
Physically-Based Reflectance for Games

9:15 - 10:15: Game Development

Dan Baker & Naty Hoffman



Game Development

- Game Platforms (Dan)
- Computation and Storage Constraints (Dan)
- Production Considerations (Naty)
- The Game Rendering Environment (Naty)



In this section, we will first discuss current and next-generation platforms on which games run. Next we will discuss the computation and storage constraints game developers face when developing for these platforms, the various production considerations which affect reflectance rendering in games, and finally the rendering environment within which reflection models are used in games.

Game Platforms



In this section, we discuss the platforms on which games run. First we shall give a brief overview of the rendering pipeline on the relevant platforms, and then we shall detail some of the characteristics of current and next-generation game platforms.

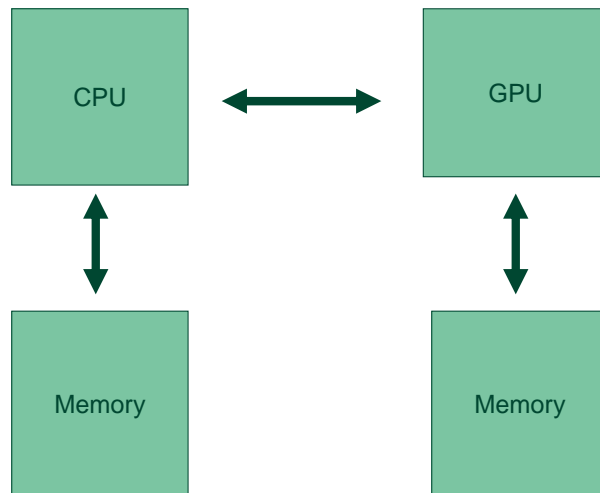
Modern Game Hardware

- Hardware is different, but trends are similar
 - Multiple CPUs
 - Custom GPU, but all similar
 - Programmable Shading
 - Bandwidth is getting pricier
 - Computation is getting cheaper



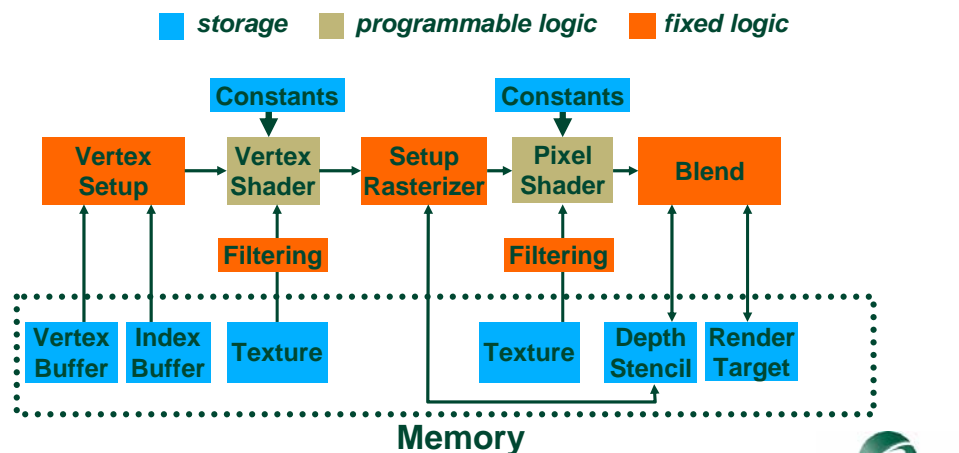
To some extent, the various game platforms are converging. They all use multiple-core central processing units and GPUs with similar architectures.

Basic Hardware Architecture



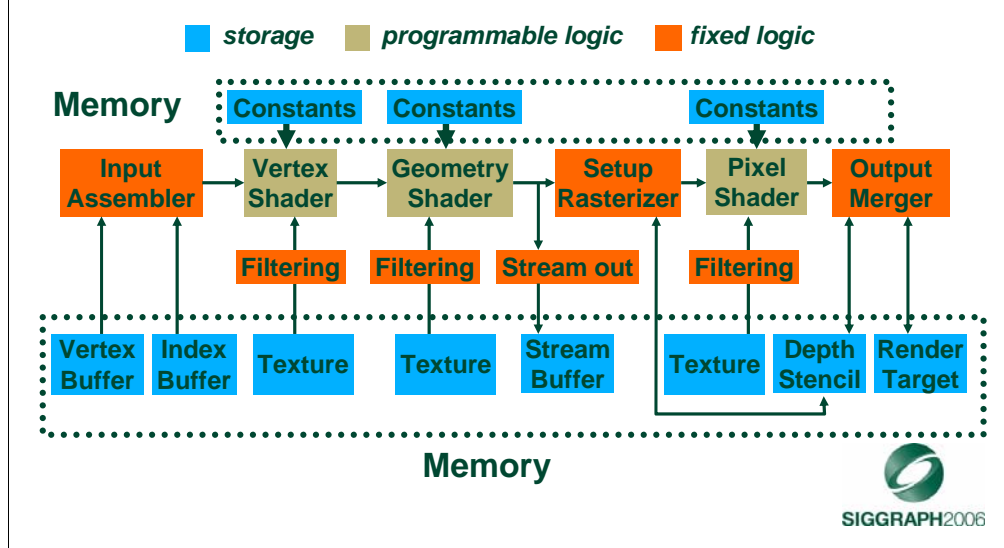
Basic block diagram of the hardware architecture of modern game platform. On some systems, like Xbox 360, the GPU and CPU memory is merged.

Shader Model 3 (Xbox360/PS3/D3D9)



This represents the main functional units in the GPU pipeline as it exists today. The Xbox 360, the Playstation 3 and current PCs all share this GPU architecture. For the purposes of this talk, we are most interested in the two programmable shader units. Currently there is a programmable vertex shader or vertex program, and a programmable pixel shader or fragment program. The rest of the pipeline is fixed-function (but highly configurable) logic.

Shader Model 4 (D3D 10)



This shader model is the one exposed in Direct3D 10. Hardware supporting this model should be available later in 2006. Most of the differences with future hardware are related to memory access and placement, however the introduction of the Geometry Shader fundamentally alters the kinds of lighting we may chose to perform, since we now can light at a triangle level if we chose to.

Game Development

- Game Platforms (Dan)
- Computation and Storage Constraints (Dan)
- Production Considerations (Naty)
- The Game Rendering Environment (Naty)



Now we shall discuss the computation and storage constraints game developers face when developing for these platforms, in particular as they pertain to rendering reflection models.

Computation and Storage Constraints

Platform	Xbox 360	Playstation 3	PC (2007)
CPU	3.2 GHz	3.2 GHz	3 GHz
	3 Cores	1 Core + 7 SPU's	2 Cores
GPU	shader 3	shader 3	shader 4
Memory	512 MB	512 MB	2048 MB
Bandwidth	21.2 GB/s	25 GB/s	6 GB/s
	(256 GB/s)	25 GB/s	60 GB/s



First, we discuss the computation constraints game developers face when developing for these platforms, in particular as they pertain to rendering reflection models. Next, we will discuss the relevant storage constraints.

The PS3 and PC both have two buses, a graphics bus and a CPU bus. The first number is the CPU bus, the second the GPU. The Xbox 360 has one bus, and high-speed embedded RAM for the back-buffer (for which the bandwidth number is given in parenthesis).

Some of these computational resources are reserved for operating system tasks (some SPU's on the PS3, one thread of one core on the Xbox 360, and a variable amount on the PC).

CPU Differences

- Total CPU power similar, but PC's take ~30% hit due to OS
- CPU and system memory are shared by ALL other game systems, physics, AI, sound, etc.
- Typically end up with far less than 50% of CPU for Graphics



Cache behavior is also a big problem with multiple CPUs and other tasks. Specifically, other processes and jobs can pollute the working set, vastly slowing down a processors performance. Latency varies from platform to platform, but is on the order of 100-500 cycles.

Polygon / Topology Shading

- In offline rendering, shading is typically performed on polygons (micropolygons)
- GPUs currently perform shading on individual vertices, and then pixel fragments
 - Polygon or topology computations done on the CPU
- Shader model 4 adds geometry shaders on GPU
- On PS3, these computations can be done on SPUs
- Performance characteristics are still unclear



SIGGRAPH2006

The shader model exposed on current GPUs allows for stream computation on individual vertices or fragments only, so any computations which are performed on triangles or otherwise require topology or connectivity information must be restricted to fixed-function hardware, or performed on the general-purpose CPU. In the near future, GPUs supporting shader model 4 will have the geometry shader which enables programmable triangle / topology computations. On the Playstation 3, the SPUs can fulfill a similar role.

Vertex Processing

- Most games are not vertex bound
- Much more likely to be memory bound
- Performance can be gained by moving operations from pixel to vertex shader



It is, however, easy to be vertex bound with high overdraw scenes. In these situations, a Z prepass blocks pixels from being shaded when not visible, but a vertex is always shaded, thus, in the future expect most vertex shaders to primarily transform geometry only.

Pixel Pushing

- ~500 instruction pixel shaders are here!
 - Well, almost...
- Instructions ~1 cycle, usually 4 way SIMD
 - 500 cycles per pixel at 800x600, 60fps = 14.4 billion pixel cycles / sec
- 500 MHz GPU with 32 shader cores
- Predicated on ability to do Z-Prepass
 - Not valid for translucent materials



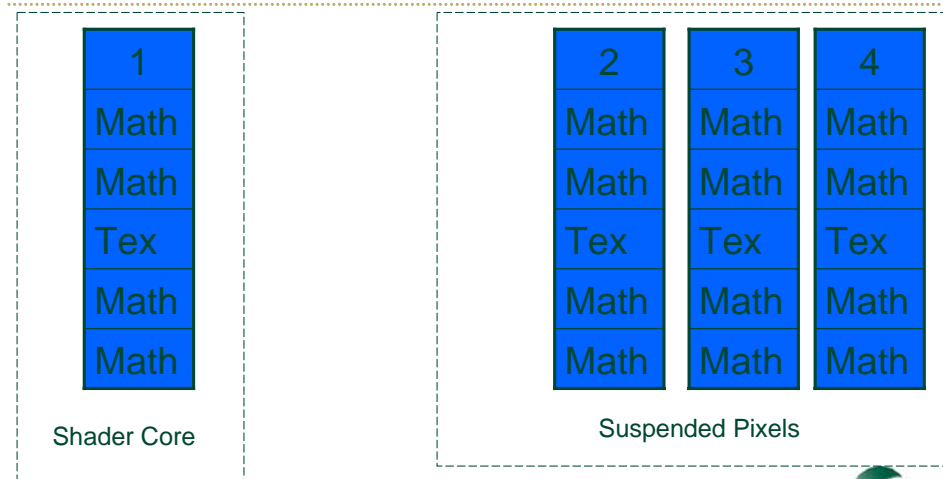
500 instructions is high end, but recall that with flow control, in a typical scenario not all of these instructions are executed. Still, 500 cycles is feasible on high end hardware.

But...

- Computation follows Moore's law, but memory does not
- Texture loads expensive – bandwidth is high
- Pixel shaders must execute together for gradients – so a cluster of pixels pays the price of the most expensive one
- Latency hiding mechanism isn't free

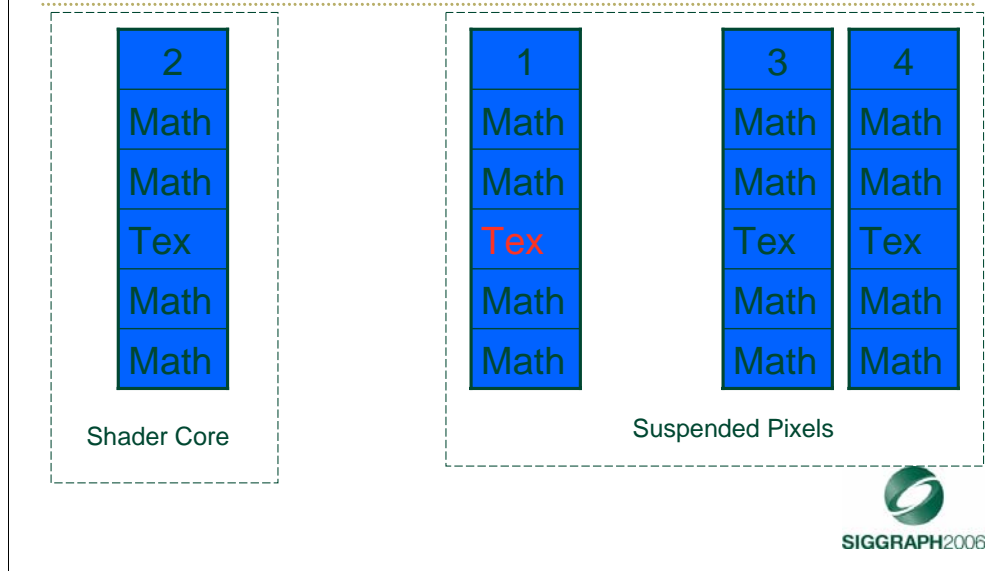


Shader Execution



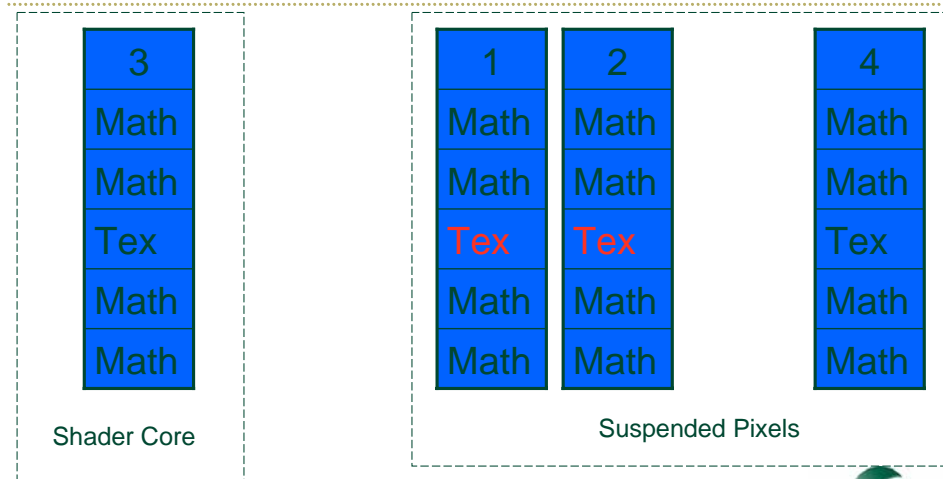
In this theoretical GPU, we have one shader core and 3 suspended pixels. The shader core starts by executing shader instance #1, which corresponds to a specific pixel on the screen.

Shader Execution



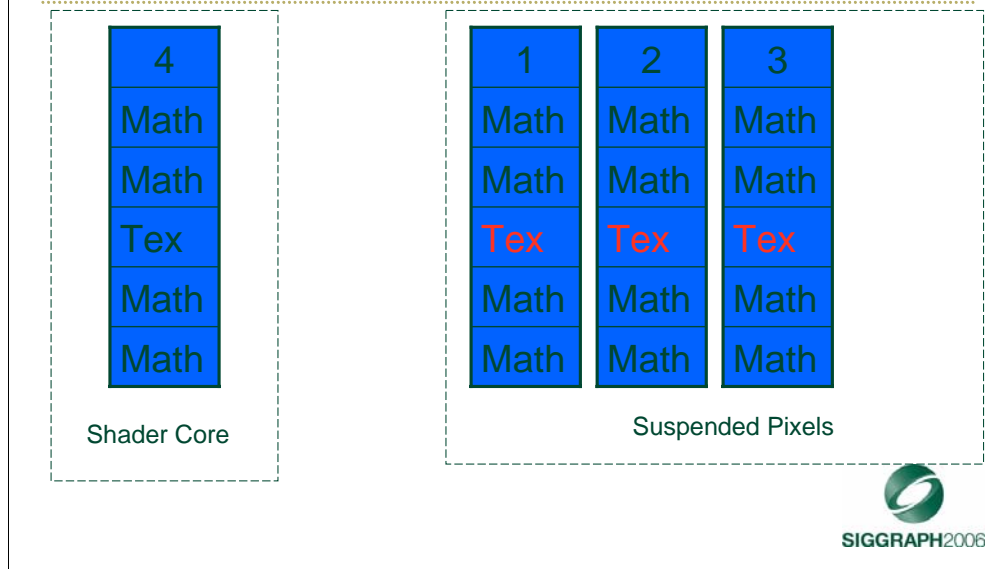
When the GPU hits the Texture instruction, it halts the pixel thread, and puts it in the suspended pixels. The program counter and register state is halted at that texture instruction. Shader instance 2 is now loaded into the core.

Shader Execution



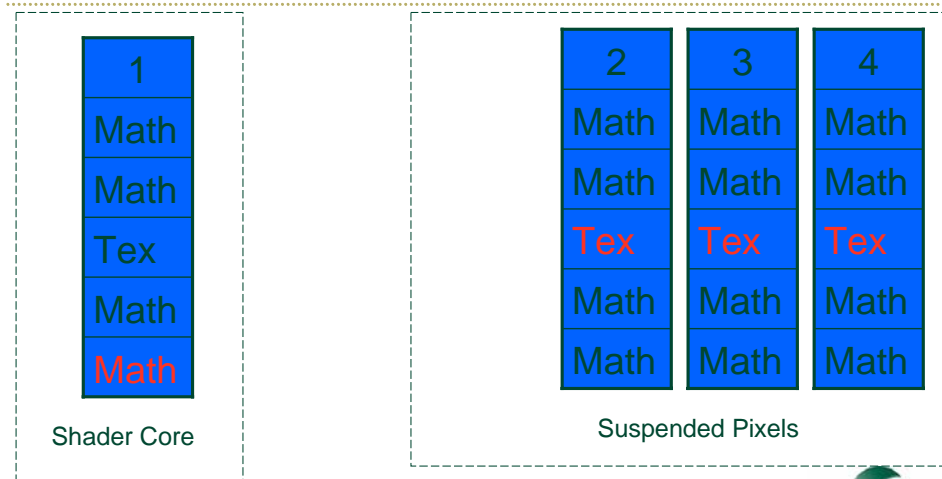
Now, shader instance 3 is loaded into the shader core, and executed until the texture instruction.

Shader Execution



Finally, the shader core gets to pixel instance #4, here we have 3 partially executed pixels waiting to be completed.

Shader Execution



Finally, the shader core reloads pixel #1, and the texture load data should now be ready, having been sent across the bus

Textures and Data

- Artists like to make very high resolution textures filled with parameters for each texel
- Bad as GPU costs are, texture costs are worse
- A character model might have four 2048x2048 textures - 64 MB of RAM!
- Could only have 6 characters at this rate!



Storage constraints are often more important in Console development than in PC development, since there is no guarantee of a hard drive, RAM is limited, and seek time on a DVD is slow. Additionally, high texture use causes bandwidth loads which can be a performance bottleneck.

Typical Texture Memory Budgets

- Last generation, typical character ~1-4 MB
- This generation action game: 10-20 MB (< 20 characters)
- For MMORPG, still about 1-4 MB because of the large number of unique characters
- For strategy games per-unit budget < 1 MB!
- Terrain – 20-40 MB, often just a window of a giant texture



These figures are composite numbers, but represent fairly typical scenarios found in real world situations.

Game Development

- Game Platforms (Dan)
- Computation and Storage Constraints (Dan)
- Production Considerations (Naty)
- The Game Rendering Environment (Naty)



Now we shall discuss the various production considerations which affect reflectance rendering in games.

Production Considerations

- The Game Art Pipeline
- Ease of Creation



First we will discuss the production pipeline for creating game art, in particular as it pertains to reflection models. Then we shall discuss issues relating to reflection models which affect the ease of art creation.

Modern Game Art Pipeline

- Initial Modeling
- Detail Modeling
- Material Modeling
- Lighting
- Animation



This is a somewhat simplified and idealized version of the modern game development pipeline. In reality, production does not proceed in an orderly fashion from one stage to the next – there is frequent backtracking and iteration. Also, not all stages are present for all types of data – the lighting stage is typically performed for rigid scenery and animation is typically performed for deforming characters, so both are almost never performed on the same model. We will discuss different types of game models later in the course.

Initial Modeling

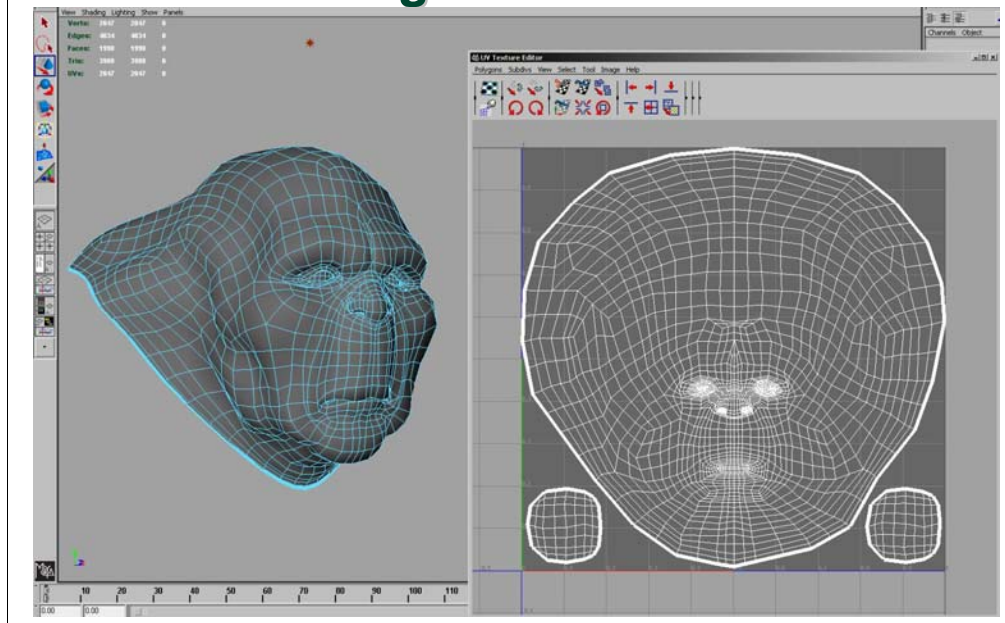
- Performed in a general-purpose package
 - Maya, MAX, Softimage, Lightwave, Silo
- Modeling geometry to be used in-game
- Creating a surface parameterization for textures (UV mapping)



The software packages in which the initial modeling is performed are traditionally the general-purpose workhorse of game art development, although many of their functions are being taken over by a new class of specialized detail modeling packages. The in-game geometry (as opposed to detail geometry which is only used to create textures) is edited here. This geometry needs to be amenable to level-of-detail (LOD), animation, surface parameterization and efficient in-game processing and rendering – these all impose constraints on how the geometry can be modeled.

Creating a surface parameterization is an important part of the artist' task. This parameterization is used for the various textures which will be attached to the model. Automatic tools for generating these parameterizations are rarely sufficient. In the past, these parameterizations included large amounts of repetition to save texture memory, now the growing emphasis on normal maps (which can rarely be repeated over a model) means that most parameterizations are unique over the model (or at least over half a model in the common case of bilaterally symmetrical models). This may also differ between scenery and character models.

Initial Modeling



Here we see an example of initial modeling of a game character's head in Maya. The in-game geometry can be seen, as well as the UV parameterization. Note that the parameterization is highly continuous. This is more important than low distortion, and is the reason why parameterizations are so commonly generated by hand. Discontinuities in the UV parameterization will require duplicating vertices along the discontinuity, and may also cause undesirable rendering artifacts in some cases if they are too prevalent throughout the model.

As mentioned before, the construction of the in-game model and its parameterization must obey multiple constraints and requires much skill and experience to perform well.

Modern Game Art Pipeline

- Initial Modeling
- Detail Modeling
- Material Modeling
- Lighting
- Animation



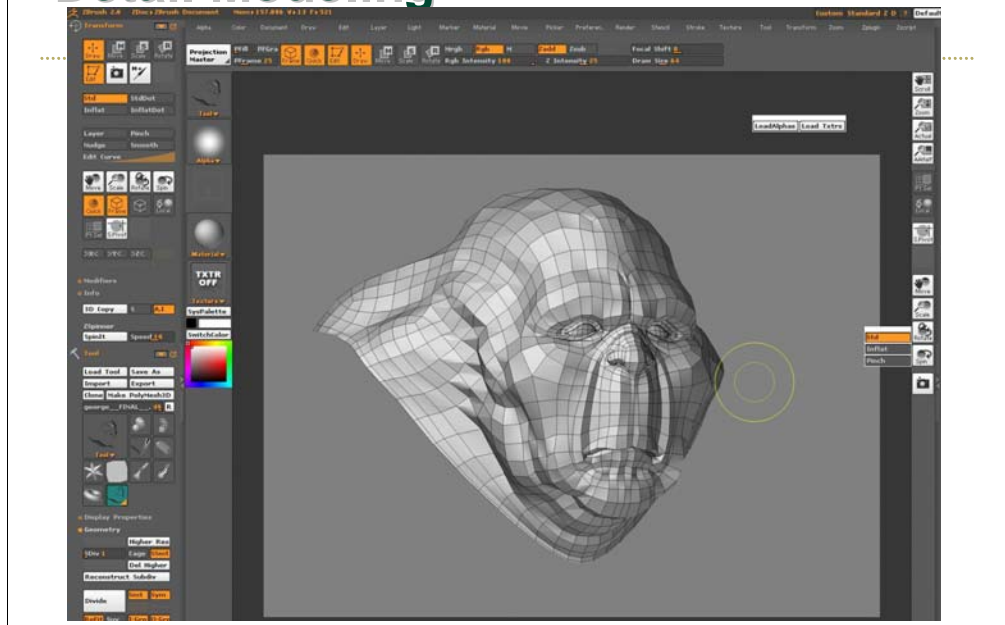
The next step is detail modeling. This has become more important in recent years.

Detail Modeling

- In the past this included only 2D texture painting, usually performed in Photoshop
- Now a new class of specialized detail modeling packages has arisen
 - ZBrush (the pioneer), Mudbox
 - Enable powerful sculpting / painting of surface geometry detail, and 3D painting of colors
- 2D texture painting apps still used

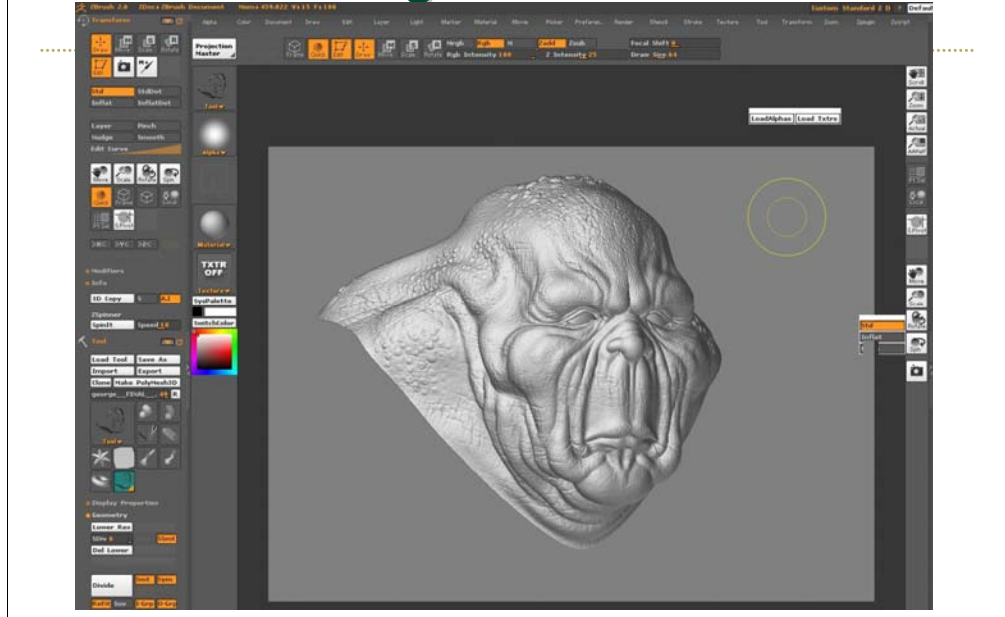


Detail Modeling



Here we see the same character, after he has been exported from Maya and imported into ZBrush. ZBrush was the first detail modeling package and is still by far the most popular.

Detail Modeling



The relatively low-resolution in-game model has its resolution dramatically increased and smoothed, then fine geometric details are painted / sculpted into the model. These details will not be present in the in-game geometry, instead they will be “baked” into textures of various kinds.

Detail Modeling



The most common type of texture used to represent the added geometric detail is the tangent-space normal map. This is a texture where the colors represent the coordinates of surface normal vectors in the local frame of the in-game (original, lower-resolution) mesh. This local frame is referred to as a tangent space. Detail modeling applications have various controls as to how the normal map is created and in which format it is saved. The generation of such a normal map can be seen as a process of sampling the surface normals of the high-detail model onto the surface parameterization of the low-detail model.

Here we see another reason why the surface texture parameterization created by the artist in the previous step is important. This parameterization is not only used to map textures such as the normal map onto the surface; it is also used to define the local surface frame into which the surface normals are resampled.

The normal map extraction is not always performed in the detail modeling application – sometimes the high-detail geometry is exported, and both it and the low-detail geometry are imported into a specialized normal map generation application, such as Melody from NVidia. General-purpose modeling packages such as Maya also provide normal map generation functionality.

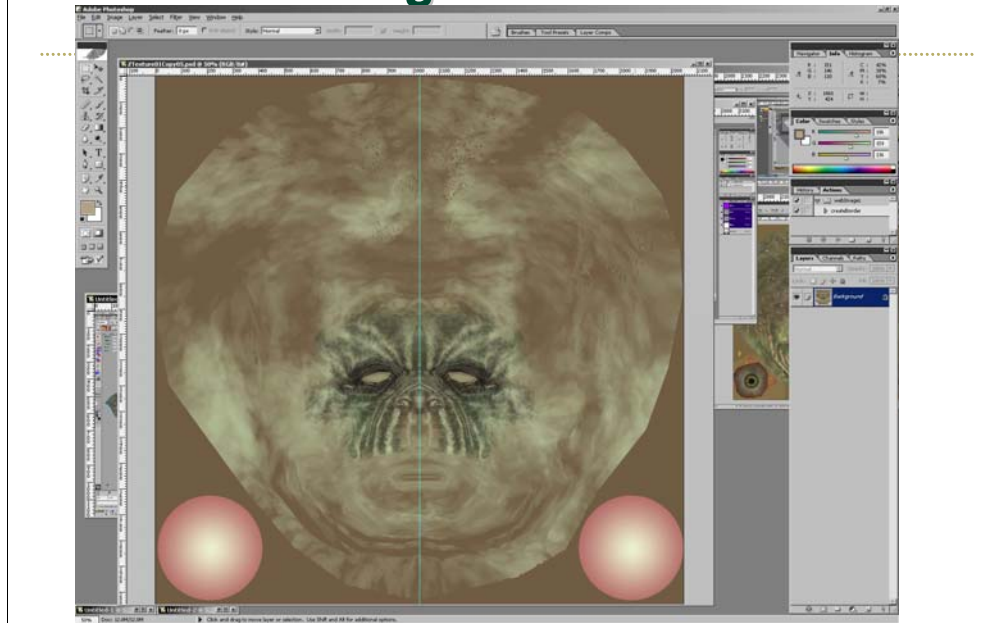
Detail Modeling



Here we see the resulting normal map texture which was generated from ZBrush. In this common representation, the red, green and blue channels of the texture represent the X, Y and Z coordinates respectively of the surface normal vector in tangent space. The -1 to 1 range of the coordinates has been remapped to the 0 to 1 range of the texture channels. The common purple color we see here is $RGB = \{0.5, 0.5, 1\}$, which represents a surface normal of $\{0, 0, 1\}$ in tangent space, namely a surface normal which is aligned with that of the underlying low-resolution surface. Divergences from this color represent areas where the surface normal diverges from that of the low-resolution surface.

Note that many other representations of the normal map are possible. For example, recently it has become quite common to only store the X and Y components of the normal, generating the Z component in the shader based on the fact that the normal vector is known to lie on the upper hemisphere.

Detail Modeling



Many other surface properties besides surface normals are stored in textures. The most common (and until recently, the only) such texture is the diffuse color or spectral albedo texture. We see here the color texture which in an early state of work in Photoshop (still by far the most common 2D texture editing application). More recently, many other surface properties are stored in textures. Any BRDF parameter can be stored in a texture, although if the parameter affects the final radiance in a highly non-linear manner, the hardware texture filtering will not produce the correct result. There will be more discussion on this subject later in the course.

A highly continuous parameterization will make it much easier to paint the texture, which is yet another reason why this is a desirable property.

Detail Modeling



Here is the result of applying the work-in-progress texture we just saw to the model in ZBrush. It is possible to work closely between the 2D texture painting application and the 3D detail modeling application, changes in one will be quickly mirrored in the other so the artist has immediate feedback of the results of their changes.

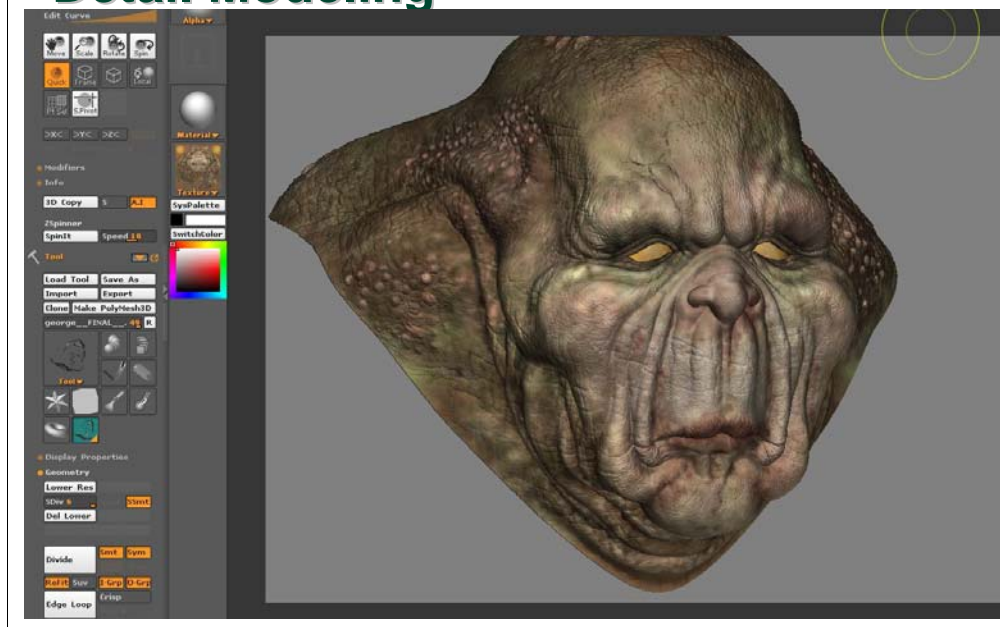
Detail modeling applications also allow painting surface colors directly onto the surface of the model, which is starting to become a popular alternative to 2D painting tools. In most cases, both options are used together since each type of tool is convenient for performing certain operations.

Detail Modeling



Here we see the final painted texture in Photoshop,

Detail Modeling



And in ZBrush.

Modern Game Art Pipeline

- Initial Modeling
- Detail Modeling
- Material Modeling
- Lighting
- Animation



The next step is material modeling. This is an important step for this course, and uses the results of the previous step.

Material Modeling

- This is usually performed back in the general-purpose modeling application
- The various textures resulting from the detail modeling stage are applied to the in-game model, using its surface parameterization
- The shaders are selected and various parameters set via experimentation (“tweaking”)



This is where the artist will choose from the available shaders, so if there are shaders supporting different reflectance models or BRDFs the artist will decide which is most applicable to the object. The entire object does not have to use the same shader, although there are significant performance advantages to maximizing the amount of geometry which can be drawn with a single shader.

Material Modeling



Here we see the in-game model with the normal map generated from ZBrush and the color map painted in Photoshop applied. The open dialog allows the artist to select the shaders used, and once selected, to select which textures are used and various other parameters which are not derived from textures. For example, in this shader although the diffuse color is derived from a texture, the specular color is a shader setting. However, this color is multiplied with a per-pixel scalar factor derived from the Alpha channel of the color texture. The specular power is also a setting and not derived from a texture. In general, deriving parameters from textures makes the shader more expressive, enabling the artist to represent different kinds of materials in a single shader (which as we have mentioned, has some advantages). On the other hand, increasing the amount of textures used uses up more storage, and may also cause the shaders to execute considerably more slowly.

Modern Game Art Pipeline

- Initial Modeling
- Detail Modeling
- Material Modeling
- Lighting
- Animation



The next step is lighting. This refers to “pre-lighting”, which is the pre-computation of lighting values.

Lighting

- Actually pre-lighting
 - Pre-computation of the effects of static lights on static geometry, usually with global illumination
- In the past, just irradiance data “baked” from a GI renderer or custom tool
- Now more complex data often used
 - A simpler example is ‘ambient occlusion’ data



This is only done for certain kinds of geometry and materials, and we will discuss it further in a later section of the course. One noteworthy detail about lighting tools is that they are sometimes used to generate “ambient occlusion” factors into vertices or textures for use in later rendering.

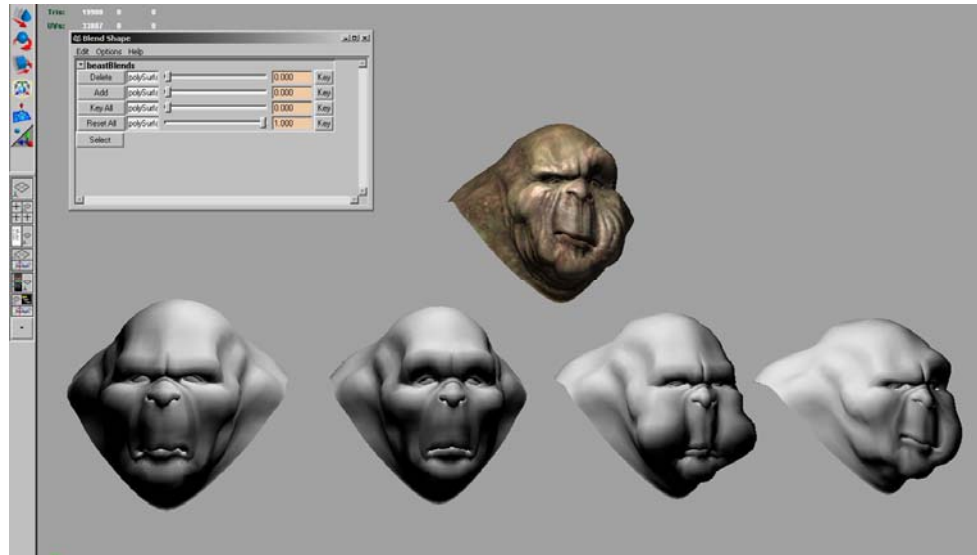
Modern Game Art Pipeline

- Initial Modeling
- Detail Modeling
- Material Modeling
- Lighting
- Animation



Finally, we have animation. This is only relevant for certain kinds of models (scenery is rarely animated). The two most common kinds of animation used in games are bone deformation and blend shapes (also known as morph targets).

Animation



Here we see the final character being animated in Maya, using blend shapes or morph targets. The animation is most commonly performed in the general-purpose modelign application.

Background and Foreground

- Game scenes are commonly separated into:
- Background
 - Mostly static
 - Scenery (rooms, ground, trees, etc.)
- Foreground
 - Objects which move and / or deform
 - Characters, items, (sometimes) furniture, etc.



We have mentioned several times that different kinds of models are sometimes handled differently in the art pipeline. These are the main types of game models.

Note that which category an object can belong to is somewhat dependent on the specific game; in one case a chair may be static and a background object, in another game the chair may be movable by the character and act as a foreground object.

Background

- Background is sometimes further separated:
- Unique, seamlessly connected parts
 - Terrain, walls / floors, etc.
- Instanced discrete objects
 - Rocks, trees, (sometimes) furniture, etc.



Note that furniture may be foreground or instanced background, it will usually not be unique background however.

Production Considerations

- The Game Art Pipeline
- Ease of Creation



Now we shall discuss issues relating to reflection models which affect the ease of art creation.

BRDF Parameters

- Preferably, BRDF parameters should clearly control physically meaningful values
 - Directional-hemispherical reflectance of surface reflection (“specular color”)
 - Albedo of body reflection (“diffuse color”)
 - Surface roughness



BRDF Parameters

- Parameters which map to physical quantities enable artists to explore parameter space
 - Changing parameters such as surface reflectance, body reflectance and roughness independently
 - Avoiding inadvertently creating materials which reflect significantly more energy than they receive
 - Mimicking the appearance of actual materials when desired



BRDF Parameters

- In previous game generations, these features were less important
 - Visuals tended to be more stylized
 - Lighting environments less complex
 - limited to a 0 to 1 range of values
 - Simplistic indirect / ambient light models
 - Games now use more realistic lighting and scenes, which requires more care in material creation



SIGGRAPH2006

We will show specific BRDFs and show how to make their parameters more physically meaningful in a later section.

Shift-Variance

- Also important for ease of use
- Very few game objects are composed of purely homogeneous materials
- At least some of the BRDF parameters must be easily derivable from textures



Shift-Variance

- It is also important to be able to combine the BRDF easily with normal mapping
- Anisotropic BRDFs may require even more control over the local surface frame per-pixel
 - “Twist maps” which vary the tangent direction
 - More on this later in the course
- Some BRDF rendering methods have difficulty supporting shift-variance



Since shift-variance is so important for artist control, BRDF rendering methods which preclude it will have limited applicability in games.

Game Development

- Game Platforms (Dan)
- Computation and Storage Constraints (Dan)
- Production Considerations (Naty)
- The Game Rendering Environment (Naty)



Finally in this section, we shall discuss the rendering environment within which reflection models are used in games.

The Game Rendering Environment

- Lighting Environments
- Bump, Twist and Relief Mapping



First we shall discuss the different kinds of lighting environments which are used to light reflection models in games. Next, we discuss issues relating to per-pixel detail methods such as bump, twist and relief mapping as they affect the implementation of reflection models.

Lighting Environments

- Game environments are lit by many different sources of light
 - Outdoor, daytime environments by sun and sky
 - Indoor environments by a combination of artificial lighting and sunlight
- Surfaces are lit by indirect as well as direct lighting



Lighting Environments

- The shaders used in games do not support arbitrary lighting
- Lighting is simplified in some way, usually into several terms which are handled in different ways by shaders
- The simplified lighting used by a shader is called its *lighting environment*



This is a repeat of a previous slide. We bring it here again because this division is very significant for how lighting is handled in games.

Background and Foreground

- Game scenes are commonly separated into:
- Background
 - Mostly static
 - Scenery (rooms, ground, trees, etc.)
- Foreground
 - Objects which move and / or deform
 - Characters, items, (sometimes) furniture, etc.



This is a repeat of a previous slide. We bring it here again because this division is very significant for how lighting is handled in games. Lighting is often handled quite differently on background and foreground geometry.

Prelighting

- Static and dynamic lights often separated
- Static lighting can be precomputed
 - On static (background) geometry: lightmaps or “pre-baked” vertex lighting
 - On dynamic (foreground) geometry: light probes
 - Can include global illumination effects
- Dynamic lights can be added later
 - Due to the linearity of lighting



A common trend in game development is to move as much computation as possible to “tools time” – pre-compute data using custom tools which is later used in the game. This usually involves a computation vs. memory tradeoff which must be carefully considered, but in many cases it is worth doing.

In many games, much of the lighting is assumed not to change (most games do not feature changing sun and sky light with time of day, and do not often allow changing interior lighting during gameplay). The affect of these static lights can be computed ahead of time and stored in different ways.

Prelighting and Global Illumination

- Since prelighting is computed ahead of time, global illumination is commonly used
- Similar to global illumination for rendering
 - Difference is final data generated (surface lighting information and light probes vs. screen pixels)



Prelighting on Background

- Usually (low-resolution) textures or vertex data
- Combined with (higher-resolution, reused) texture data
- Effective compression
 - Memory limits preclude storing unique high-frequency lighting data over the entire scene



The low-resolution lighting information is usually stored over the scene in a unique manner (no repetition or sharing, even between otherwise identical objects). Memory limitations therefore constrain this information to be stored at a low spatial frequency. For this reason it is very advantageous to combine it with higher-frequency data which can be repeated / reused between different parts of the scene (such as albedo textures or normal maps).

Prelighting on Background

- In the past, irradiance values
 - Combined with higher-frequency albedo data
 - Diffuse (Lambertian) surfaces only
- More complex lighting is becoming common
 - Pioneered by Half-Life 2 (Valve, 2004)
 - Directional information allows incorporating
 - High-frequency normals
 - Other (not general) BRDFs

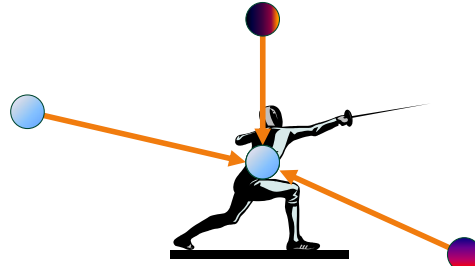


The use of simple prelighting (irradiance values only) dates back to around 1996 (Quake by id software). The use in games of prelighting incorporating directional information was pioneered by Valve with the release of Half-Life 2 in 2004. There separate irradiance information is stored for each of three surface directions which form an orthonormal basis. The actual surface normal is combined with this data in the pixel shader to compute the final lighting. This lighting information is used for diffuse lighting only in Half-Life 2, however extensions for certain types of specular reflectance have been suggested. In general, arbitrary BRDFs cannot be rendered with this type of lighting data.

Directional prelighting is currently an active area of research and development in the game industry (and hopefully outside it; see “Normal Mapping for Precomputed Radiance Transfer” by Peter-Pike Sloan, SI3D 2006).

Prelighting: Light Probes

- Approximate light field
- Sampled at discrete positions in scene
- Radiance or irradiance as function of direction
 - Spherical harmonics, environment maps, etc.
- Foreground objects interpolate, then apply



The foreground objects move throughout the scene, are not seamlessly connected to other objects and are usually small in extent compared to the spatial variations in the scene lighting. For these reasons, a common approximation is to treat the prelighting as distant lighting and compute it at the center of the object. This is usually achieved by interpolating between scattered (regular or irregular) spatial samples called *light probes*. Each light probe stores a function over the sphere, either of incident radiance (as a function of incident direction) or of irradiance (as a function of surface normal direction). The position of the object center is used to interpolate these to get an interpolated light probe to use for lighting the object.

Similarly to background prelighting data, this is usually applied to Lambertian surfaces, with occasional extensions for some limited types of specular reflectance. Commonly to handle highly specular reflections, high-resolution environment maps are used, with very sparse spatial sampling (perhaps a single map per scene).

Dynamic Lights

- Lights that change in position, direction, and/or color
 - Flashlight, flickering torch on wall, etc.
- In games, almost always point or directional lights



If a lights can change in position, direction (such as a flashlight carried by a character) or perhaps just in its color and intensity (such as a flickering torch on a wall) then it cannot be handled via prelighting.

The Reflection Equation with Point / Directional Lights

- Important case for real-time rendering
 - Lighting from infinitely small / distant light sources
 - Ambient / indirect lighting is computed separately
 - Point lights characterized by intensity I and position
 - Directional lights characterized by direction and I/d^2

$$L_{point_e}(\omega_e) = \sum_l \frac{I_l}{d_l^2} f_r(\omega_l, \omega_e) \cos \theta_l$$

This is a repeat of a previous slide. We bring it again here to discuss a common issue with how point / directional light intensity values are commonly represented in games. Note that arbitrary BRDFs are relatively easy to handle with this kind of lighting, since the BRDF just has to be evaluated at a small set of discrete directions.

Light Intensities

- Lambertian surface lit by single point light

- Radiometric version

$$L_e = \frac{I_l}{d_l^2} \frac{\rho}{\pi} \cos \theta_l$$

- “Game version”

$$L_e = \frac{i_l}{d_l^2} \rho \cos \theta_l$$

$$i_l \equiv \frac{I_l}{\pi}$$



Before we continue our discussion of point and directional lights, there is an issue which should be noted. The intensity values commonly used for point and directional lights (both in game engines and in common DCC applications such as MAX and Maya) are closely related, but not equal to the radiometric radiant intensity quantity. The difference will become clear if we examine a Lambertian surface lit by a single point light, both as a radiometric equation and in the form commonly used in game engine. Here we see that the light “game intensity” value (here denoted by i_l) differs from the light’s radiant intensity value (I_l) by a factor of π .

This factor needs to be noted when adapting reflectance models from books or papers for use in games. If the standard “game intensities” are used, then the BRDF needs to be multiplied by a factor of π (this multiplication will tend to cancel out the $1/\pi$ normalization factors which are common in many BRDFs). Game developers should also note the presence of this factor when using other kinds of lighting than point lights, since it may need to be removed in such cases. Another option is for game developers to use the radiometrically correct quantities at least during computation (using them throughout the pipeline could be difficult since lighting artists are used to the “game intensity” values).

Point Light Distance Attenuation

- Another divergence from radiometric math $L_e = \frac{I_l}{d_l} f_r(\omega_l, \omega_e) \cos \theta_l$
- Unlike the $1/d_l$ factor, f_d usually clamps to 1.0 $L_e = i_l f_d(d_l) \pi f_r(\omega_l, \omega_e) \cos \theta_l$
- Reasons:
 - Aesthetic, artist control $L_e = i_l(d_l) \pi f_r(\omega_l, \omega_e) \cos \theta_l$
 - Practical (0 at finite d_l)
 - Simulate a non-point light



Another way in which the math typically used in games for lighting with point lights differs from the radiometric math for point lights is in the distance attenuation function. A straight inverse square function is rarely used, usually some other 'distance attenuation' function is used. These functions are typically clamped to 1 close to the light (unlike the inverse square factor, which can get arbitrarily large values close to the light) and decrease with increasing distance, usually reaching 0 at some finite distance from the light (again unlike the inverse square factor).

Such 'tweaked' distance attenuation factors have a long history and are not unique to games, being in the earliest OpenGL lighting models and being used in DCC applications as well as other types of production rendering (such as movie rendering). Note that games will often use these distance attenuation functions in the prelighting stage as well as for dynamic runtime lights.

There are three main reasons for using a function other than a straight inverse square falloff:

- 1) More control by the artist to achieve a certain 'look' for the scene
- 2) Performance – if the light's effect drops to 0 at a finite distance, then it can be left out of lighting computations for objects beyond that distance
- 3) Actual light sources are not zero-size points and have some spatial extent. These lights will have more gradual falloffs than point lights, and this visual effect can be simulated by tweaking the falloff function.

Spotlights and Projected Textures

- Real light sources usually have some angular variation in emitted radiance
- This is often simulated by applying an angular falloff term, modulating the light by a projected texture, etc.



Usually this will result in the light only emitting radiance within a cone or frustum. A light which illuminates in all directions (also called an *omni light*) can also project a texture using a cube map, this can be useful to simulate some types of light fixtures.

Other Point / Directional Lights

- Many other variations are possible
 - Vary the direction to the light to simulate various shapes of light such as linear lights
 - Combine various distance and angular falloff terms
- In the end, these all boil down to different ways of computing i_l and ω_l for the rendering equation:

$$L_e = i_l(d_l) \pi f_r(\omega_l, \omega_e) \cos \theta_l$$



For an example of a complex point light variation used in film rendering, see “Lighting Controls for Computer Cinematography” (Ronen Barzel, Journal of Graphics Tools 1997).

The computation of the direction and intensity of the light are typically performed in the pixel shader, and the result is combined with the BRDF to compute the final lighting result (more details about this later in the course).

Shadows

- Dynamic point and directional lights cast sharp shadows
- To incorporate shadows, modulate i_l by an occlusion factor



Shadows

- Games usually use some variation on depth maps for shadows from dynamic lights
 - Stencil shadow volumes also (more rarely) used
- To simulate non-point lights, soft shadows are often desirable
 - Usually achieved by multiple depth map lookups



Shadows

- Other approaches used for light occlusion (though not for point / directional lights) include ambient occlusion terms and precomputed radiance transfer (PRT)
- Some of these will be discussed later in the course



Dynamic Texture Lighting

- We saw environment maps used as prelighting
- They can also be rendered dynamically
- Other types of texture-based lighting possible
 - Planar reflection maps



Environment maps are textures parameterized by directions on the sphere and used for distant lighting. A planar reflection map is a texture which represents the scene reflected about a plane, this would be used to render reflections on a planar surface. Other types of texture-based lighting are also used.

Combining Prelighting and Dynamic Lights

- Sometimes as easy as just adding them
- But there are often complications
 - Dynamic objects casting shadows from lights which were included in prelighting
 - Efficiency issues (divergent representations for prelighting and runtime lights may cause duplication of computation)
 - Different interactions with BRDFs



An example of the first issue is an outdoor scene with sunlight. We would want to include the sunlight in the prelighting on static objects to take account of indirect illumination resulting from sunlight, but we also want dynamic objects to cast shadows from the sun on static objects, which is difficult if the sunlight has already been accounted for in the prelighting. There are various solutions to this problem, and similar problems, which are outside the scope of this course.

An example of divergent representations for prelighting and runtime light is a foreground object lit both by a spherical harmonic light probe and several point lights. If the object is Lambertian, the spherical harmonic coefficients for the lights can often be combined with those from the light probe interpolation, thus resulting in improved efficiency.

An example of prelighting and dynamic lights interacting differently with BRDFs: again a foreground object lit by a combination of spherical harmonic light probes and point lights. If the object is not Lambertian, the reflectance models used for the prelighting and runtime lights will differ.

Light Architecture

- Usually games will have a general structure for the light environments supported
 - This will affect shaders, lighting tools, and data structures and code used for managing lights



Light Architecture

- Light architectures vary in the number of lights supported in a single rendering pass
 - Pass-per-light (including a separate pass for indirect / prelighting)
 - All lights in one pass: feasible on newer hardware, but potential performance and combinatorial issues
 - Approximate to N lights: like previous, but excess lights are merged / approximated



Pass-per-light architectures have the advantage of simplicity. Shaders only have to be written to support a single light, and only one variation needs to be written for each type of light. The object has to go through the entire graphics pipeline for each light affecting it, which can introduce significant overhead. On the other hand, efficiencies can be gained by only rendering the part of the screen covered by the current light. Certain shadow approaches (such as stencil shadows) require a pass-per-light approach.

All-lights-in-one pass architectures only process the object once, regardless of lighting complexity. There are two approaches to writing shaders for such an architecture: a single shader using dynamic branching (which is slower), or compiling a shader variation for each possible light set (which may cause a combinatorial explosion). Limits on the number of textures may restrict the number of lights casting shadows.

A common approach is to assume a standard light set (for example, an ambient term and three point lights) and in cases where the actual set of lights affecting an object is more complex than the standard set, merge the excess lights together, or discard them, or otherwise approximate the actual light set with a standard light set. This architecture can yield the highest (and more importantly, the most consistent) performance, but introduces complexity and possible temporal artifacts ('pops') in the approximation process.

The Game Rendering Environment

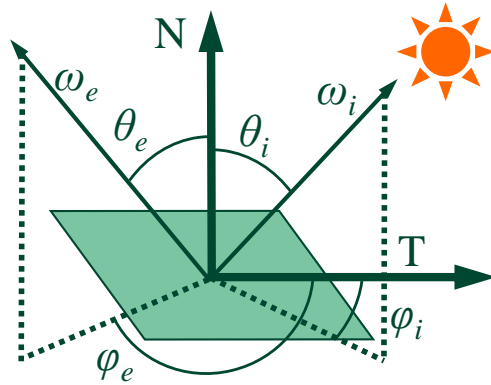
- Lighting Environments
- Bump, Twist and Relief Mapping



Now we shall discuss issues relating to per-pixel detail methods such as bump, twist and relief mapping as they affect the implementation of reflection models.

The BRDF

- Incident, excitant directions defined in the surface's *local frame* (4D function)



This is a repeat of a previous slide. This will be relevant for the discussion of bump, twist and relief mapping.

Local Frames for BRDF Rendering

- Isotropic shaders only require a surface normal direction or vector
- In the past, this has usually been stored on the vertices
- More recently, bump or normal mapping has become more common
 - Storing normal information in textures
 - Local frame now varies per-pixel



Local frames for BRDF Rendering

- Also, some anisotropic surface shaders (such as hair) utilize *twist maps*
 - Rotate the local frame per-pixel
- Although normal and twist maps create a per-pixel local frame, in practice computations can still occur in other spaces such as world space or the ‘unperturbed’ tangent frame



SIGGRAPH2006

Twist Maps

- Just as normal maps can have various representations, so can twist maps
 - Angle of rotation of tangent about normal
 - Cosine and sine of angle
 - Etc.
- Unlike normal maps (which are becoming ubiquitous) twist maps are relatively rare



SIGGRAPH2006

Relief Mapping

- Recent (and popular) development
- Texture coordinates are perturbed to account for relief or parallax effects
- Various flavors exist
- Affects the reflectance computation, since all texture data must use perturbed texture coordinates

