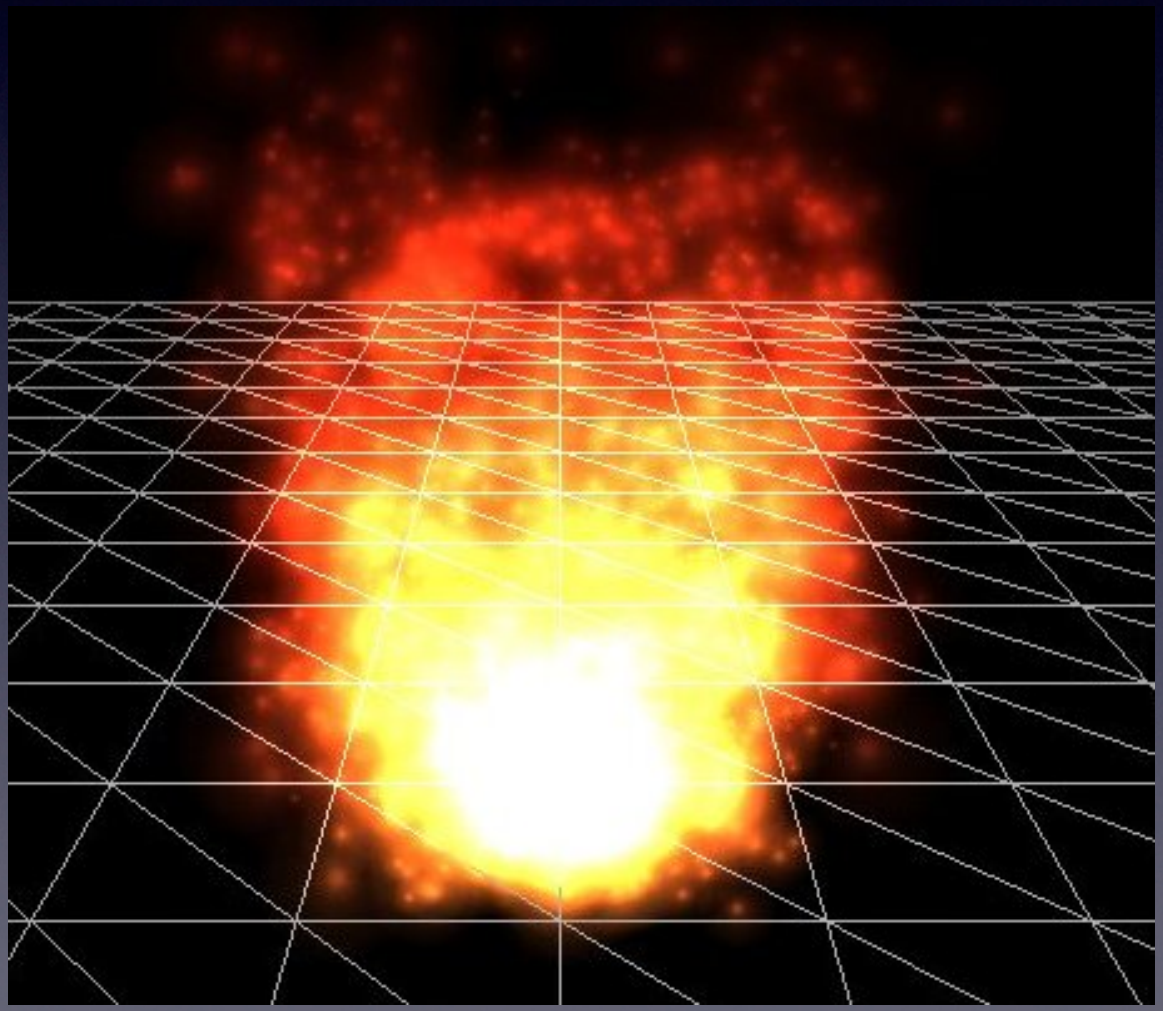
- Good for implementing spacial effects (volumetric lights, smoke)
- Can render using point sprites if fully aligned

Billboards and particles

- Align on one axis, good for foliage
- Can be expensive, tough to do alignment on GPU
  - Need lots of extra information in vertices

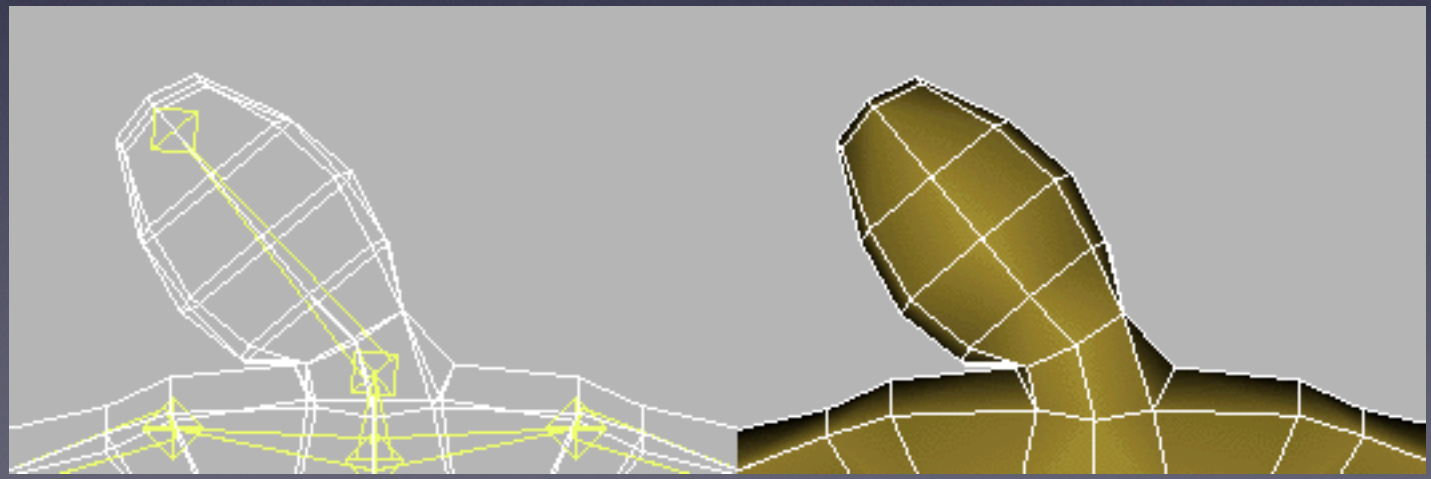
Particles

- Particles are generally rendered same as billboards, but have a complicated simulation
- (Almost) always done fully aligned



Skinning

- Skinning
  - Continuous mesh with vertices weighted to a skeleton
  - Complicated vertex transform that combines matrices
    - Each vertex holds n (usually 4) indices and weights into skeleton matrix list
  - Can be done on GPU, but can be done on CPU for performance reasons



Other important techniques

- Better approaches to handling transparency
  - Depth peeling
- Forward+ (clustered) rendering
  - Apply deferred-rendering-like techniques to forward rendering
  - Gives more flexibility than deferred techniques
- Physically-based rendering
  - Be as consistent as possible across different types of object
  - Sample real-world materials and lights
  - Compare to real world or non-real time rendering approach for accuracy

Display Latency

- Several sources of latency in rendering system
  - Controller, AI update, render, scan out, processing in display
- Lots of techniques for improving visual fidelity increase latency
  - Double/triple buffering
  - Motion compensation in TVs
- Crucially important with VR style hardware (Oculus Rift, et al.)

Wrap up

- Graphics programming is a synthesis of many disciplines
- Ultimate goal is to realize the visual design of the game
- Know the hardware
  - How do we process and render geometry
- Know the techniques
  - Many and varied
- Watch what others are doing
  - Follow the research
  - Look at other games
  - Check out the demos on Nvidia and ATI sites