2015

Real-Time Graphics 1

SUMMARY BASED ON THE LECTURE SLIDES

BY MARTIN WINTER

CONTENTS

Introduction 8
First Generation 8
Second Generation 8
Third Generation 8
Fourth Generation 8
Fifth Generation 9
Graphics Pipeline 10
Application Stage 10
Geometry stage 10
Rasterization Stage 11
Display Stage 11
Display Synchronization 11
Multi-Threaded Rendering Pipeline 12
Warping 13
Unified Shader Model 13
Geometry Shader 13
Compute Languages 13
Modern APIs 13
OpenGL 14
Basic Concepts 14
Context 14
Resources 14
Object Model 14
Shaders 15
Vertex Shader 15
Rasterizer 15
Fragment Shader 15
Texture Mapping 16
Texture types 16
Planar Parametrization 16
Cylindrical/Spherical Parametrization 16
Box Parametrization 16

Texture Addressing 16
Aliasing 17
3D Texture Mapping 17
Multipass Rendering 17
RTT (Render to Texture) Method 18
Blending Method 18
Multitexturing 18
Light mapping 18
PTM (Projective Texture Mapping) 19
Texture Animation 19
Texture Compression 19
Environment Mapping 19
Cube Mapping (Box Parametrization) 19
Per-Pixel Lighting 20
Bump Mapping 20
Coordinate Systems 20
Reflective Bump Mapping 21
Bump Mapping issues 21
Parallax Mapping 21
Filtering Bump maps 21
Shading Models 22
BRDF 22
Light sources 22
Phong Shading 23
Fresnel Model 23
Lambert Shading Model 23
Microfacet Model 24
Blinn-Phong Model 24
Comparison between different Shading techniques 24
Geometric Attenuation 25
Anisotropic Reflection 25
Retro-Reflection 25
Textures and Framebuffers 26

Sampler objects 26
Framebuffer Objects (FBO) 26
Alpha – Channel 26
Blending 26
Deferred Shading 28
Deferred Rendering 28
Bloom 28
Depth of Field (DoF) 29
Non-photorealistic rendering 29
Deferred Rendering Advantages 29
Deferred Rendering Disadvantages 29
Image Space Special Effects 30
Billboards 30
Particle Systems 30
Fog 30
Bokeh 31
Motion blur 31
Lens Flare 31
Planar Reflections 31
Stencil Buffer 32
Transparency 32
Depth Peeling 32
Per-Pixel Linked Lists 32
Shadowing with environmental lighting 34
Ambient Occlusion 34
SSAO 34
SSDO 34
Shadows 36
Real-Time Shadows 36
Static Shadows 36
Approximate Shadows 36
Planar Projected Shadows 36
Shadow Mapping Alogrithm 37

Aliasing 37
Shadow map 37
Shadow Volumes 38
Silhouette Detection 38
Stencil Shadow volume | Depth pass 39
Geometric Soft Shadows 39
Global Illumination 40
Radiometry 40
Path Tracing 40
Two-Pass Global Illumination 40
Radiosity 40
Photon Mapping 40
Render Cache 41
Real-Time global illumination 41
Instant Radiosity 41
Incremental Instant Radiosity 41
Imperfect Shadow Maps 41
Reflective Shadow Maps 41
Precomputed Radiance Transfer 42
Diffuse Light transfer 42
Gibbs' Phenomenon: Ringing 43
Prt Summary 43
Voxel cone tracing 43
Spatial Data Structures 44
Regular Grid 44
Bounding Volume Hierarchy 44
Quadtree 44
Octree 45
Binary Space Partitioning (BSP) Tree 45
Generation of BSP trees 45
Painter's Algorithm 45
kd Tree 45
Bintree 46

Topological Data Structures 46
Bottom-Up Bounding Volumes 46
Scene Graph 46
Intersection | Collision 48
Analytical 48
Geometrical 48
Separating Axis Theorem (SAT) 48
Rules of Thumb for Intersection Testing 49
Ray / Triangle 49
Point / Plane 49
Sphere / Plane | AABB / Plane | Ray / Polygon 50
Volume/Volume Tests 50
View Frustum Testing 50
Dynamic Intersection Testing 51
Dynamic Separating Axis Theorem 51
Collision Detection 51
Collision Detection with Rays 51
Dimensionality Reduction 51
Collision detection for many objects 52
BVH Building Example 52
Pruning 53
Sweep-and-Prunge Algorithm 53
Visibility 54
Visibility culling 54
Visibility from a point 54
Visibility from a Region 55
Umbra and Penumbra 55
Occlusion culling from a region 56
Potential Visible Set (PVS) 56
Occlusion culling in Practice 56
Occlusion culling – general idea 57
Occluder shadows 57
Occluder selection 57

Lazy Update of SVDS 57
Approximate front-to-back sorting 57
Hardware Occlusion Query 57
PVS Precomputing for Regions 58
Cells and Portals 58
Occluder Shrinking 58
Shafts: Occlusion tracking 58
Blocker Extention 58
Extended projections 59
Level of Detail 60
Traditional Approach 60
LOD Selection 60
Static LOD Selection 60
Reactive LOD Selection 60
Predictive LOD selection 60
LOD switching 61
LOD Creation 61
LOD by Subdivision Surfaces 61
LOD by Shading and Rendering 61
LOD by Geometric Simplification 61
Polygonal meshes 61
Manifold Meshes 61
Topological Validity 62
Geometric Validity 62
Simplification for LOD 62
Simplification operators 62
General Geometric Replacement 63
Simplification Frameworks 63
Local simplification Frameworks - variants 63
Global Geometry Simplification 63
Vertex Clustering 64
Simplification Objectives 64
Error measures 64

Quadric Error Metric 64
Optimal Vertex Placement 65
Boundary Preservation 65
Image-driven simplification 65
Appearance-Preserving Simplification 65
LOD Representation Frameworks 65
Discrete LOD 65
Continuous LOD 65
Terrain LOD 66
Regular Grids 66
Tins 66
LOD Hierarchy Structures 66
HDR-Rendering 68
Tonemapping 68
Operators 68
Offline vs Real-time Tonemapping 69
Light bloom 69
Virtual Textures 70
Wrap-Around updates 70

INTRODUCTION

Real-time Rendering is all about delivering rendering effects like *shadows, advanced shading, culling, bloom, texturing* and many more, and all of that in "real-time" (meaning 60 fps or more in today's standards). Real-time graphics started out with early research being done in the field and first deployed in **flight simulation**. Later on it found use also on **SGI workstations** and later came to the PC with dedicated graphics processors. Both PC and SGI were developed side by side, but were defined through their respective feature set, where early SGI implementations had **hardware geometry** but no **texturing** while the PC implementation boasted **hardware texturing** but no geometry.

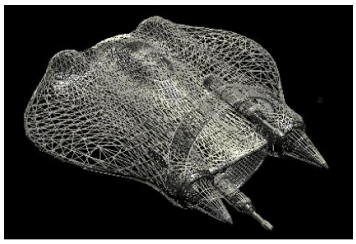
Advancements in Real-time graphics come primarily on three categories

- **Performance**: Triangles/sec, Fragments/sec
- **Features**: Texture Mapping, Programmable Shaders …
- **Quality**: Numeric Precision, Supersampling, Antialiasing …

Performance in software is generally **inversely proportional** to the **algorithm complexity**, Hardware on the other hand is different, as performance is either **invariant to complexity** (e.g. texture-filtering) or **falls of catastrophically**, if the need arises for software fallbacks.

In Hardware, **Pipelining** can lead to *free* features like geometric transformations, where functionality is implemented directly in hardware and can be performed very efficiently, rendering without it may be a lot slower.

FIRST GENERATION



The first generation of graphics hardware used mainly the **wireframe models**, which was mainly used prior to 1987, it was possible to do certain operations on **Vertices** (transform, clip, and project), on **Fragments** (color interpolation) and use a **frame buffer**.

SECOND GENERATION

The second generation of graphics hardware used **shaded solids**, this was mainly between 1987 and 1992, it added operations for **Vertices** (lighting calculations), per **Fragment** (depth interpolation, triangles) and for the **frame buffer** (depth buffering and blending).

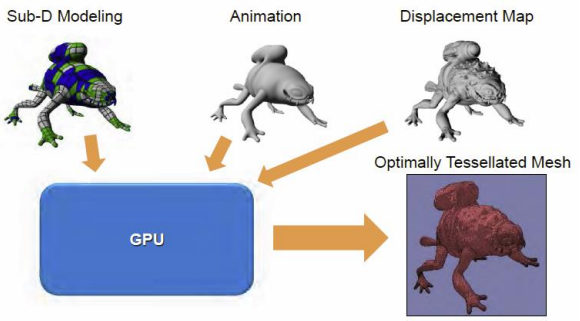
THIRD GENERATION

The third generation introduced **texture mapping**, this was used between 1992 and 2000 and also introduces operations per **Vertex** (texture coordinate transformations) and per **Fragment** (texture coordinate interpolation and texture evaluation and filtering).

FOURTH GENERATION

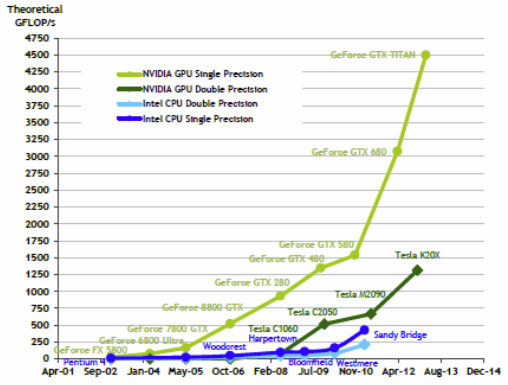
The fourth generation enabled **programmable shading**, which replaced parts of the fixed function pipeline, **DirectX 8/9** introduced further shaders like **Vertex Shaders** and **Fragment Shaders**.

With **DirectX 10/11** additional possibilities were added through the **Unified Shader Model** as well as **Geometry Shading** and **Tessellation** (receives a low-detail mesh as input and outputs a high-detail mesh



FIFTH GENERATION

The fifth generation introduced **general-purpose computing** on the GPU which was enabled by the unified shader architecture and utilizes the massive parallelization on the graphics card.



This enables a lot of **GPGPU applications** to use the graphics card, for example to do ray tracing, photon mapping, physics simulation, scientific computation and many more.

This is trend also holds due to the huge performance increases that GPUs typically get with every new generation, as the GPUs computing power increases much faster compared to the CPU.

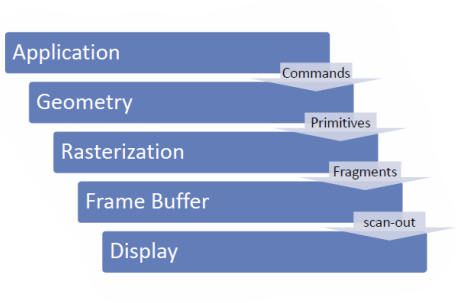
Those performance increases can be tracked by analyzing synthetic benchmarks, which provide peak numbers that only show trends, as those are typically not achievable in practical situations, as a lot of small problems influence the results like

- State changes, pipeline stalls
- CPU or driver issues
- Non-optimal geometry arrangement
- Bandwidth
- Cache inefficiencies

Today's benchmarks mainly look at framerates in games instead of raw textured triangle counts or shader instruction counts, this is also due to the fact that development nowadays is mainly driven by games, which is a thriving and ever growing industry.

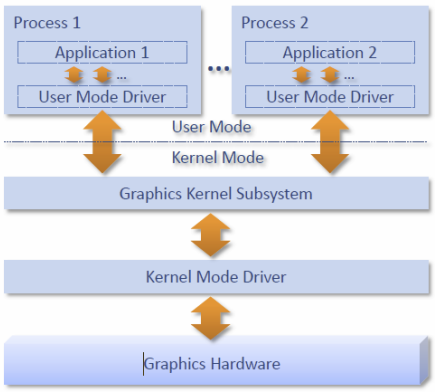
GRAPHICS PIPELINE

The goal is to achieve a computer-generated imagery (**CGI**) of complex 3D scenes in real-time, which is *computationally extremely demanding* on current setups and requires specialized hardware. Most of **RTG** is based on the **rasterization** of graphic primitives, which is implemented **in hardware** on the GPU. The output of the rasterization process are fragments, which then get merged into pixels.



APPLICATION STAGE

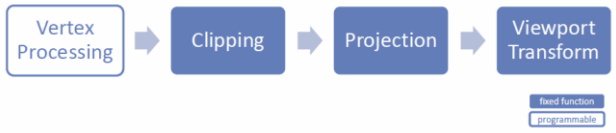
This stage is used to generate the **database**, which is usually done only once, it can be loaded from disk or generate algorithmically and from it acceleration structures can be built. Also in this stage reside the **input event handlers** and also the main loop is executed from here, which issues graphics commands that are then processed by further stages. The **result** therefore is a **stream of graphics commands**, consisting of primitive specification, resource management and graphics state modifications.

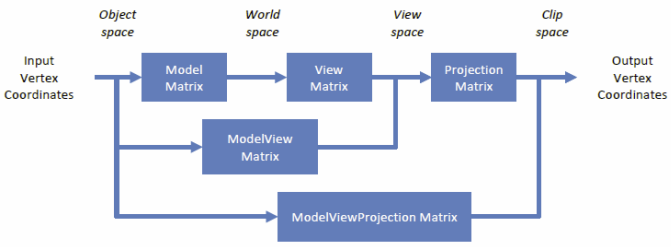


Typically, there is a large **user mode graphics driver** (uses fine grained memory management), which translates graphics commands into instructions for the hardware, prepares the command buffers and provides those to the **graphics kernel subsystem**, which now can schedule access to the hardware itself through a smaller **kernel mode graphics driver**. This actually submits the command buffers to the hardware itself and uses a coarse grained memory management.

The user mode graphics driver is also the part were newer APIs like DX12, Mantle or Vulkan try to provided lower-level and more efficient implementations.

GEOMETRY STAGE





This stage performs all the geometry processing, starting with the **Vertex Processing**. Here an **input vertex stream**, which is composed of several, arbitrary vertex attributes, is transformed into a **stream of vertices mapped to the screen** by the **vertex shader**, so object space vertices are transformed into screen space vertices.

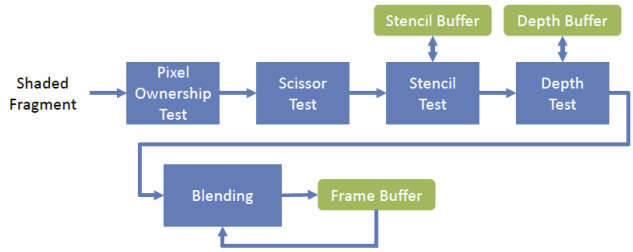
Further geometry stages include

- **Clipping:** Primitives not entirely in view are clipped to avoid projection errors
- **Projection:** The clip space coordinates are projected onto the image plane, those coordinates are called normalized device coordinates
- **Viewport Transform:** Maps resolution-independent normalized device coordinates to a rectangular window in the frame buffer, the so-called viewport

RASTERIZATION STAGE

The rasterization stage has multiple tasks which include

- **Primitive assembly:** Backface culling and primitive setup for traversal
- **Primitive traversal:** Here the sampling occurs, where triangles are turned into fragments and the individual vertex attributes are interpolated
- **Fragment Shading:** The Fragment Colors are computed from the given interpolated vertex attributes, here also textures are applied and lighting calculations can be done.
- **Fragment Merging:** From those fragments, the pixel colors can be computed, and depth test, blending … is performed, as multiple primitives may cover the same pixel and the final pixel value needs to be calculated.



Different rules may apply for different primitive types, this needs to be **non-ambiguous** to avoid artifacts.

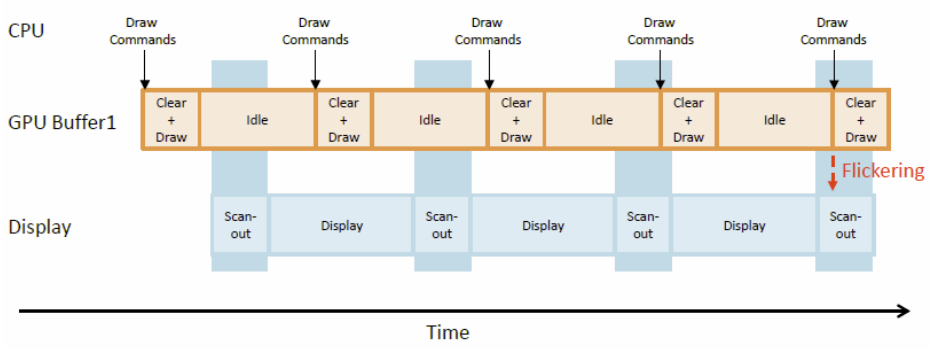
DISPLAY STAGE

At the display stage, what remains is **gamma correction** and today a digital scan-out, which is available in different formats, with various bit precision and different buffer techniques.

DISPLAY SYNCHRONIZATION

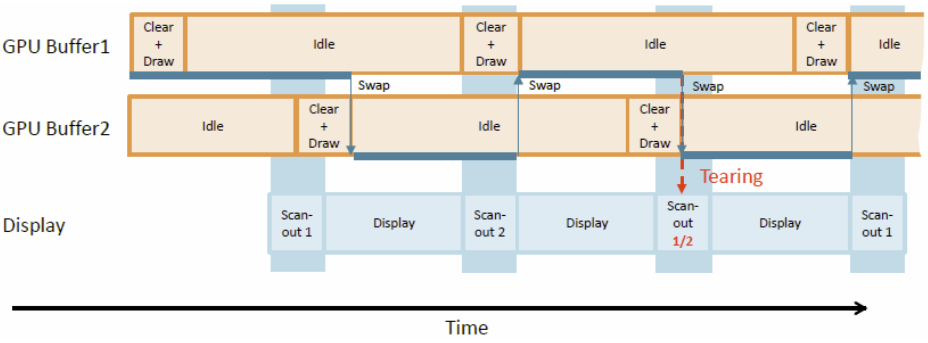
There is **need for synchronization** when granting access to the frame buffer to the GPU or the display, as it should not be possible to change the frame buffer, while a scan-out is occurring. Therefore it is important to **provide proper buffering** in the whole graphics pipeline.

SINGLE BUFFERING



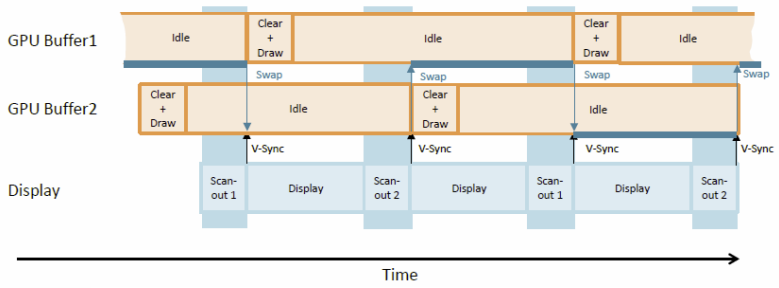
DOUBLE BUFFERING / V-SYNC

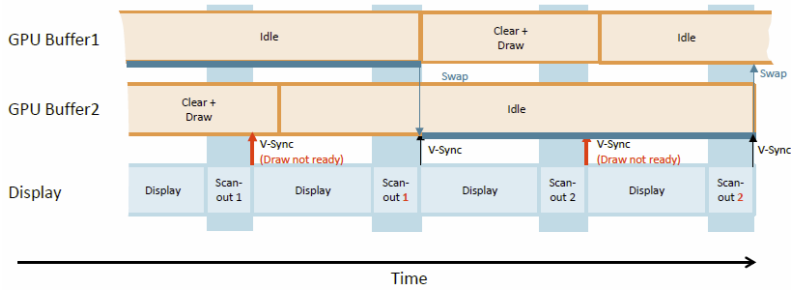
Here, the front buffer is used for the scan-out, while the *swapBuffers* functionality changes between front and back buffer, so **no flickering** but **tearing artifacts** occur.



DOUBLE BUFFERING WITH V-SYNC

When **rendering is fast,** the rendering itself is limited by the display refresh rate, swaps only occur after a scan-out is finished. It also adds **additional latency** at maximum of one frame time.

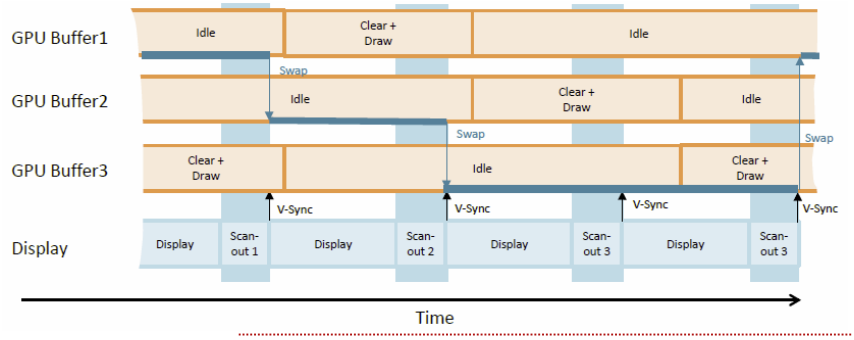


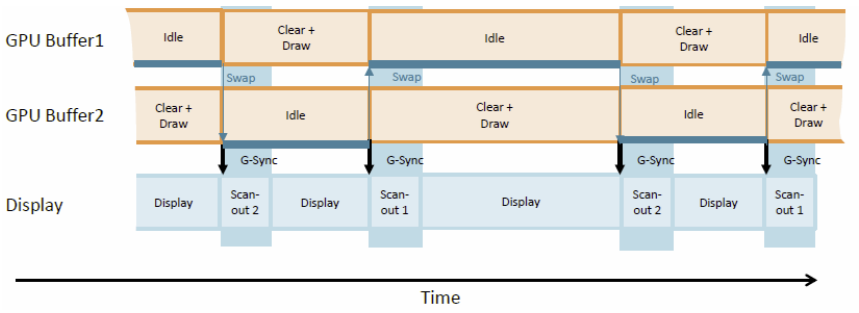


When **rendering is slow**, frame rates may only be integer fractions of the highest possible display refresh rate, so for a display refresh rate of 60 Hz this means 60, 30, 15, 12 and so on. This can be counteracted by using **Adaptive V-Sync** which automatically turns of V-Sync if the rendering is to slow.

TRIPLE BUFFERING WITH V-SYNC

Triple buffering itself only makes sense in combination with V-Sync, here, when **rendering is slow**, the third buffer allows for **continuous drawing**, while when **rendering is fast**, the two back-buffers can be alternated, which may lead to **wasted frames**.



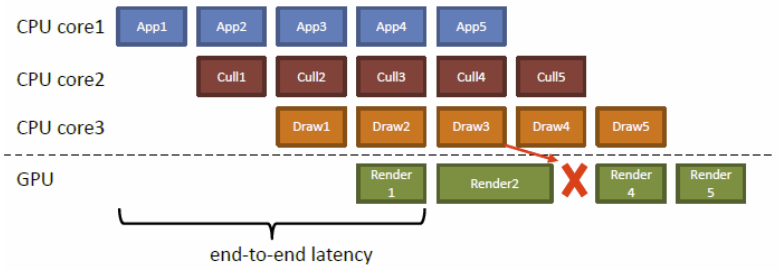


G-SYNC / FREESYNC

The goal here is to trigger the display scan-out by the GPU, this requires additional display hardware features, the two competing technologies are G-Sync (NVIDIA) and FreeSync/AdaptiveSync(AMD).

MULTI-THREADED RENDERING PIPELINE

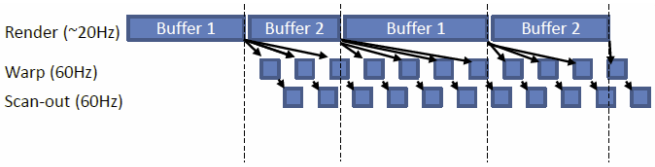
The **Pipelining** principle is used heavily, also together with the CPU, where individual stages can be performed on different cores to enable pipelining.



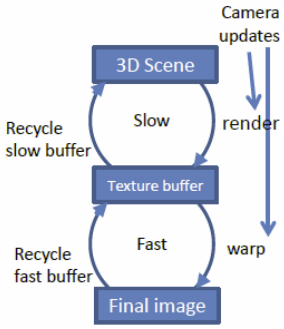
The **App** stage is responsible for simulating the 3D-world, the **Cull** stage determines the objects in the view frustum, the **Draw** stage issues the OpenGl commands finally to the driver, everything must work at target frame rate, otherwise frames may be lost. The goal is always to reach minimal latency.

WARPING

A technique to increase performance is warping, where to passes are rendered, once to a texture and the second pass the does a perspective texture mapping with new camera parameters.



Using this we can get **Multi-Framerate Rendering**, where rendering and displaying is actually decoupled.



In a **slow pass**, the image is rendered to a texture (which is larger than the actual viewport) and in a **fast pass**, the latest rendered image is warped using the latest camera parameters. This uses perspective texture mapping as discussed before. The Oculus Rift uses a similar technique called **Asynchronous Warping** for the head tracker.

UNIFIED SHADER MODEL

The **unified shader model** is available since shader model 4.0 and is implemented as an **ALU**, which provides the same instruction set and capabilities for all shader types. This enables **dynamic load balancing** and also provides floating point or integer operations everywhere.

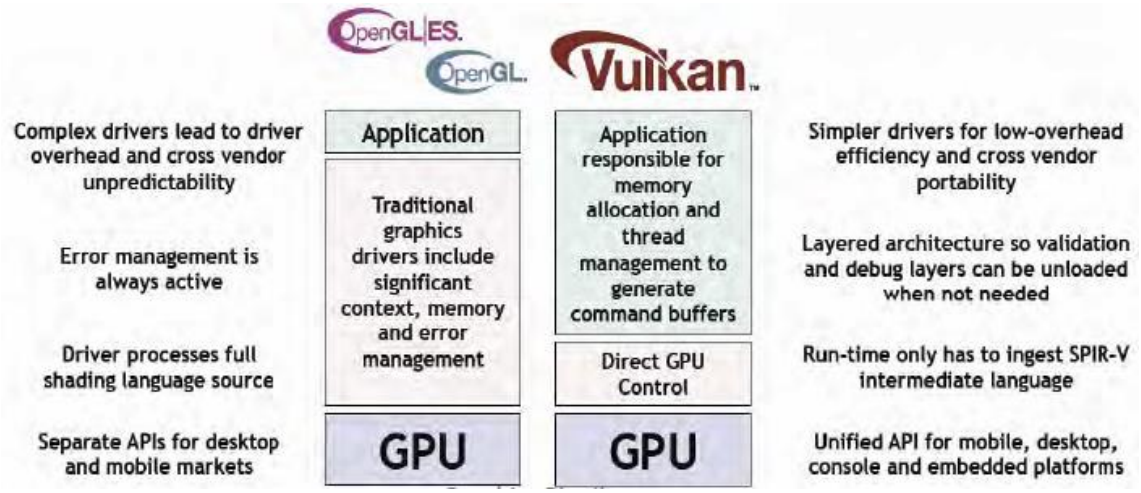
GEOMETRY SHADER

The **Geometry Shader** resides between the vertex and fragment shader and can generate primitives dynamically, this can be used to procedurally generate geometry (e.g. growing plants) but it can also produce particles, shadow volumes via extrusion or other displacements.

COMPUTE LANGUAGES

With help of the unified shader architecture, it is possible for **GPU Compute languages** to access the unified shader cores through a C-like language, one is **CUDA**, which runs multiple threads (one per shader) and accesses shared memory and also **OpenCL**, which provides a massively parallel programming API which runs on the CPU and GPU.

MODERN APIS



OPENGL

OpenGL is a **low-level graphics API** specification which defines an *abstract rendering device* and a *set of functions* which may operate on this device. It is **not a library** as the implementation is platform depended, although the interface is not. It is a so-called **immediate mode API**, which means that it issues drawing commands and has no concept of permanent objects. It comes as part of the graphics driver or as a runtime library built on top of the driver and can be initialized through a platform specific API like WGL (Windows), GLX (Unix/Linux) or EGL (mobile devices).

BASIC CONCEPTS

OpenGL operates in a certain **context**, in which **resources** are kept. It may define **objects**, that are represented through object names and has a state that if bound to the context is also mapped into the context's state. It may be bound to so-called bind points (also called targets).

CONTEXT

The context represents an instance of OpenGL, multiple of those may exists in a process and resources can be shared between those. Each thread has a *current context* and there exists a **one-to-one mapping**, so only one current context per thread and the context is only current in one thread at the same time. All of the OpenGL operations then **work in the current context**.

RESOURCES

Resources act as **sources of input and sinks for output**, this includes buffers (linear chunks of memory), images, state objects and much more.

OBJECT MODEL

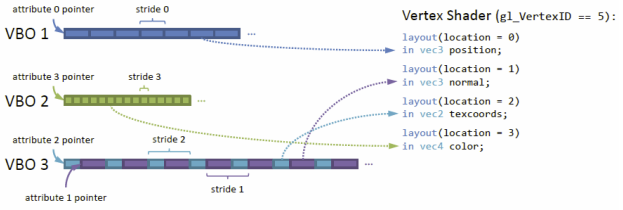
OpenGL is object-oriented in a strange way, where instances are **identified by name** (like an unsigned integer handle) and **commands work** on so-called **targets**. Objects can be bound to those targets, whereas the issued command will then act upon this object. Commands are reminiscent of methods and the target is similar to the type in classical object-oriented programming.

To **bind** an object to a target essential means **activating** the object, this means that object become *current* for that target. Objects are also first created when first bound. **Exceptions** to this are **shader** and **program objects**.

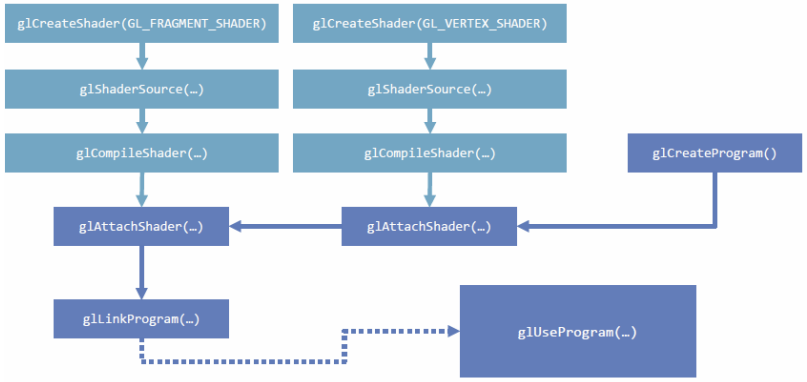
Primitive types include *GL_POINTS, GL_LINES, GL_LINE_STRIP, GL_TRIANGLES* and *GL_TRIANGLE_STRIP.* After the pipeline itself is configure, a draw call can be issued to actually draw some of those primitives.

VERTEX SPECIFICATION

The vertex specification wires together the pipeline input and configures the **vertex fetch**, in doing so it reads the vertex attribute data from buffers, performs the necessary data format conversion and feeds the vertex attributes into the vertex shader. The state configured in the vertex specification stage is held in a **Vertex Array Object (VAO)**, which holds all the buffer object bindings and vertex attribute mappings.



SHADERS



Shaders are the programs that can be used in the programmable parts of the pipeline, they are part of the pipeline as for example the vertex or fragment shader and are compiled from GLSL code (a C-like OpenGL shading language). Those shaders are then linked together into a **program object**, which holds all the linked shaders and represents the whole pipeline.

VERTEX SHADER

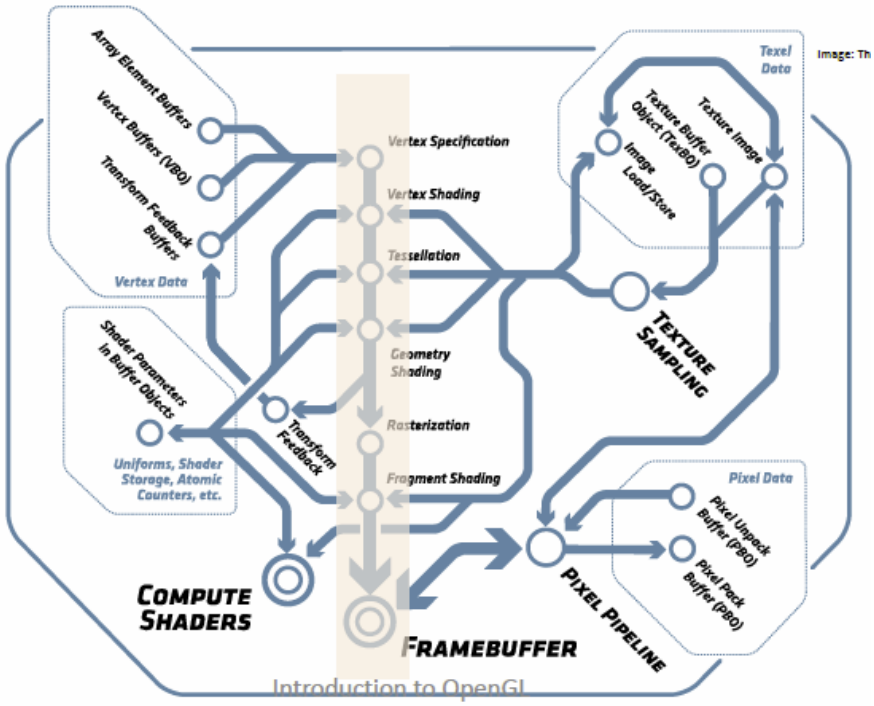
A **vertex shader** processes each vertex, it receives *vertex attributes* and also emits *vertex attributes*, which can be manifold but must include *gl_Position*.

RASTERIZER

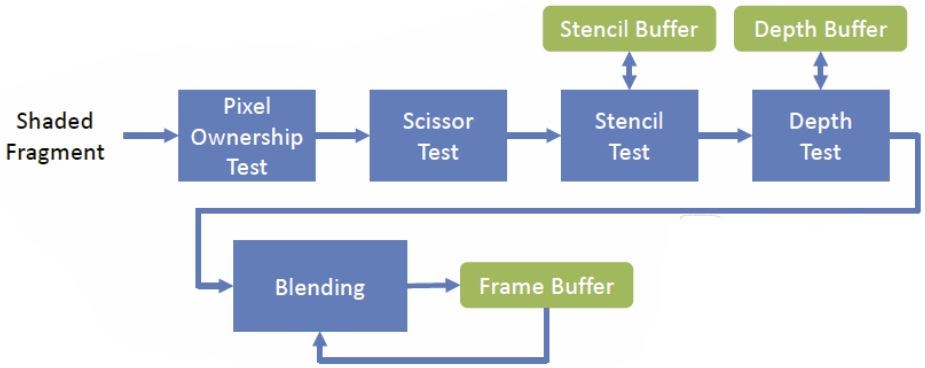
This part is part of the **fixed-function pipeline** and rasterizes the given primitives. It receives *primitives* like vertex attributes and emits *fragments*, which are essentially interpolated vertex attributes.

FRAGMENT SHADER

This is a programmable shader once again and processes each fragment. It receives *fragments* (interpolated vertex attributes) and emits a *fragment color.*



The **fixed-function parts** of the pipeline can be accessed through **built-in variables** and finally, **fragment merging** occurs which evaluates the final color of the pixels on screen.



TEXTURE MAPPING

Texturing can be used to enhance the visual appearance of plain surfaces and objects by applying fine structured details. It remains the basis for most real-time rendering effects and provides the look and feel of a real surface. **Texture Mapping** is described as

*A **function** that is evaluated for every **fragment** of a surface.*

The texture itself is usually given as a 2D image and my hold arbitrary data which can be interpreted by fragment programs. Textures can be used to simulate **color, reflection, gloss, transparency, bump** and much more.

TEXTURE TYPES

There are various types of textures, the main ones include *one-dimensional functions* (parameter can have an arbitrary domain)*, two-dimensional functions, three-dimensional functions* and *raster images* (texels, most often used).

1D Textures can be used anywhere where a linear value is used in a shader, **2D Textures** are directly related to surfaces and can map various stuff like color, height, glossiness and much more. **3D Textures** may represent procedural textures, where the texture information itself holds the spatial information of the object in 3D.

The process of **texture mapping** is generally the process of finding the corresponding *(u,v)* coordinates for each vertex. The problem is, that typical objects are quite complex and the parametrization is difficult and there are several approaches to achieve this.

PLANAR PARAMETRIZATION

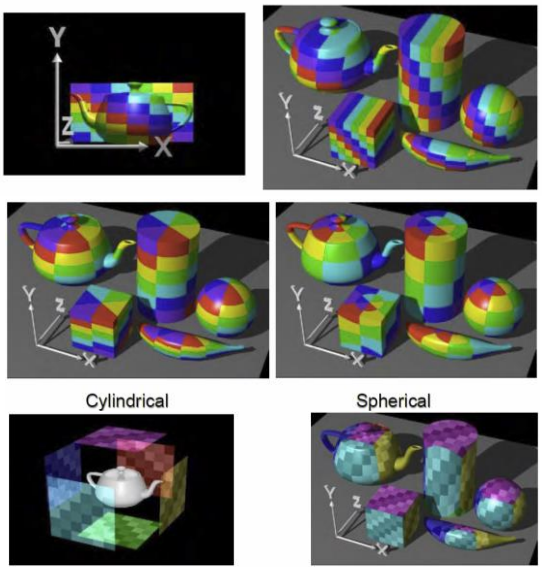
Here the object is placed in some coordinate space and the texture is mapped planar to its surface. This only looks good from the front.

CYLINDRICAL/SPHERICAL PARAMETRIZATION

The angle between vertex and object center is computed and mapped, this is easier to visualize in a polar/spherical coordinate system.

BOX PARAMETRIZATION

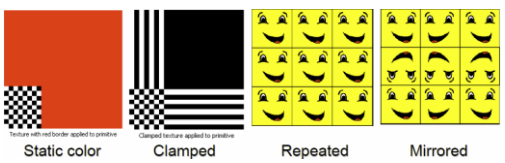
This is mainly used for environment mapping



But *no singular parametrization technique is optimal* for every situation and all mappings have distortions and singularities, those often have to be fixed manually in CAD software.

TEXTURE ADDRESSING

Textures are addressed with their respective *(u,v)* coordinates which reside in the [0,1] space. There are several options for dealing with pixels outside of that, those so-called **texture addressing modes** include *repeat, mirror, clamp or static color*.



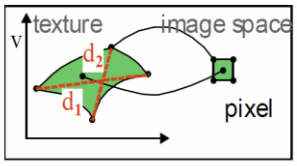
ALIASING

If pixels map to too many texels, **Aliasing** can occur as the screen resolution is too low for the detail available in the texture data. The correct pixel value is typically the weighted mean of the pixel area projected into the texture space and there are two approaches to deal with the problem here.

Direct Convolution calculates the weighted mean of the relevant texels at runtime in an approximation with a fixed number of samples.

Via **Prefiltering** it is possible to store the texture at different resolutions, this is also called **MIP Mapping**, which is heavily used in real-time graphics and is simple and also memory efficient, as the texture is always reduced by a factor of 2. The memory overhead is just 33%, when the channels are stored in an efficient way. Another possibility is given, if the texture resolution is too small, then simply the highest mip level is up-sampled, for example by linear interpolation.

$$D := ld(\max(|d_1|,|d_2|))$$



T_0 := value from table D_0 = trunc (D)
T_1 := value from table $D_1 = D_0+1$
(both by interpolation)

$$\text{pixel value} := (D_1-D)\cdot T_0 + (D-D_0)\cdot T_1 = (D_1-D)\cdot(T_0-T_1) + T_1$$

Is the texture look up done simply by calculating **nearest neighbor**, then the result is not good, when using **linear interpolation**, some aliasing can be eradicated, but higher frequencies still have problems. **Mip Mapping** deals with this problem, but introduces another problem, **blur**.

The problem with mip mapping is the **anisotropic distortion**, as mip mapping blurs equally in every direction, this approach is not optimal if the distortion due to projection is not uniform. This problem can be dealt with by using **anisotropic filtering**, where the texture sampling pattern is adapted to the shape of the projected pixel (approximation of ideal solution). This can be implemented through **rip mapping**, where the prefiltering is also done along anisotropic (non-uniform) scales.

3D TEXTURE MAPPING

This also often referred to as **solid texturing** and directly maps the object space to the texture space which enables the possibility to open up objects. With this approach it is also possible to generate **procedural solid textures**, either witch 3D functions or with projections of 2D data.

VOLUMENTRIC TEXTURES

Volumetric textures are an explicit, sampled function which is represented as a 3D voxel array of texture values. Those representations most commonly are used in medical scanners as it works in real time, but a lot of memory is needed.

MULTIPASS RENDERING

The standard OpenGL lighting model is **local** and therefore limited in complexity. But many effects become possible when using multiple rendering passes, effects like *environment maps, shadow maps, reflections, mirrors, transparency*… There are mainly two methods, which can be mixed as well:

- **Render to auxiliary buffers and use result as texture** (environment maps, shadow maps, requires FBO support)
- **Redraw scene using fragment operations** (reflections, mirros, light mapping, uses framebuffer blending)

RTT (RENDER TO TEXTURE) METHOD

This method is also called the auxiliary buffer method and works like this

1. Frame buffer with color attachment is bound
2. Scene (or subset) is rendered, possibly using a custom shader program
3. Frame buffer is unbound, color attachment is bound as texture
4. Scene is rendered to the back buffer where the rendered texture can be used by a custom shader or the fixed-function pipeline

BLENDING METHOD

This methods is older and compatible with more OpenGL implementations and requires two passes

1. **First Pass**: Establish z-buffer (and maybe stencil), usually with diffuse lighting
2. **Second Pass**: Z-testing only and render the special effect using blending

The resulting color is then the blended **sum** of the incoming fragment color and the framebuffer color, both weighted with special factors. It does not have to be the sum, it can also be the **min, max, subtract** or other operations. An example would be transparency blending, where the alpha and color weights can be defines separately.

$$C = C_s \cdot \alpha + C_d \cdot (1- \alpha)$$

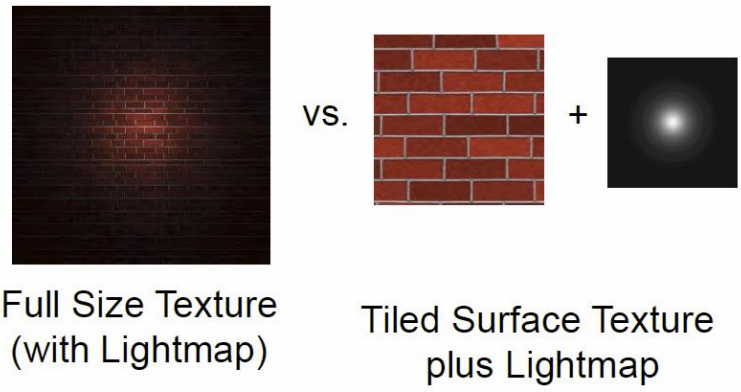
MULTITEXTURING

The idea is to **apply multiple textures in one pass**, which then can be combined arbitrarily and also enables indirect lookups, where the 1st texture lookup is used to determine the index into the 2nd texture for example. Today this is mostly used in pixel shaders and is standard functionality since OpenGL 1.2.1.

LIGHT MAPPING

Light mapping **precalculates the diffuse lighting** on static objects, as this is view independent and only needs to be available in low resolution. The **advantages** are that it is more accurate than vertex lighting, it can take global effects like shadows or radiosity into account and also runs on older cards that do not have runtime lighting.

The first step is the **map generation**, where a single map for a group of coplanar polygons is used and mapped back into worldspace to calculate the actual lighting. The second step then is the **map application**, where the textures are either **premultiplied with the light maps** (not dynamic, large textures) or it is done via **multitexturing** at runtime (fast and flexible).



The main problem with premultiplication is, that it requires more memory, as the full size texture needs to reside in memory, whereas with multitexturing a tiled texture plus a low resolution light map are sufficient to achieve the same effect with less memory requirements.

PTM (PROJECTIVE TEXTURE MAPPING)

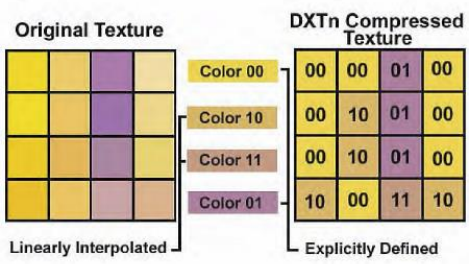
This resembles the effect of a video projector, textures are mapped using **perspective projection**. The object coordinates are mapped to the light frustum, which then can be expressed as a 4x4 projective transform matrix. A usage example would be a flashlight effect, but it is also the standard way of texturing reconstructed models with camera images in Computer Vision.

TEXTURE ANIMATION

It is also possible to animate textures by specifying an arbitrary 4x4 matrix for each frame, which enables for example the possibility of moving water or the sky. With custom shader programs it is also possible to use video as texture, which allows for arbitrary texture animations.

TEXTURE COMPRESSION

Texture compression can free up a lot of memory, one example would be the **S3TC texture compression (DXTn)** which represent a 4x4 texel block by two 16 bit colors (5/6/5bit), 2 bits are stored per texel. To uncompress, 2 additional colors are created between the two colors and 2 bits are used to index this color map.



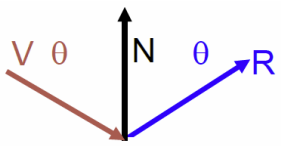
ENVIRONMENT MAPPING

With the help of environment mapping it is possible to use textures to **create reflections**, this technique uses texture coordinate generation, multitexturing, RTT and more. The environment map is index via the orientation of the viewpoint, the trick is, that the reflecting object is shrunk to a single point (or the environment is infinitely far away, this works as the eye is not were good at detecting the fake. Typical mappings were discussed before (spherical, cube, box, dual paraboloid…).

CUBE MAPPING (BOX PARAMETRIZATION)

Cube Mapping is implemented directly in the OpenGL pipeline and these maps can be generated in advance or programmatically, they remain the primary method of generating reflections in real-time 3D. The cube map is accessed via vectors expressed as 3D texture coordinates, the projection from 3D to 2D is then done by hardware. The correct ray/quad intersection although is too slow, instead the highest magnitude component selects which cube to use and other components are divided by this factor, the resulting 2D coordinates select a texel from this face. But this does not define the environment mapping itself, the need to generate useful texture coordinates remains.

To get this cubic environment mapping, it is necessary to generate view of the environment for each of the cube faces, which represent a 90° view frustum, the hardware can be used to render this map directly to a texture. The reflection vector can then be used to index the cube map which is generated automatically by the hardware.



$$R = V - 2 (N N^T) V$$

The cube map needs the reflection vector in world coordinates, therefore the texture matrix needs to be multiplied with the inverse 3x3 view matrix.

Cube mapping allows texture mapping with **minimal distortion**, the creation and mapping are hardware accelerated and it is also possible to generate the data dynamically. In dynamic scenes it is also possible to speed up the process by using lower resolutions and maybe not update every frame, which is sufficient most times.

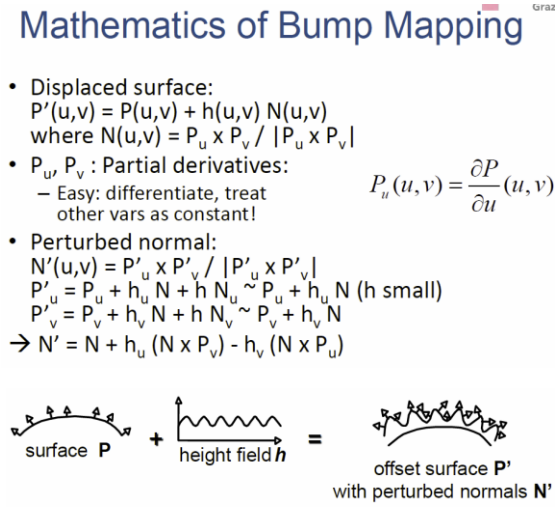
PER-PIXEL LIGHTING

By using **per-pixel lighting** it is possible to produce better specular highlights and simulate smooth surfaces by calculating the illumination at each pixel, e.g. with Phong shading. This is done by **custom shader programs**, as the vertex normal are passed to the *fragment shader*, which are interpolated on the way there by the *rasterizer*.

BUMP MAPPING

It is also possible to simulate rough surfaces by calculating the illumination at each pixel, the per-pixel normals are needed for Bump mapping, and it is also possible to replace the geometry information with a texture. The original idea comes from **ray tracing**, instead of interpolating the normals, a map is used for supplying the normals.

So **per-pixel lighting is a special case of bump mapping**, it is simply **bump mapping without the bump map**, the normal vector is interpolated instead of looked up. With bump mapping, the Surface P is modified additionally by a 2D height field (bump map) *h*.



BUMP MAP REPRESENTATIONS

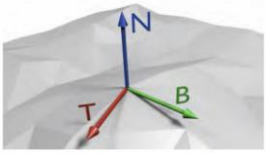
- **Height fields:** Must approximate derivatives during rendering
- **Offset maps:** Encode orthogonal offsets from unperturbed normals, requires renormalization, calculated using finite differencing on the height field
- **Normal (perturbation/rotation) maps:** Encode direction vectors, no normalization required and calculated via the standard method

COORDINATE SYSTEMS

The questions is where to calculate the lighting, in **object coordinates**, **world coordinates** (native space for light vector, env-map but not explicit in OpenGL), **eye coordinates** (native space of the view vector) or in **tangent space** (native space for bump/normal maps).

Most often, **tangent space** is used and this process is called **normal mapping**, those maps appear usually blueish, as the perturbed normals mostly face up (translates to color blue).

TSM = $\begin{pmatrix} Tx & Bx & Nx & 0 \\ Ty & By & Ny & 0 \\ Tz & Bz & Nz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$



The tangent space itself is a *concept from differential geometry* and is the **set of all tangents on a surface**, a orthonormal coordinate system for each point on the surface, this is a natural space for normal maps as a vertex normal is simply N=(0,0,1) in this space. Every vertex needs the three vectors T,B and N (stored in attribute arrays) which are interpolated for each fragment and can then be set up as a **tangent space matrix TSM** as shown above.

- **For each vertex**: L and V are transformed with TSM and normalized and the normalized H is computed from those.
- **For each fragment**:
  - L and H are interpolated, renormalized and N' = texture(s,t) is fetched
  - Compute $max(L \cdot N', 0)$ and $max(H \cdot N, 0)^s$ and combine using the standard phong equation

REFLECTIVE BUMP MAPPING

To achieve reflections, it is possible to use **EMBM** (Environment Map Bump Mapping), this is an example of dependent texturing.

- **For each Vertex:**
  - Transform V into world space
  - Compute tangent space to world space transform (T,B,N)
- **For each Fragment:**
  - Interpolate and renormalize V
  - Interpolate frame (T,B,N)
  - Look-up N = texture(s,t)
  - Transform N from tangent space to world space
  - Compute reflection vector R using N
  - Look-up C = cubemap(R)

BUMP MAPPING ISSUES

Bump mapping may create artifacts, for example when computing shadows, as the shadows are not changed by the bump map as needed, also there is no parallax (changing the viewpoint does not change the bump map). Also it is not effective if either the light or the object moves, in these cases light maps should be used, except for specular highlights.

PARALLAX MAPPING

This is also called **relief mapping** and is technically enhanced bump mapping, as the texture look up is simply displayed according to the viewing angle. This allows for more realism with motion parallax and steeper height maps with occlusion.

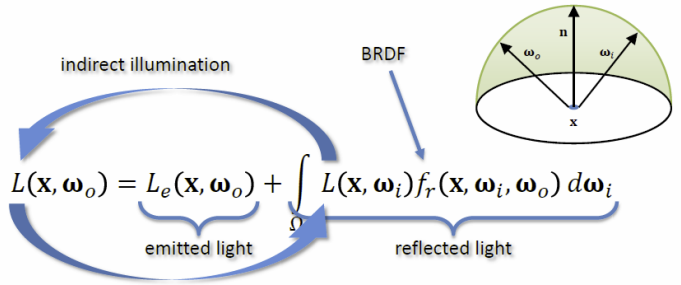
It uses raycasting in the fragment shader, which is computationally expensive, but the silhouettes and shadows have the correct shape. It can replace almost all geometry with good relief maps and results therefore in a raycasting engine.

FILTERING BUMP MAPS

Ideally the reflection is computed at different positions with different normals and the result is averaged, but **in practice** the perturbed (normalized) normals are averages and are used to compute the reflection. This is then further corrected for diffuse lighting, if no self-shadowing is involved.

SHADING MODELS

When it comes to **shading**, the general goal is to find out, what color the individual pixel should have and thereby also how light is reflected by objects in the scene.



BRDF

The **BRDF** (Bidirectional Reflectance Distribution Function) describes how a surface reflects light from an incoming direction ω_i at a location x into the outgoing direction ω_o.

It has the following properties

- **Reciprocity** : $f_r(x, \omega_1, \omega_2) = f_r(x, \omega_2, \omega_1)\ \forall \omega_1, \omega_2$
- **Energy conservation** $\int f_r(x, \omega_i, \omega_o) d\omega_i \leq 1\ \forall \omega_o$
- **Positivity** : $f_r(x, \omega_1, \omega_2) \geq 0$

LIGHT SOURCES

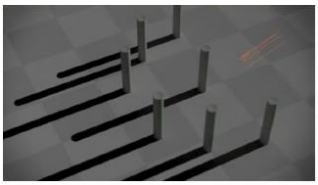
The most general form of describing a light source would be the **area light**, but this requires heavy integration and allows an analytical solution only in trivial cases, as it is necessary to gather light from a range of directions at every location.

To make this problem feasible, only analytically feasible light sources are considered like the **point light** or the **directional light**, if the light only arrives from a single direction, the integral vanishes.

POINT LIGHT

The light source is **infinitesimal small**, so at every surface location p, the incoming light comes from a **single location**, only the light position is needed to calculate the shading according to a point light.

DIRECTIONAL LIGHT



The light source is assumed to be **infinitely far away**, therefor at every surface location p, the incoming light comes from the **same direction**, therefore only the light direction is needed to calculate the shading.

SPOT LIGHT

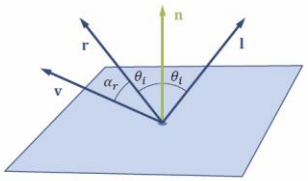
Works similar to a point light, but the light volume is restricted to a cone, the shading according to this light source can be done with the given **light position**, **cone direction, inner cone angle** and **outer cone angle**.

PHONG SHADING

The **phong shading model** is not a physically meaningful BRDF, but it is phenomenological model. **Highlights** are always **circular** and **energy conservation** is **ignored**, Normalization would be needed for larger terms *m*.

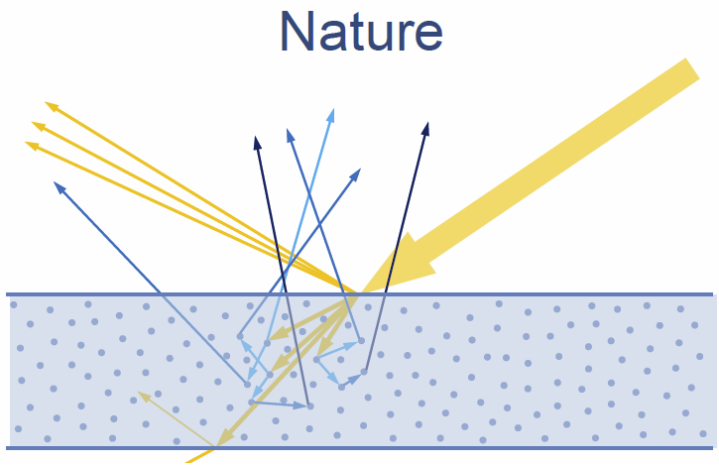
- $\mathbf{L}_o = \mathbf{c}_e + (\mathbf{c}_d \circ \cos\theta_i + \mathbf{c}_s \circ (\cos\alpha_r)^m) \circ \mathbf{B}_L$
  - $\mathbf{c}_e$ Emissive color
  - $\mathbf{c}_d$ Diffuse color
  - $\mathbf{c}_s$ Specular color
  - m Specular power
  - $\mathbf{B}_L$ Light color

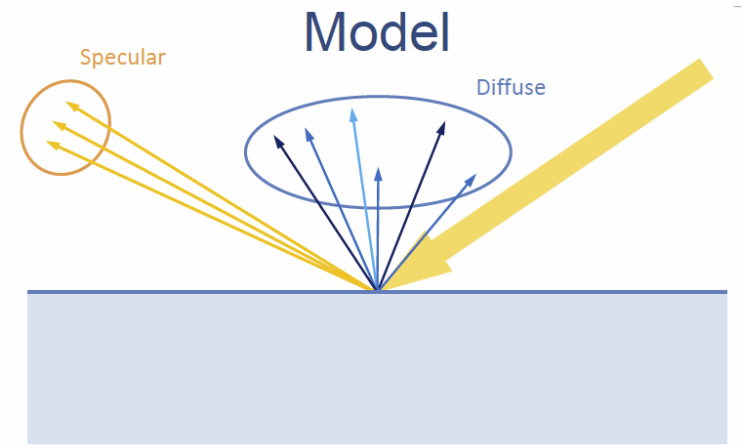
All vectors assumed to be normalized!



$\mathbf{r} = 2\mathbf{n}(\mathbf{n} \cdot \mathbf{l}) - \mathbf{l}$

All light-matter interaction can be described by describing **emission, scattering** and **absorption**.

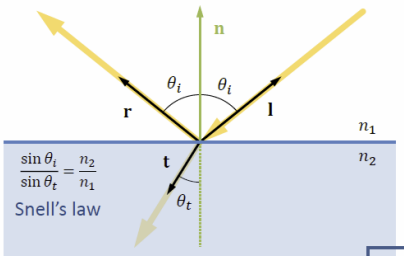




FRESNEL MODEL

As the phase speed of light is different in different media, at a boundary between two media part of the light wave is **directly specularly reflected** and part of the light wave is **transmitted**, given that the material is not transparent. This transmitted part becomes the diffuse reflection. The ratio between those two parts is described in the Fresnel equations.

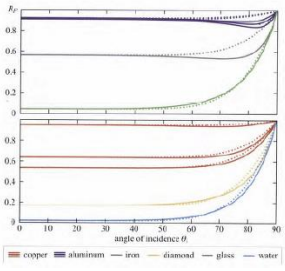
$R = R_0 + (1 - R_0)(1 - \cos\theta_i)^5$



$$R_s = \left| \frac{n_1 \cos\theta_i - n_2 \sqrt{1 - \left(\frac{n_1}{n_2}\sin\theta_i\right)^2}}{n_1 \cos\theta_i + n_2 \sqrt{1 - \left(\frac{n_1}{n_2}\sin\theta_i\right)^2}} \right|^2$$

$$R_p = \left| \frac{n_1 \sqrt{1 - \left(\frac{n_1}{n_2}\sin\theta_i\right)^2} - n_2 \cos\theta_i}{n_1 \sqrt{1 - \left(\frac{n_1}{n_2}\sin\theta_i\right)^2} + n_2 \cos\theta_i} \right|^2$$

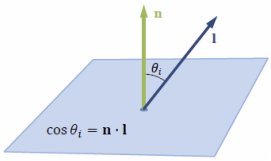
Specular $R = \frac{R_s + R_p}{2}$ $T = 1 - R$ Diffuse



Easier (faster) to compute than the original Fresnel equations are **Schlick's approximation of Reflection**.

$$L_o = c_d \circ \max(\mathbf{n} \cdot \mathbf{l}, 0) \circ I_L$$

- c_d Diffuse reflectance („albedo")
- I_L Irradiance from light

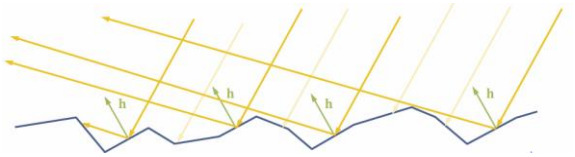


LAMBERT SHADING MODEL

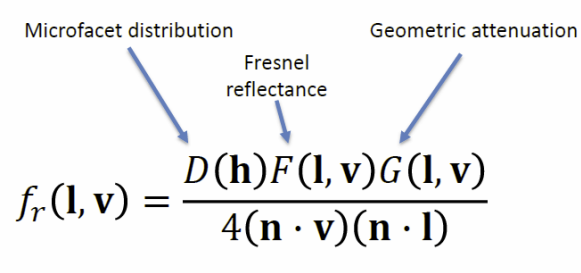
Lambertian Shading assumes a **perfectly diffuse reflector**, which scatters light evenly in all directions, which makes it view independent and only dependent on the orientation of the surface towards the light.

MICROFACET MODEL

The surface is assumed to be made up of lots of tiny mirrors and the goal is to model the distribution of mirrors. Only mirrors oriented halfway between light and view direction reflect light into the camera.



The direction *h* describes the halfway-vector between light and view direction.



$$f_r(\mathbf{l},\mathbf{v}) = \frac{D(\mathbf{h})F(\mathbf{l},\mathbf{v})G(\mathbf{l},\mathbf{v})}{4(\mathbf{n}\cdot\mathbf{v})(\mathbf{n}\cdot\mathbf{l})}$$

The **microfacet distribution** tries to measure the fraction of microfacets oriented in direction *h*, this can be done by using for example the Beckmann Distribution.

$$D(\mathbf{h}) = \frac{1}{4m^2(\mathbf{n}\cdot\mathbf{h})^4}\exp\left(\frac{(\mathbf{n}\cdot\mathbf{h})^2 - 1}{m^2(\mathbf{n}\cdot\mathbf{h})^2}\right)$$

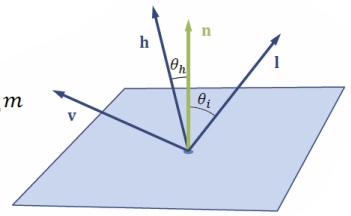
m Root mean squared slope of microfacets (corresponds to roughness)

BLINN-PHONG MODEL

The **Blinn-Phong model** tries to approximate the energy conservation.

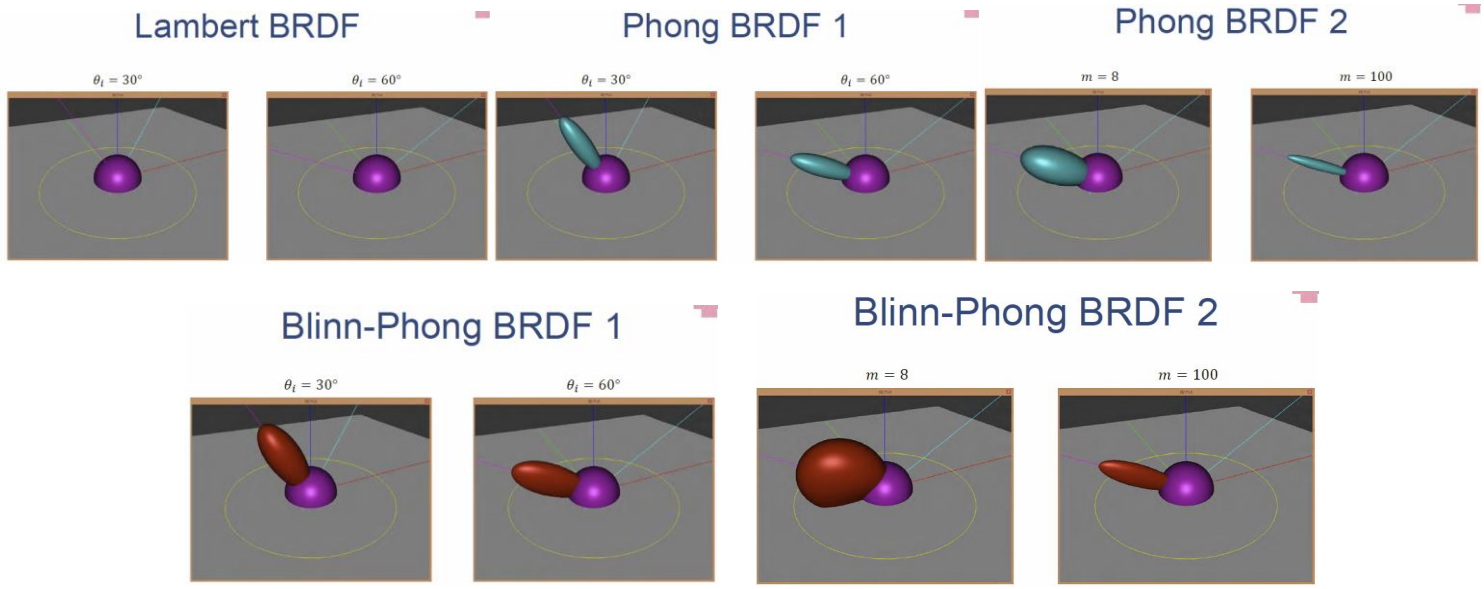
- $f_r = \frac{\mathbf{c}_d}{\pi} + \frac{m+8}{8\pi}\mathbf{c}_s(\cos\theta_h)^m$
  - $\mathbf{c}_d$ Diffuse color
  - $\mathbf{c}_s$ Specular color
  - m Specular power

All vectors assumed to be normalized!



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

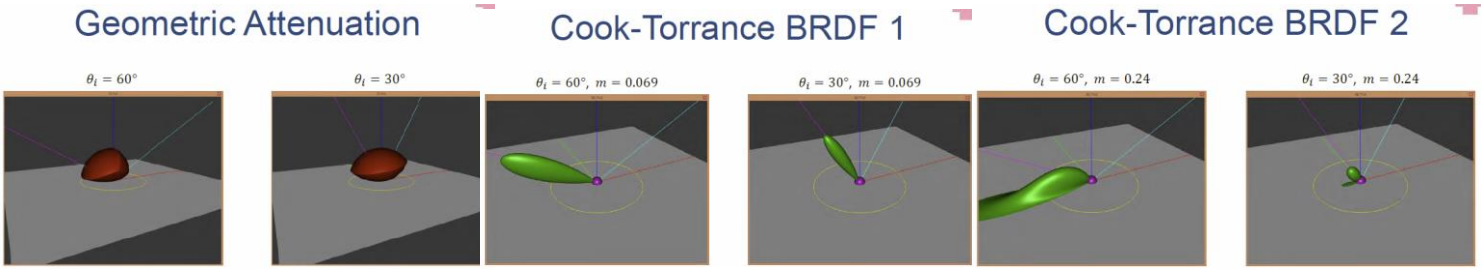
COMPARISON BETWEEN DIFFERENT SHADING TECHNIQUES



GEOMETRIC ATTENUATION

This accounts for

- **Shadowing** (Light blocked from reaching microfacet by other microfacets)
- **Masking** (Reflected light blocked from reaching camera by other microfaces)



ANISOTROPIC REFLECTION

Also considers the viewing direction and not just the inclination, is used for example as a brushed metal effect.

RETRO-REFLECTION

Deep cavities on a surface reflect the light back to the light source which leads to brighter areas at steeper angles (for example near the silhouette of the object).

BRDF:

outgoing radiance

$$f(\mathbf{l}, \mathbf{v}) = \frac{dL_o(\mathbf{v})}{dE_i(\mathbf{l})}$$

irradiance from light source:

light source intensity

irradiance incident on surface

incoming irradiance

$$E_L = \frac{I_L}{r^2}$$

squared distance to light source

radiance towards viewer:

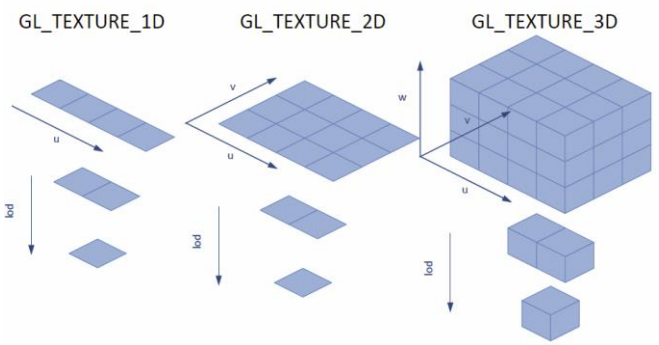
$$L_o(\mathbf{v}) = \sum_{k=1}^{n} f(\mathbf{l}_k, \mathbf{v}) \circ E_{L_k} \cos\theta_{i_k}$$

sum up contributions of each light source

TEXTURES AND FRAMEBUFFERS

OpenGL stores **texture images** in so-called **texel arrays**, which have a certain dimensionality and format. **Texture objects** hold one or more texture images (e.g. mip levels, array slices…) and **sampler objects** configure the actual texture sampling.

A texture can be loaded by allocating storage first with *glTexStorage*()* and then setting the texture data by calling *glTexSubImage*()*. The internal texture format ranges from integer to floating point for the color and depth values.

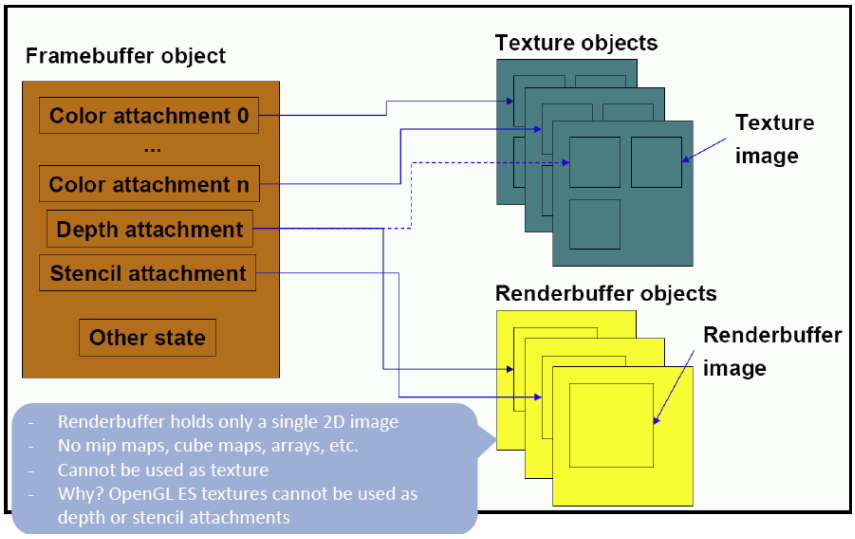


SAMPLER OBJECTS

Sampler objects are used to configure the texture sampling itself by specifying the filtering, addressing mode and much more. As there is only a limited number of texture units available, one is selected by using *glActiveTexture()*, a texture is bound to this unit as well as a sampler object and then the texture data can be sampled into the shader. As discussed previously in texture addressing, there are many different mode to address the texture and to handle border cases.

FRAMEBUFFER OBJECTS (FBO)

The framebuffer is the output stage of the render pipeline, where the default framebuffer is associated with the OpenGL context. A **Framebuffer Object (FBO)** replaces the default render target and contains independent *logical buffers* like color, depth and stencil buffer, those buffers can be texture objects or simply render buffers.



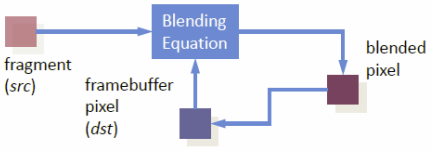
ALPHA – CHANNEL

Alpha measures the opacity of something to simulate translucent objects, composite images or enables a way of Antialiasing. If blending is not enabled, this effect is ignored. To be even able to use the alpha channel, the requirement is to have the objects depth sorted, as the z-buffer alone is not really helpful for partially transparent pixels.

BLENDING

Blending combines new pixels with what is already in the framebuffer (does not always require the alpha channel), there are different settings for blending that can be used to achieve a certain effect.

$$\mathbf{c}_d \leftarrow \mathbf{c}_s \otimes \mathbf{w}_s + \mathbf{c}_d \otimes \mathbf{w}_d$$



GL_ZERO: multiply color with (0,0,0)
GL_ONE: multiply color with (1,1,1)
GL_DST_COLOR: multiply with dst-color
GL_SRC_COLOR: multiply with src-color
GL_SRC_ALPHA: multiply with src-color's alpha value
GL_DST_ALPHA: multiply with dst-color's alpha value
GL_ONE_MINUS_SRC_ALPHA: mult. with 1 - src-alpha
GL_ONE_MINUS_DST_ALPHA: mult. with 1 - dst-alpha

GL_FUNC_ADD

$$\mathbf{c}_d \leftarrow \mathbf{c}_s \otimes \mathbf{w}_s + \mathbf{c}_d \otimes \mathbf{w}_d$$

GL_FUNC_SUBTRACT

$$\mathbf{c}_d \leftarrow \mathbf{c}_s \otimes \mathbf{w}_s - \mathbf{c}_d \otimes \mathbf{w}_d$$

GL_FUNC_REVERSE_SUBTRACT

$$\mathbf{c}_d \leftarrow \mathbf{c}_d \otimes \mathbf{w}_d - \mathbf{c}_s \otimes \mathbf{w}_s$$

GL_FUNC_MIN

$$\mathbf{c}_d \leftarrow \min(\mathbf{c}_s, \mathbf{c}_d)$$

GL_FUNC_MAX

$$\mathbf{c}_d \leftarrow \max(\mathbf{c}_s, \mathbf{c}_d)$$

Blending Examples (1)

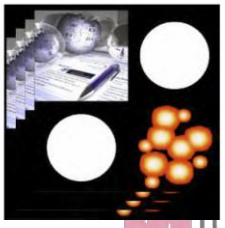
Texture with alpha channel for the following examples...

Blending Examples (2)

- `glBlendFunc(GL_ZERO, GL_ONE);`
  - Nothing is drawn
  - Not really useful...

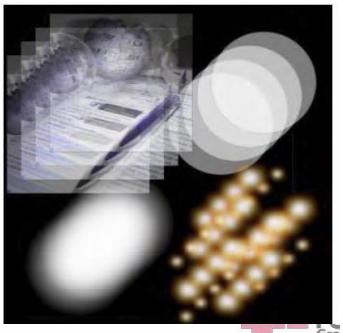
Blending Examples (3)

- glBlendFunc(GL_ONE, GL_ZERO);
  - Everything works as if blending was disabled
  - Alpha values also ignored



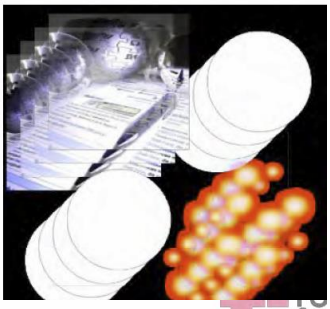
Blending Examples (4)

- `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);`
  - Most common blending method
  - Simulates traditional transparency
  - Source color's alpha (usually from texture) defines opaqueness of object to render



Blending Examples (5)

- `glBlendFunc(GL_SRC_COLOR, GL_ONE_MINUS_SRC_COLOR);`
  - Source color's color value defines opaqueness of object to render
  - Rarely used – saves alpha channel

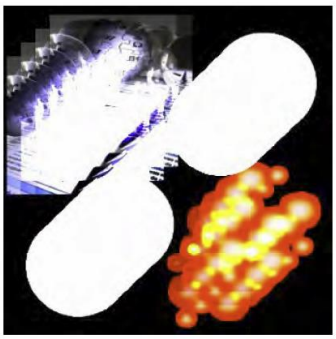


Blending Examples (6)

- `glBlendFunc(GL_ONE, GL_ONE);`
  - Source and destination color are simply added
  - Values >1.0 are clamped to 1.0 (saturation)
  - Good for fire effects

Blending Examples (7)

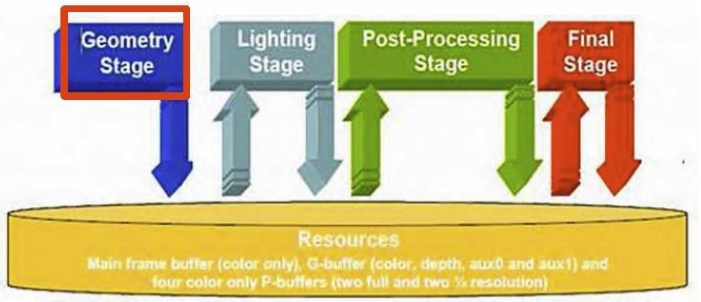
- `glBlendFunc(GL_SRC_COLOR, GL_DST_COLOR);`
  - Both colors are squared before adding
  - Similar to GL_ONE, GL_ONE but with very high contrast



DEFERRED SHADING

Conventional forward shading happens after rasterization either in the fixed-function pipeline or a fragment shader and observes the complexity = O(#Lightsources * #Objects). This leads to the possibility of **overdrawing**, whereas shading is calculated in complex and large virtual environments for pixels that will be overdrawn in the final image.

The idea now is to perform the shading at the end of the rendering process, therefore only one shading operation is needed per pixel.

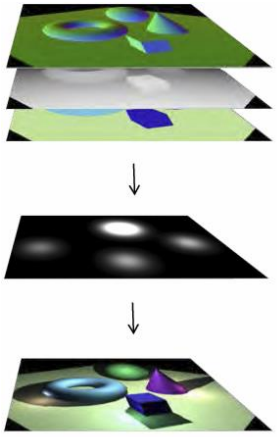


To do so, the rendering pipeline is split into two separate stages, first the **geometry transformation** and **rasterization** is performed, after that the actual **shading** occurs in the **lighting stage**. In an optional final stage, post processing can be applied to the image.

To pass data from one stage to the next, **G-Buffers** are used, these buffers store multiple values per pixel, including **color, depth, normal vector, position, object identity, etc**…

DEFERRED RENDERING

Deferred Rendering **(DR)** is a framework implemented as a post-processing effect, which supports various effects like *SSAO, non photo-realistic rendering, HDR rendering, deferred shading* and many more, but the most important sub-type is **deferred shading**.

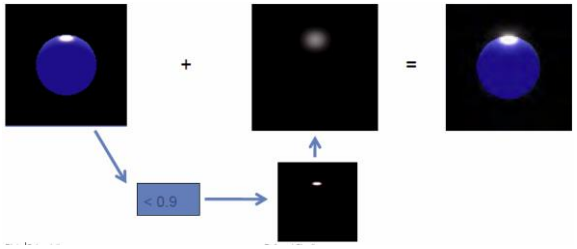


The **Geometry Stage** is performed as usual, geometry transformation are applied, everything is rasterized and material and texturing is applied, but **no shading**. Instead of shading, the intermediate results (diffuse color, depth, position, normal vectors…) are stored in **G-Buffers.**

In the **actual shading stage** a screen-sized quad is rendered, the fragment shader reads the G-Buffers and performs the shading and post-processing and stores this result finally in the framebuffer.

Via **Postprocessing**, many effects can be added like Anti-Aliasing or Bloom (based on blurring, bright areas spill light). **Blurring** can be achieved by using a Gaussian Filter and many other effects also are based on the Gaussian filter. The nice thing about it is, that it is **separable**, as even 5x5 filters would require 25 texture lookups. By separating the 5x5 filter, it is possible to achieve the same effect by doing 2 passes, one filters with a 5x1 filter in *u*-direction and one filters with a 1x5 filter in *v*-direction. The Lookups can be performed via linear filtering, so a 5x1 filter only requires 3 lookups.

BLOOM



Before the Gaussian filtering, the rendered texture intensities are modified, the glowing part is clamped and filtered and finally added back to the image again. It is typically applied to a down sampled render which effectively increases kernel size, but sharp highlights are lost. By combining differently down sampled and filtered textures many effects are possible.

A **star effect** can be achieved by filtering *u* and *v* separately and adding the textures also separately. **Bloom** can be used to disguise aliasing artifacts on materials and works best for shiny materials and the sun or sky. Nowadays it is a little bit overused and can smudge out a scene too much, as contrast and sharp features are lost.

DEPTH OF FIELD (DOF)

DoF **simulates a camera property**, as lenses can only focus on a certain depth level at a time, objects at the depth level appear sharp while the rest appears blurred depending on the distance to the focal plane. It can be used to **guide the user's attention** towards a certain spot in the image.

For each pixel in the fragment shader, the **circle of confusion (CoC)** is computed based on the depth buffer and then the image is blurred using a convolution or random sampling, the needed window size depends on the CoC. The problem is, that sharp foreground objects leak onto the blurry background, to deal with this issue, the depth values need to be examined, and closer pixels can be discarded.

But **this effect does not occur with small apertures**, which is mostly the case in computer graphics that uses pinhole cameras with infinitely small aperture. This effects **is simulated**, either by adapting the camera model (which is not possible in the standard OpenGL pipeline) or it is approximated by blurring the image based on depth buffer values.

Problems associated with the Postprocessing DoF effect are color bleeding and discontinuities at the silhouettes, those can be counteracted by using a bilateral filter or advance techniques based on diffusion for example.

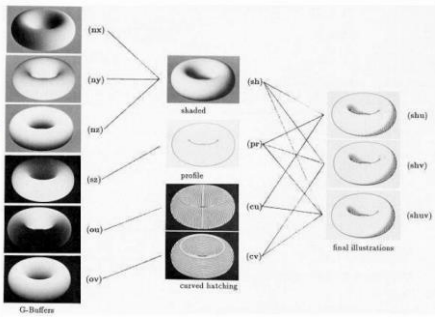
NON-PHOTOREALISTIC RENDERING

There are various types of NPR, but it often emphasizes the object edges and silhouettes. Edges can be found by search for depth discontinuities by using the depth buffer. This can be done using 1st order differential operators like **Sobel** to get the profile and 2nd order differential operators like **Laplace** to get the internal.

Problems arise when small discontinuities are present in the z-domain, a better solution is to do edge detection on the **G-Buffer**, which makes use of all the available data and uses both the depth and the normal buffer for edge detection.

The G-Buffer can also be used to produce the illusion of illustration, to get a **hatching** effect, textures can be created from brush strokes which follow the surface coordinates of the object.

nx: normal vector z
ny: normal vector y
nz: normal vector z
sz: screen coordinate
ou: coordinate *u* (on curved surface)
ov: coordinate v (on curved surface)



DEFERRED RENDERING ADVANTAGES

Geometry only needs to be rendered once, complex shading and post-processing is done per pixel which reduces the complexity significantly and is independent of the geometry and depth complexity, which makes it a good fit for games.

DEFERRED RENDERING DISADVANTAGES

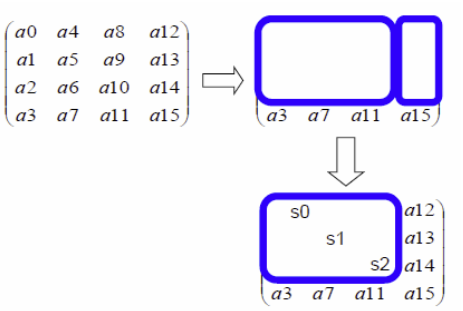
But it requires more memory and also frequent read/write operations and cannot use hardware anti-aliasing. Also, if advanced effects are needed (like transparency, ghosting…), a per-pixel sorting is required. **Forward Shading** may be faster if there is only a low number of lights, only few post-processing effects and a low depth complexity.

IMAGE SPACE SPECIAL EFFECTS

The goal is to add special effects after the rendering is complete, in **image space**, so that it is independent on the geometric scene complexity and can also use often used image processing techniques.

BILLBOARDS

Billboards are also called impostors or sprites and are prerequisites for many effects, they are represented through a textured triangle which is aligned in a special way (may face the viewer or aligned with some axis). Due to the simple nature and small memory footprint, it can be used in large quantities and allows for **fast hardware rendering**, as it is never examined in detail e.g. clouds in the sky.



Billboards facing the camera need to be transformed to always face in the right direction, to achieve this, the **ModelView** matrix is changed, the **rotation** is **removed** but the **scale** is **maintained**. By doing so, the Billboard will appear at the right position and distance and also face the camera.

Billboard clouds are a set of billboards with different sizes or orientation, which also be created procedurally or animated by a physical simulation.

PARTICLE SYSTEMS

Particle Systems try to **model objects changing over time**, typically those are many **small objects** and the main distinction is between **state-less and state-full particles**. It is used to model many natural phenomena like rain, snow, clouds, explosions…

All particles of a system use the same update method and also share the same properties, a particle system therefore handles the initialization step, the updating of all particles, the randomness to simulate the real world and the rendering of those particles. Particles can be described through parameters (location, speed, lifetime), which may change over time. The particles themselves can appear in many different shapes, also those shapes can vary over time.

As most particle systems mimic physical phenomena, the motion may be controlled by external forces like gravity or collision and particles can interfere with other particles and thereby cause a more entropic movement.

Most times, correct modelling is not as important as the associated memory consumption and rendering speed.

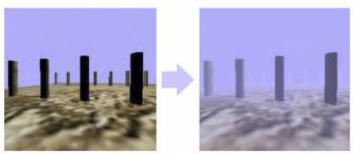
FOG

Fog is an **atmospheric effect** (scattering of light) and is mostly used as a stylistic element or to give clues about the depth, but also to hide artifacts in the distance.

The intensity of the fog scales with the distance to the camera, this leads to the term **distance fog**, where the surface color is blended with the fog color. The **fog factor** f can be linear, exponential or even squared exponential to give different stylings to the fog.

$$\mathbf{c} = f\,\mathbf{c}_s + (1-f)\,\mathbf{c}_f$$

$\mathbf{c}_s$ surface color
$\mathbf{c}_f$ fog color
f fog factor



Linear fog: $f = \frac{d_{end}-d}{d_{end}-d_{start}}$

Exponential fog: $f = e^{-d_f \cdot d}$

Squared exponential fog: $f = e^{-(d_f \cdot d)^2}$

BOKEH

The word **bokeh** means something like *blur* and is an effect of a camera lens system, which is essentially the CoC shaped by the aperture of the camera, this gives a filmic look to a scene. It can be simulated by **convoluting with an aperture-shaped filter kernel**, but artifacts may pose a problem with color bleeding and discontinuities.

A real-time approach is based on splatting, where a **textured quad is drawn for each pixel** using the geometry shader, each is textured with a bokeh shape, and whereas the color is given as the product of the scene color times the bokeh and then blended together via additive blending. In a **normalization pass**, the bokeh weights are accumulated in the alpha channel and divided by alpha to normalize.

To optimize this approach, the final image can be split into layer to avoid artifacts and regions further away can be rendered in lower resolution for the blurring.

MOTION BLUR

Motion blur is a property of the human eye (moving the eye causes blur) and cameras (due to long exposures), adding this to real-time graphics adds realism and can also cover up some performance problems.

Motion blur is a **continuous effect**, but it can also be approximated **discretely**. The simplest method is to render the object at earlier positions with varying transparency, but for this the object needs to be rendered multiple times. It makes more sense and it's more efficient to render the object to a buffer and simply copy this buffer with varying transparency.

To implement **continuous motion blur**, for each pixel it is necessary to compute how the pixel moves over time, form a **velocity buffer** by looking at the current and previous ModelViewProjection matrix and sample a line along that direction to accumulate the color values.

As with all blur operations, the artifacts include **color bleeding** (slow foreground objects bleed into fast background) and **discontinuities at silhouettes**.

LENS FLARE

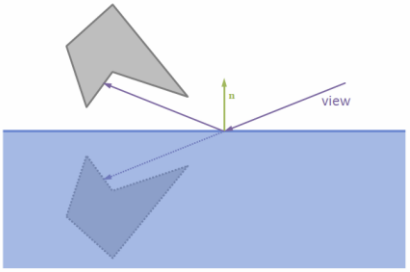
Lens flares occur when imperfect lenses are used in cameras and are essentially a shortcoming that photographers try to avoid, but it looks realistic and fancy and appear as a star, ring or hexagonal shapes.

To implement **lens flare rendering**, a lens flare texture is chose and rendered with differently sized textured quads and alpha blending on the line between the light source and the image center.

PLANAR REFLECTIONS

Reflections on planes are done by drawing the desired object, drawing the mirror transparent and drawing the object itself, but as this is not restricted to the mirror itself, it is necessary to clip it to this plane.

This can be done by using a stencil buffer.



STENCIL BUFFER

The stencil buffer allows to mask parts of the framebuffer, it is available in various bit depths and can be used for reflections, constructive solid geometry and shadow volumes, although it is nowadays mostly replaced by programmable shading.

To render reflections using the stencil buffer, the buffer is reset, the mirror is rendered into the stencil buffer, the mirrored object is then only rendered in the mirror and the object itself is rendered normally afterwards.

To improve the quality of the reflection, the object color is faded out with increasing distance to the reflector. Two things to watch out for are that the reflection changes the handedness, so the winding order of the faces needs to be adjusted and also if the scene intersects the mirror plane it should be clipped against the plane to avoid artifacts.

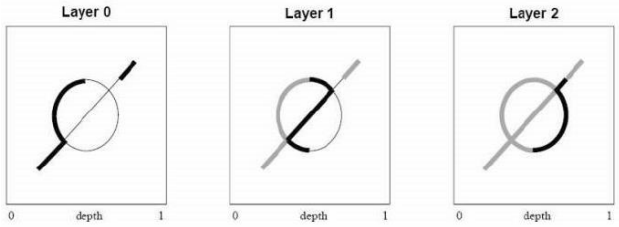
This can also be done in a shader by rendering parts of the scene below the plane into an additional texture, for water use fog to simulate the scattering.

TRANSPARENCY

Transparency is not only a matter of blending, but the order is important so the objects need to be sorted back to front. Sometimes sorting the triangles is enough, but not always, as cyclic overlaps may occur or triangles may intersect, then it is necessary to sort the fragments.

DEPTH PEELING

Via **Depth peeling** it is possible to achieve order-independent transparency, here the image is rendered multiple times, the first past finds the front-most depth/color and each successive pass finds the depth/color of the next nearest fragment on a per-pixel basis, so each pass draws what is “behind” the previous pass. Each of the passes forms a layer and a shader can be used to compare each layer with its previous layer and finally compose all layers together.



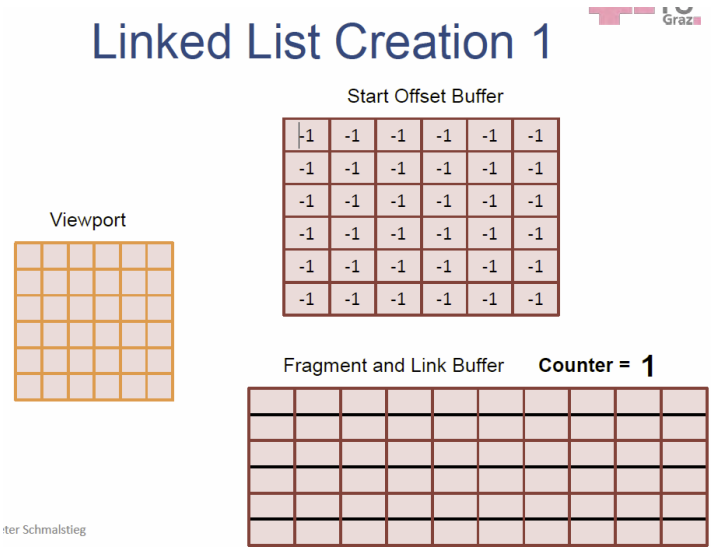
Dual Depth peeling peels front and back in one pass and keeps track of min/max depth in two textures. It compares the current depth with the min/max values and can then decide whether to render to the front layer or the back layer, this decreases the number of passes from n to n/2 + 1.

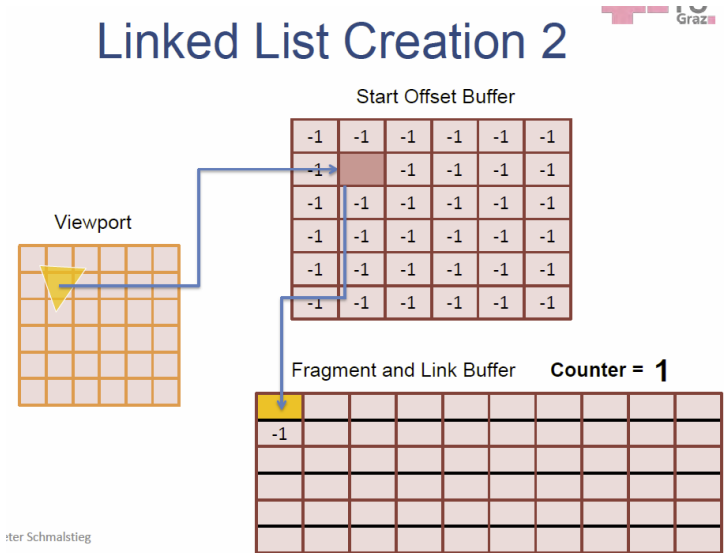
The problem remains, that both sorting triangles and depth peeling are slow so hardware support is preferable.

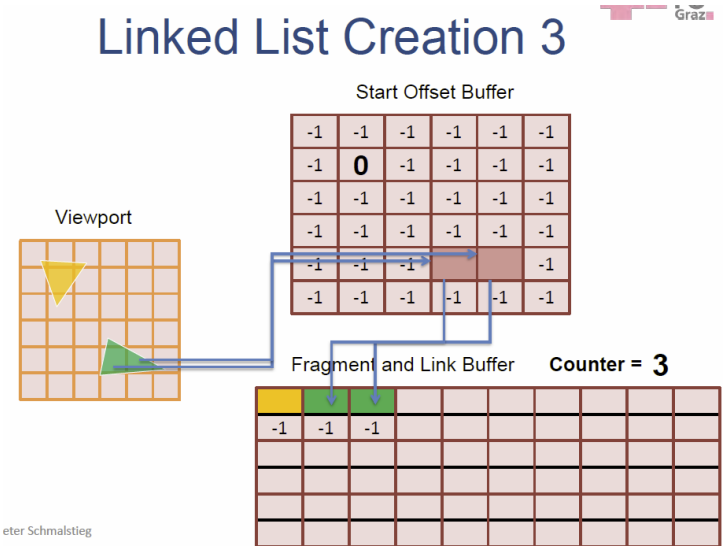
PER-PIXEL LINKED LISTS

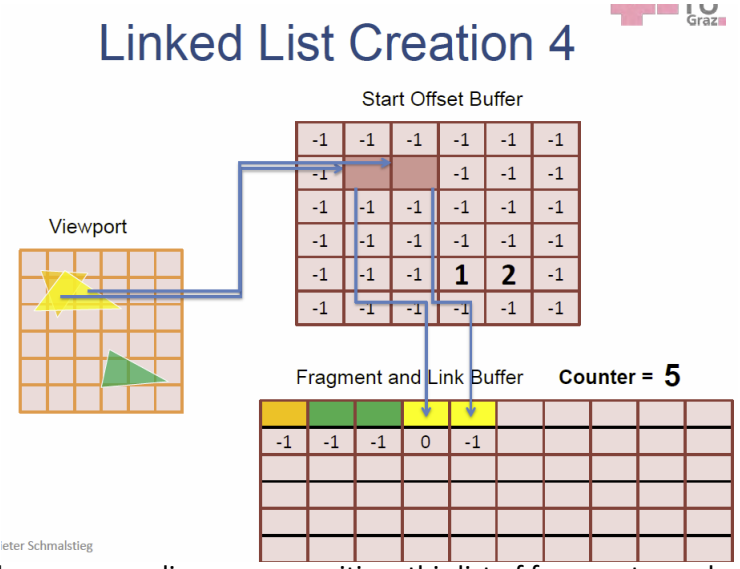
The scene is rendered and a linked list is generated per pixel, these lists can be sorted and used to compose the fragments. This requires a read/write buffer and two additional buffers (fragment and link buffer plus start offset buffer).

The **fragment and link buffer** contains all fragment data produced during rasterization and must be large enough to store all fragments. The **start offset buffer** contains the offset of the last fragment written at every pixel location, it is initialized to some magic value.

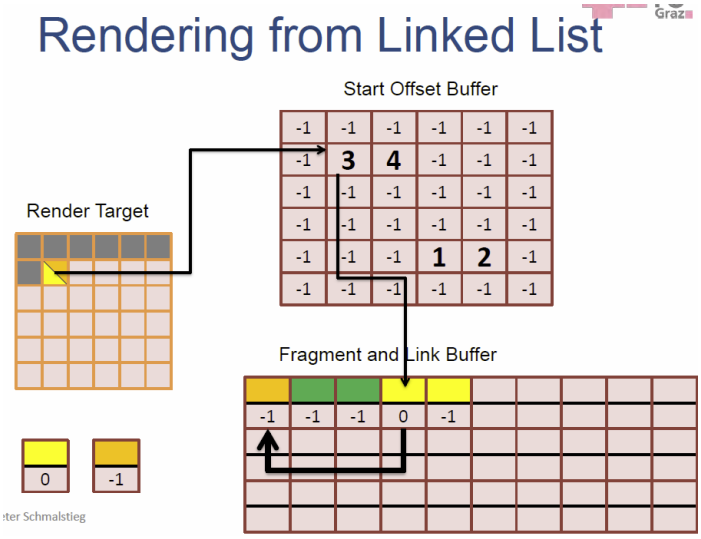


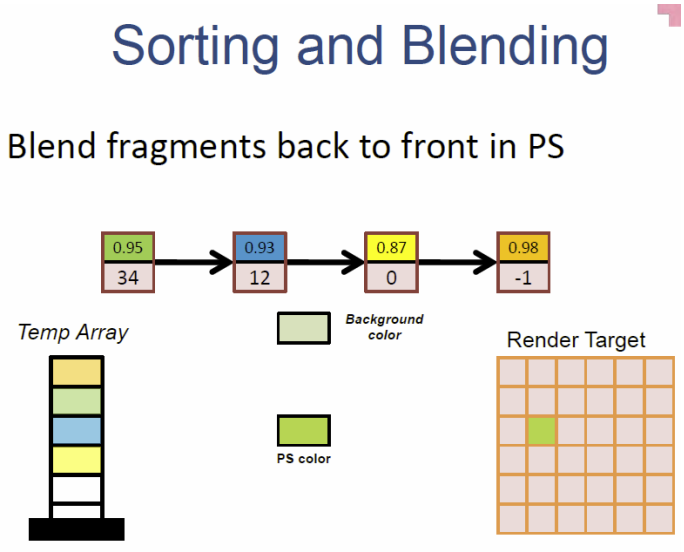






This linked list can be traversed to retrieve the fragments for the corresponding screen position, this list of fragments can be processed (sort and blend) to produce the resulting image.





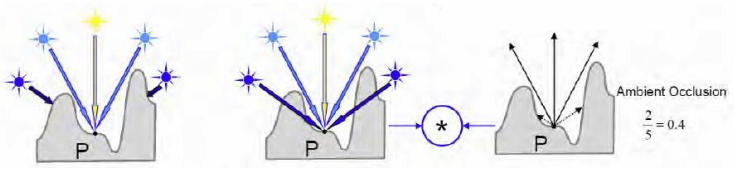
SHADOWING WITH ENVIRONMENTAL LIGHTING

Up to this point, shading is done with a known light and its position, now the shadowing is calculated dependent on the ambient light, which is independent of the light/object orientation and adds depth and contrast to images.

AMBIENT OCCLUSION

The goal is to weigh the ordinary shading using the mean visibility, which is given as 1 – AO. **Mean-Visibility** uses raycasting to find self-occlusions and for each vertex, n rays are cast into the half sphere oriented by the normal and the blocked rays are counted.

$$I = (1 - AO)(k_a I_a + k_d I_d \max(\mathbf{n} \cdot \mathbf{l}, 0))$$



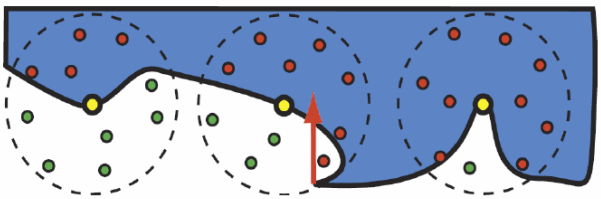
To account for Lambert's law, a cosine distribution is applied around the normal vector and the rays are also limited in length to account for other nearby objects. But this is computationally expensive.

- V_{AO} = Ratio occluded/unoccluded samples
- L_{in} = incoming radiance
- ω_i = incoming direction
- θ_i = angle to normal

$$I_{AO}(\mathbf{P}) = V_{AO} \sum_{i=1}^{N} \frac{\rho}{\pi} L_{in}(\omega_i) \cos\theta_i \Delta\omega$$

SSAO

The **screen space ambient occlusion (SSAO)** uses the z-buffer to approximate the surrounding area and comput the ambient occlusion from the neighboring pixels, whereas a random sampling from the z-buffer is used.



SSDO

With **screen space directional occlusion (SSDO)**, the directional illumination is also considered when computing the occlusion, this can also be used with **bounce**, where color bleeding of nearby materials is also taken into account.

In dynamic environments, surfaces are **approximated by oriented disks**, which allows the computation of the occlusion to be done analytically between the disks.

- Consider directional illumination

$$I_{AO}(\mathbf{P}) = V_{AO} \sum_{i=1}^{N} \frac{\rho}{\pi} L_{in}(\omega_i) \cos\theta_i \Delta\omega$$

$$I_{dir}(\mathbf{P}) = \sum_{i=1}^{N} \frac{\rho}{\pi} L_{in}(\omega_i) V(\omega_i) \cos\theta_i \Delta\omega$$

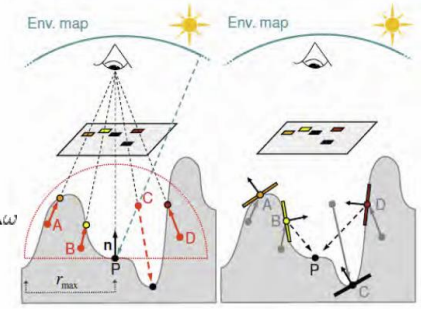
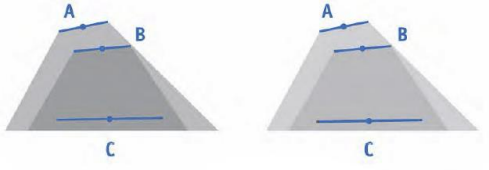


Figure 2: *Left: For direct lighting with directional occlusion, each sample is tested as an occluder. In the example, point P is only illuminated from direction C. Right: For indirect light, a small patch is placed on the surface for each occluder and the direct light stored in the framebuffer is used as sender radiance.*

- 1st pass as described
- 2nd pass: use estimated visibility from previous pass to weight occlusions
  - Elements which are in shadow will cast less shadows
  - After second pass – double shadowing is removed



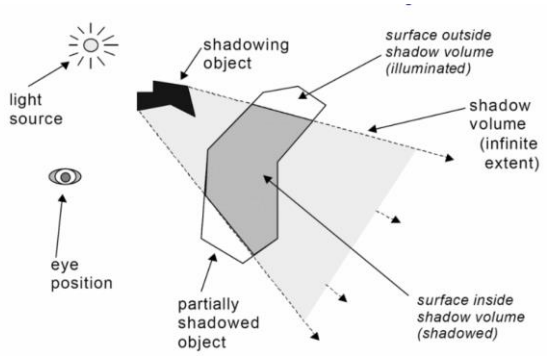
The problem arises with multiple occlusions, this can be dealt with by **using multi-pass rendering**.

“The display is the computer.”

- Jen-Hsun Huang [NVIDIA]

SHADOWS

Shadows originate from the fact, that light is blocked by objects and this gives an important visual clue considering the relative location and the position of the light. Shadows are a **non-local** effects between the **light source**, the **receiver** and the **occluder**.



Shadows are a byproduct of global light transport and therefore a *global illumination* problem which makes it hard for online rendering, as global information is typically not available during rendering. To deal with this problem, a **two-pass technique** can be used by calculating the shadow information in a first pass and then render the scene using this information.

REAL-TIME SHADOWS

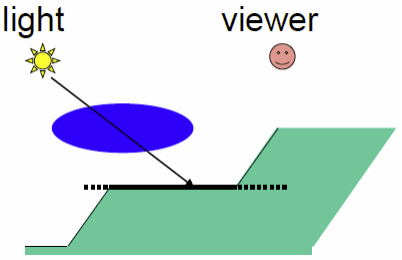
There are various ways to implement real-time shadows, which is being discussed in the next few headers.

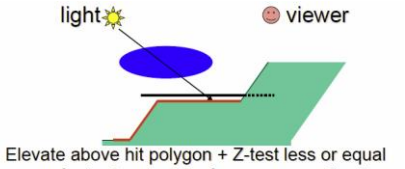
STATIC SHADOWS

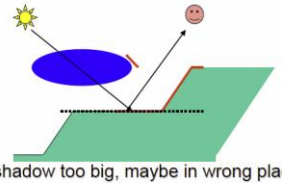
When using static shadows, it is possible to **incorporate shadows into the light maps** themselves, for each texel, a ray is cast to each light source and this information can be stored in the light maps.

APPROXIMATE SHADOWS

This kind of shadows tries to deliver shadows with **minimal cost**, as even some studies show, that the actual shape of the shadows is not nearly as important as one might think. So to do this, a dark polygon (maybe with texture) is used and a ray is cast from the light source through the object center.





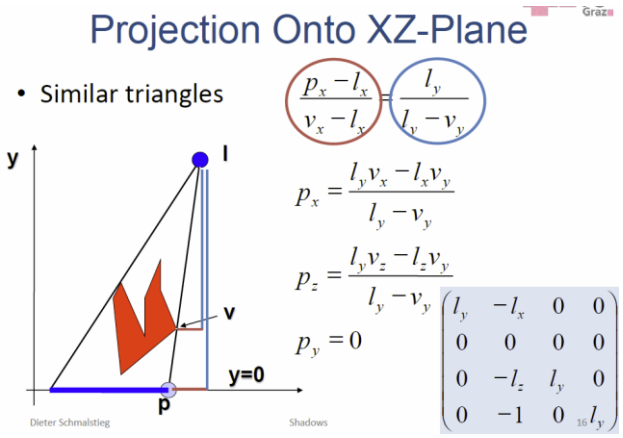


The polygon is then blended into the framebuffer at the location of the hit, rotation/scale/translation can also be applied to incorporate distance and receiver orientation.

Problems can arise with z-quantization, for example if the shadow and the hit polygon have equal z-test values, errors may occur.

PLANAR PROJECTED SHADOWS

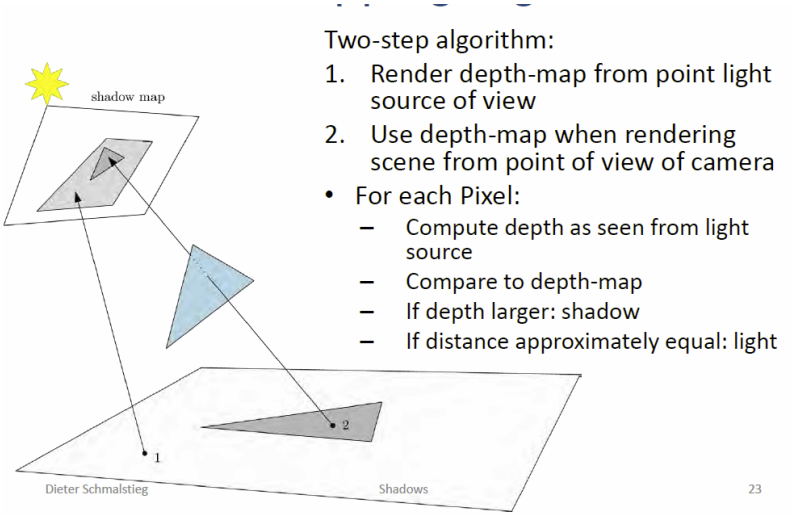
Planar projected shadows are also a very simply, yet effective technique to draw shadows. The **ModelView** matrix is used to transform the object into the plan, where it is drawn *flat* and *darkened* using the framebuffer blend. Possible artifacts are produces by z-fighting, double blending and extending off of ground region.



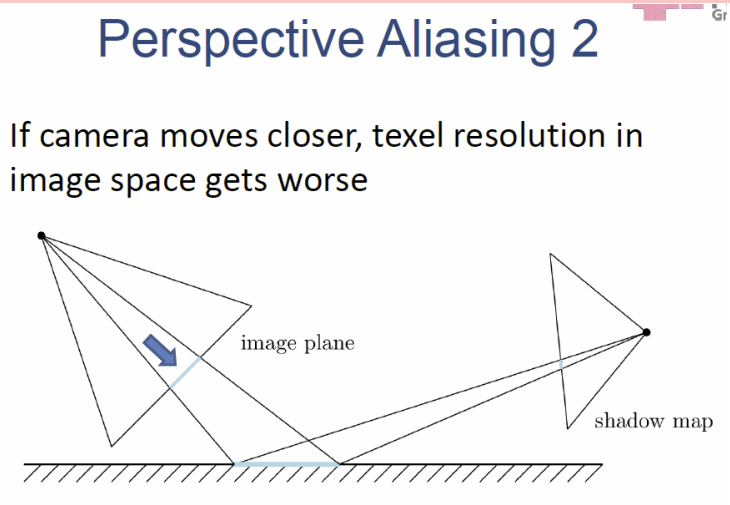
Projected Shadows can also be implemented using the **stencil buffer**, here the stencil buffer is cleared, the receiver plane is drawn to the stencil buffer, the actual 3D object is drawn and then the shadow is drawn to the stencil buffer. This eliminates the z-fighting problem, as all occurrences of 1 in the stencil buffer are replaced by 0.

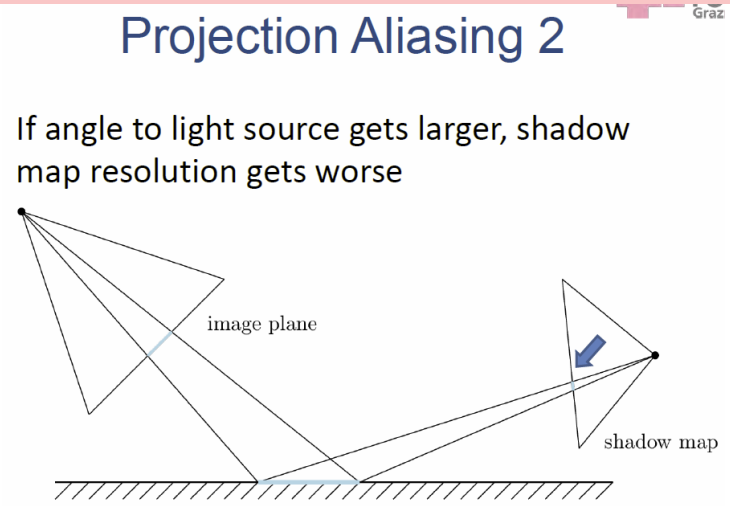
The implementation of projected shadows is very easy (first done in GLQuake) but it is only **practical for few, large receivers** and **no self-shadowing** is possible. Also other artifacts may occur, as objects behind the light source or behind the receiver may produce wrong shadows.

SHADOW MAPPING ALOGRITHM



ALIASING





Two different kinds of aliasing present a problem here, **perspective** and **projection** aliasing. To deal with the problem, the shadow map resolution could be increased, but this approach is not practical in general, better way to deal with the problem are **Anti-Aliasing** via filtering or to **optimize** the **shadow map sample distribution**.

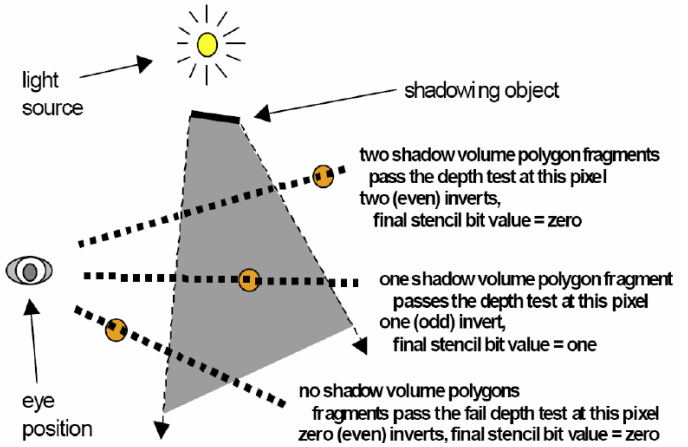
A technique to deal with this problem is called **percentage closer filtering**, where the look up result is filtered instead of the depth map values, and even better result is achieved by using poisson-disk distributed samples.

SHADOW MAP

Simple shadow maps are limited to a spotlight, it cannot handle a 360° Viewing frustum, so a better solution would be to use six shadow maps for **omni-directional** light source, but this is **expensive**. Shadow maps in hardware require a *high precision*

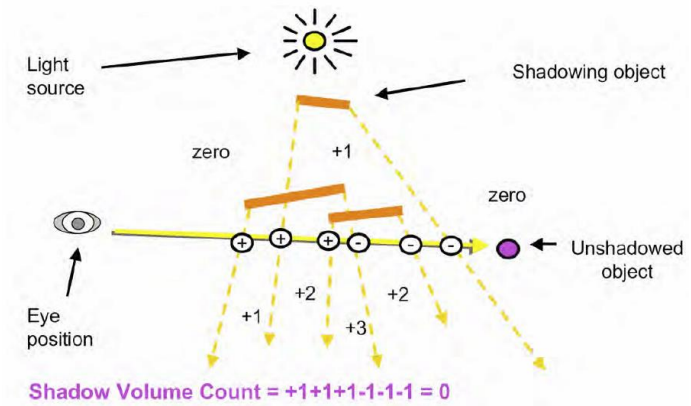
texture format and render the light space depth into the texture. The vertex shader can then calculate the texture coordinates as in projective texturing and the fragment shader can compare the depth.

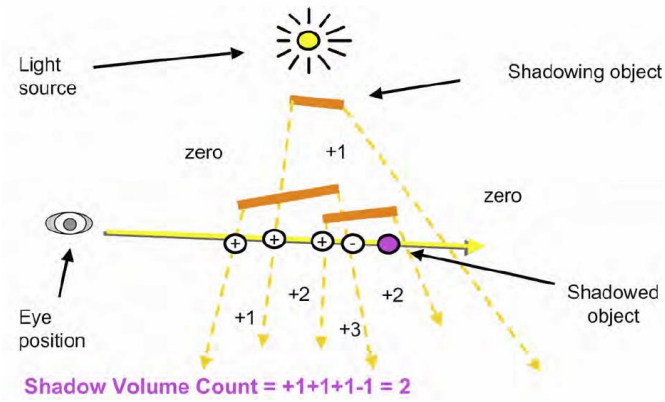
Shadow maps are therefore **fast** (only one additional pass), **independent of scene complexity**, **self-shadowing** is possible and sometimes it is even possible to reuse the depth map. But there are problems with **omnidirectional lights**, **aliasing** and also maybe with light leaks.



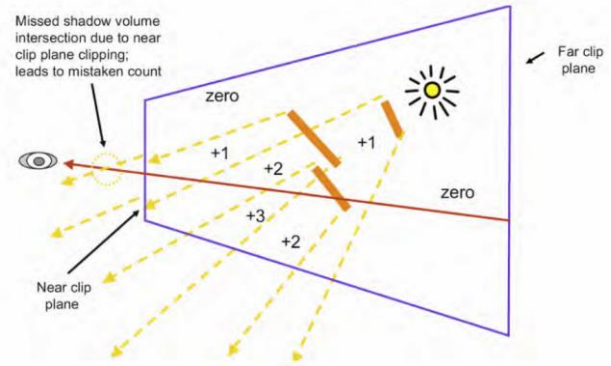
SHADOW VOLUMES

Shadow volumes are a **complex** technique to produce shadows, but can be **efficiently implemented** by **using a stencil buffer** and are **aliasing free**. View rays are intersected with the shadow volume and the number of intersections is counted until the receiver is hit.





Problems may arise, if the camera is inside the shadow volume or the shadow volume intersects the near-plane, this leads to a wrong shadow volume count.



The shadow volume itself is a closed polyhedron with 3 sets of polygons, the **light cap** (polygons facing the light), the **dark cap** (polygons facing away from the light, projected to infinity) and the **sides** (actual extruded object edges).

SILHOUETTE DETECTION

To detect the silhouette of an object, the polygons are classified into front faces and back faces, edges shared by a front and back face are part of the silhouette, those can be extruded away from the light source in the vertex shader and if necessary a front- and back-cap can be added.

STENCIL SHADOW VOLUME | DEPTH PASS

Here, the depth buffer is filled with the complete scene and depth writes are disabled after that. Then, the front-faces of the stencil volumes are rendered with stencil increment on the depth test pass (this counts only shadows in front of objects) and then the back-faces of stencil volumes are rendered with stencil decrement on the depth pass.

This leads to an incorrect result, if the camera is inside the shadow volume.

Stencil Buffering is present on all GPUs and really fast, with 2-sided stencil tests it is even possible to simultaneously look at front and back faces, this saves one pass. With **Hardware capping** it is possible to regain the depth precision with normal projection, but this requires watertight models with connectivity and watertight rasterization.

The **advantages** of **shadow volumes** include that it works for an arbitrary number of receivers and is fully dynamic, it can also deal with omnidirectional lights (unlike shadow maps), it delivers exact shadow boundaries and has inherent self-shadowing and broad hardware support via stencil buffering.

The **disadvantages** include that silhouette computation is required, it does not work for arbitrary casters like smoke, the fill-rate is intensive and it can be difficult to get right (Zfail vs Zpass).

GEOMETRIC SOFT SHADOWS

This assumes physical light sources with a real physical size which leads to soft shadows, this can be simulated by using many light points, but this is slow and does not look very good. The penumbra also needs to be explicitly calculated per pixel.

Shadow Algorithms Compared

Shadow Volumes

- Needs geometry
  - Closed manifolds
- Cost hard to predict
  - Geometric complexity of shadow volume
- Shadow rendering is bandwidth intensive ☹
  - Stencil buffer techniques
  - Additional pass
- No sampling artifacts ☺

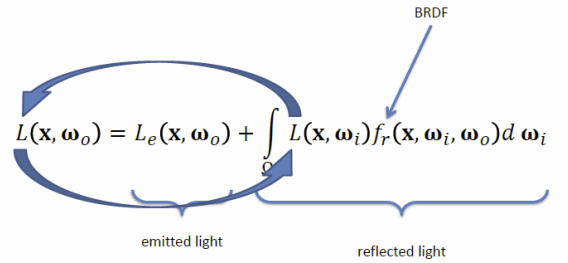
Shadow Mapping

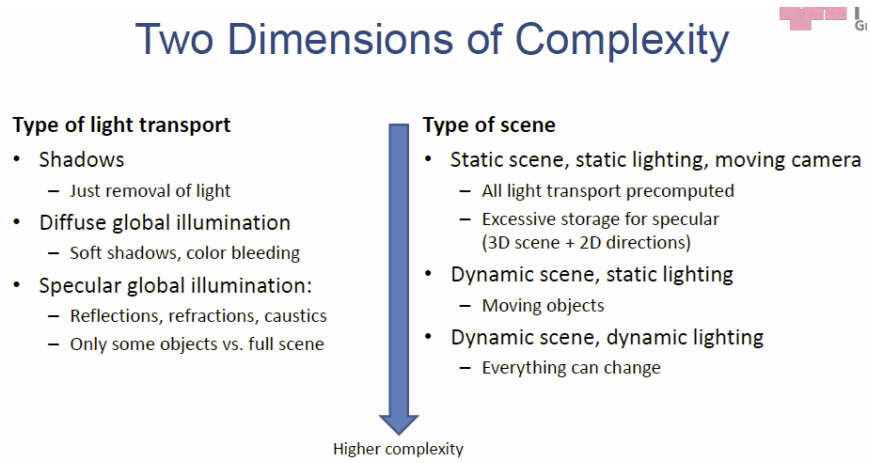
- Only depth needed
  - Everything one can render can cast a shadow
- Predictable cost
  - Rendering the scene once
- Shadow rendering is simple texture lookup ☺
  - Hardware Support
- Sampling artifacts ☹

GLOBAL ILLUMINATION

Generally it is possible to differentiate between **offline rendering** (complete scene information is available at all times) and **online rendering** (images are created while streaming scene data, surface fragments are only available during rasterization).

With **global illumination**, **objects influence each other's appearance** through *shadows*, *reflections*, *refractions*, *indirect illumination, ambient occlusion* etc.





RADIOMETRY

Already discussed in the chapter about shading models. With the **Neumann expansion**, not only direct but also indirect light transport is looked at, therefore the outgoing radiance needs to be written as an infinite series.

PATH TRACING

Path tracing is most often used as the **reference solution**, as it solves the rendering equation numerically by following all possible ray paths (specular and diffuse) and has no fundamental limits concerning realism. But it is **not really efficient**, as it is hard to find a path that connects camera and light source with a strong specular transport.

TWO-PASS GLOBAL ILLUMINATION

Two-pass global illumination computes the light transport in the scene in the 1st pass and in the 2nd pass collects the lighting information to generate the final image. Every pass chooses independently the type of light transfer, the rendering method and the update rate.

Examples of Two-Pass Methods

Method	Transport	1st pass result	2nd pass
Radiosity	Diffuse	Light maps	Render using texture maps
Photon mapping	Diffuse + some specular	Photon map	Raytracing
…			

RADIOSITY

Radiosity is a **finite elements approximation** and discretizes the scene into patches, with the assumption that everything is perfectly diffuse. Then the light transport can be calculates by solving a system of linear equations.

PHOTON MAPPING

Photon mapping works with 2 passes, in the 1st pass the **photon map is created** by shooting photons from the light source into the scene and store particles hitting diffuse surfaces in a data structure. In the 2nd pass the scene is **rendered** using the **photon map.** At each point x, the photon contained in a sphere of radius r are collected and represent the **intensity** at this point, with this it is possible to simulate complex light phenomena, but it needs a fast spatial search (kd-tree).

RENDER CACHE

The **render cache** is a data structure that connects the passes, it consists of the **radiance cache** (outgoing illumination per point) and the **irradiance cache** (incoming illumination per point). It can be index either by accessing only the points on surfaces or everywhere in space (voxel grid). It can be organized in many ways (by position, orientation, in projective space or by both position and orientation.

REAL-TIME GLOBAL ILLUMINATION

The problem with global illumination presents itself in the extremely high complexity and also the fact, that global light transport is not a natural fit for online rendering. There is need for **approximation**.

Approximation is possible by examining global effects only on some objects, by only allowing 1-2 bounces of light, by ignoring specular light transport or even use static scenes and light sources. And another way to make this method more efficient is by using **rasterization instead of raytracing**.

Real-Time Two Pass Methods

Method	Transport	1st pass result	2nd pass
Radiosity	Diffuse	Light maps	Render using texture maps
Photon mapping	Diffuse + some specular	Photon map	Raytracing
Instant Radiosity	Diffuse	Shadow map (rasterization)	Deferred rendering (rasterization)

INSTANT RADIOSITY

The idea here is to treat model bounces as light sources and introduce **Virtual Point Lights (VPL)**, so indirect light transport becomes direct light transport.

To create those **VPL**'s, photons are shot into the scene from the point of the light source and at each hit point a VPL is created.

INCREMENTAL INSTANT RADIOSITY

This is an optimization technique that assumes a semi-static scene and exploits the frame-to-frame coherency by reusing VPLs from previous frames, this means only a couple new shadow maps are needed per new frame.

IMPERFECT SHADOW MAPS

VPL approaches do not require accurate geometry, imperfection in the shadow map does not heavily affect the visual quality, because of that, smaller and fewer points can be used to approximate the scene as a point cloud.

Using **interleaved sampling**, only a few random light sources are examined per pixel and the result is averaged by using an edge-aware (bilateral) filter.

The **problems** with instant radiosity are, that it's not really scalable and is still not enough performance wise for games.

REFLECTIVE SHADOW MAPS

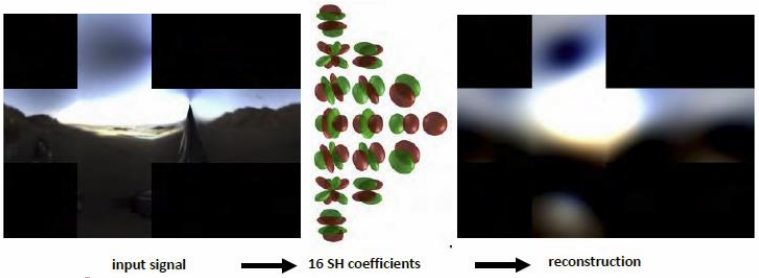
Each pixel is considered a small light source and single bounce illumination from surfaces seen by the light source is used, it is restricted to one bounce. Then the shadow map is rendered and here, additional information is stored. If too many pixel lights are presents, the number of samples is restricted to samples close to the current location. But it is a crude approximation, only allows for diffuse reflectors and does not provide indirect shadows.

PRECOMPUTED RADIANCE TRANSFER

The goals is to achieve realistic, interactive illumination of complex scenes without using ray tracing. The **PRT** method uses the fact, that the lighting environment and light transport are independent.

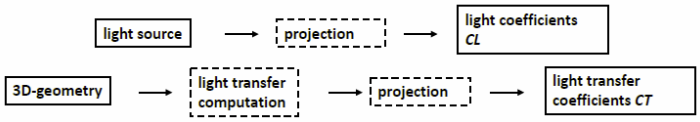
1. **Preprocessing:** precompute offline light transport (e.g. ray tracing) and produce a compressed representation of the lighting environment and of the transfer functions for each surface point
2. **Evaluate scene illumination:** This is done during rendering in real-time.

This compressed representation could be **Spherical Harmonics (SH)**, with it, it is also possible to represent a HDR environment by 4 bands (16 coefficients), and the coefficients can be computed by using projection.

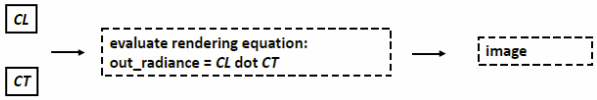


PRT Concept

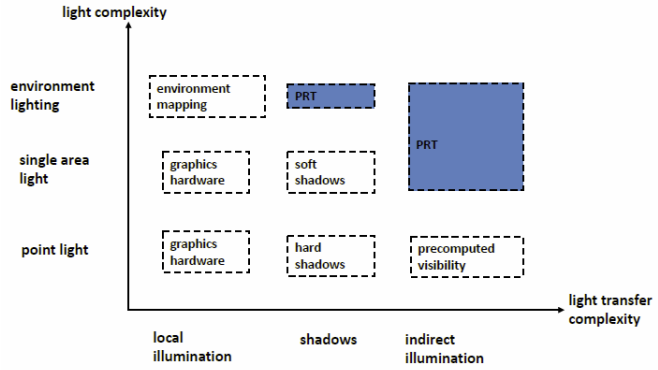
- Precomputation



- Rendering



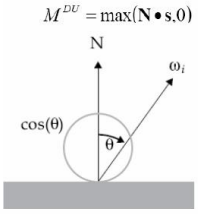
PRT – Applications

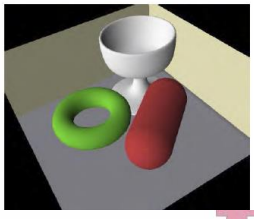


DIFFUSE LIGHT TRANSFER

Diffuse Light Transfer

$$L(\mathbf{x},\omega_o)=\int_S f_r(\mathbf{x},\omega_o,\omega_i)L_i(\mathbf{x},\omega_i)H(\mathbf{x},\omega_i)d\omega_i$$

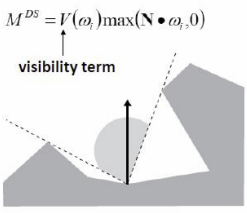


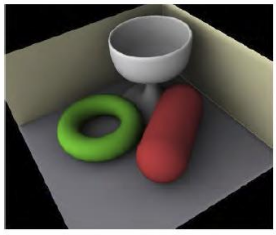


Diffuse Light Transfer with Ambient Occlusion and Interreflections

Diffuse Light Transfer with Ambient Occlusion

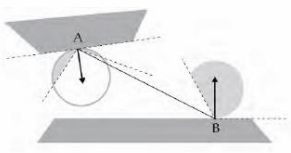
$$L(\mathbf{x})=\frac{\rho_x}{\pi}\int_\Omega L_i(\mathbf{x},\omega_i)V(\omega_i)\max(\mathbf{N}_x\bullet\omega_i,0)d\omega_i$$

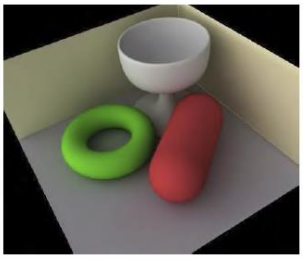




$$L_{DI}(\mathbf{x})=L_{DS}(\mathbf{x})+\frac{\rho_x}{\pi}\int_\Omega \bar{L}(\mathbf{x}',\omega_i)(1-V(\omega_i))\max(\mathbf{N}_x\bullet\omega_i,0)d\omega_i$$

Recursion

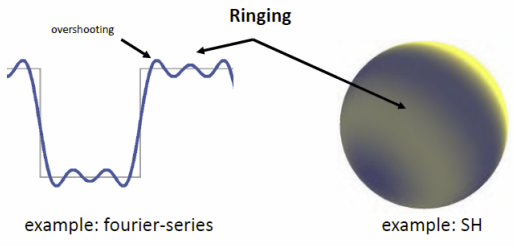




GIBBS' PHENOMENON: RINGING

Ringing is a problem that is caused by overshooting at jump discontinuities, this problem can be addressed by **windowing**, which means convolving with the window function (e.g. Hanning Window) to suppress the high frequencies that cause those problems.

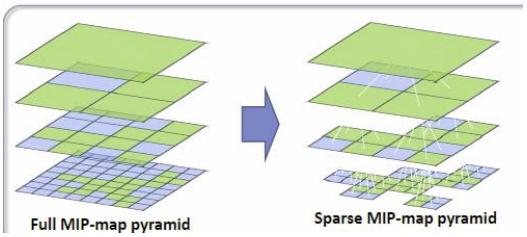
- overshooting at jump discontinuities



PRT SUMMARY

The **advantages** include that it is fast to render (1 dot product), it enables dynamic lighting environments and also supports complex light transfer like ambient occlusion and interreflections.

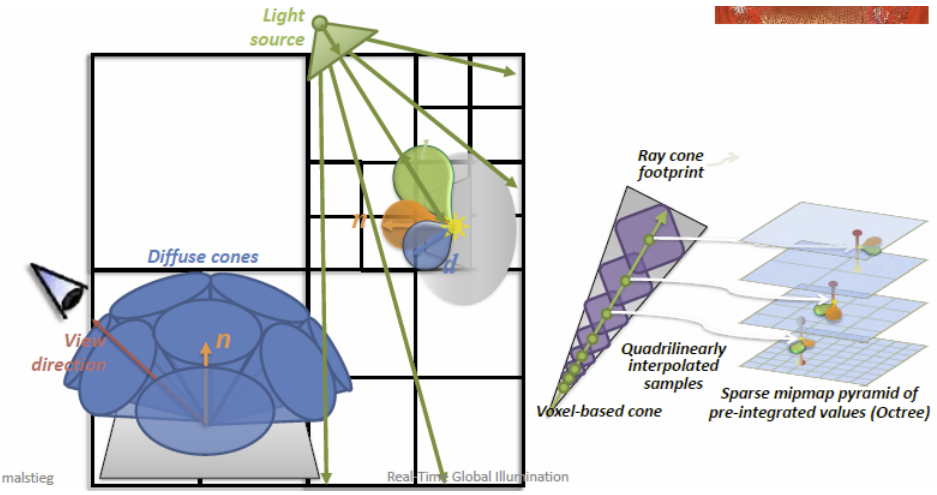
The **disadvantages** are that it is only efficient for lighting environments of low frequency (typically 9-16 SH coefficients) and can only be used with static scenes and diffuse surfaces.



VOXEL CONE TRACING

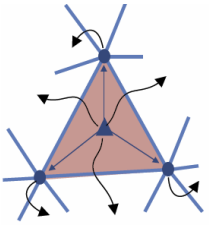
GigaVoxels is a sparse 3D MIP-map that can adapt its resolution. **Voxel cone-tracing** pre-filters geometry and irradiance and stores opacity and irradiance at all scales. Then it does a quadrilinear interpolation of the samples to reconstruct the continuous voxel size and position.

1. **Pass 1:** Lights pass, bake the irradiance and direction
2. **Pass 2:** Filtering pass, pre-filter values in the octree
3. **Pass 3:** Camera pass, this is done for each visible fragment, collect indirect diffuse and then the indirect specular part



SPATIAL DATA STRUCTURES

Triangle meshes are a compact form of describing geometry and can be **rendered efficiently** and also **manipulated efficiently**. Popular representations include independent triangles (**triangle soup**), **indexed triangle set** (store common vertices in a shared array, triangle index their vertices), **triangle strip/fan**, **data structures with neighborhood information** (support topological queries) and **winged-edge / half-edge data structures** (support arbitrary polygonal meshes and allow efficient query and storage.



A **connectivity data structure** is an extension of the indexed triangle set, where every triangle has references to its neighboring triangles and every vertex has a reference to its adjacent triangles. This enables walking through the mesh by iterating over all triangles of a vertex.

A **half-edge data structure** is a compact form of connectivity with no redundancy as it uses directed edges and stores data per half-edge: vertex attributes, pointer to next vertex, pointer to incident face and pointer to opposite half-edge.

Using a **B-rep**, it is also allowed and possible to represent polygon sets, that define no solid object.

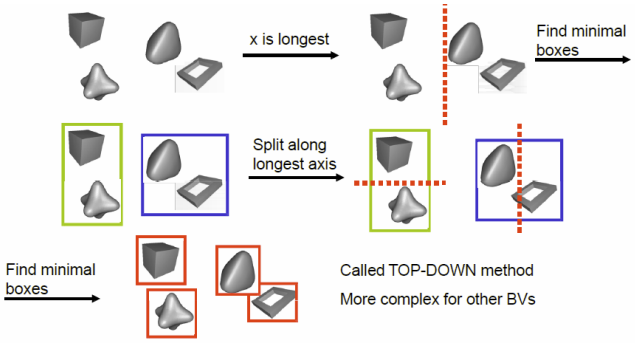
Spatial data structures now try to organize geometry and are used in every search related problem, as they enable faster processing and searching, this is done by **organizing the geometry in a hierarchy**. The geometric entities are stores typically in the CPU memory and there is traditionally no support on the GPU, but using a **compute language** like CUDA/OpenCL, some of these methods can be executed on the GPU.

REGULAR GRID

A **regular grid** is the simplest *spatial directory* and provides a **regular subdivision** with directly addressable cells. It is similar to a hashtable and allows for simply neighbor finding O(1). But the number of cells remains an issue, either too few or too many cells can cause problems, this can be solved by using a **hierarchical grid**.

BOUNDING VOLUME HIERARCHY

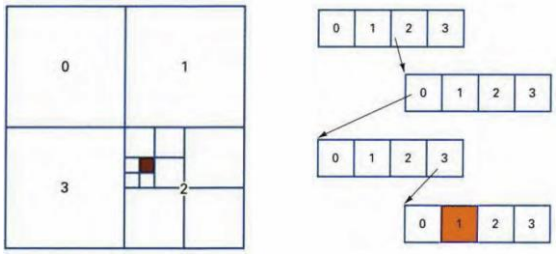
This is the most used structure in real-time graphics and stores information about the maximum extent of an object, meaning it encloses the object. The data structure is a **k-ary tree** and the **leaves hold** the **geometry**. The internal nodes have at most *k* children and have themselves like a sub-tree.



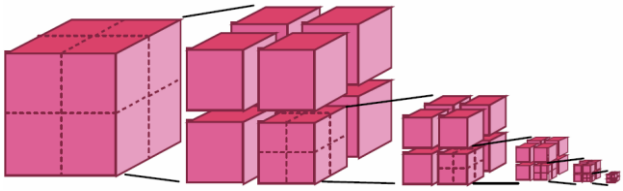
A **BVH** is created by finding the minimal box by splitting along the longest axis. This recursion stops when either the BV is empty, only one primitive is inside the BV, fewer than *n* primitives are inside the BV or the maximum recursion level is reached. Similar criteria hold for other BSP trees.

QUADTREE

A quadtree is a hierarchical subdivision of a 2D region, whereas the intermediate node divides the region into 4 quadrants.



OCTREE



An octree is built by using **recursive volumetric subdivision** and is used as a search structure for objects in other representations, e.g. B-Reps.

Extended Octrees can have additional node types like *face nodes* (contain a surface), *edge nodes* (contain an edge) or *vertex nodes* (contain a corner point).

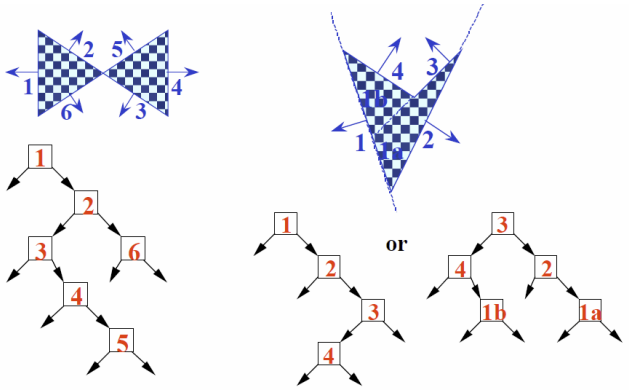
BINARY SPACE PARTITIONING (BSP) TREE

This is a special B-rep for quick rendering with visibility, this suits especially static scenes. The base plane of the polygon in a node partitions the space in two halves, everything in front of and behind the polygon. The left subtree of the node contains only polygons that are in front of the basis plane and the right subtree all polygons behind the basis plane. Polygons that lie in both halves are divided by the base plane into two parts.

GENERATION OF BSP TREES

The BSP tree is a linear list for convex objects, if they are not convex, the following algorithm is performed

1. Find the polygon who's plane intersects the fewest other polygons and cut these in two
2. Divide the polygon list into two sets, polygons in front of that plane and behind that plane
3. The polygon found in 1. Is the root of the BSP tree, the left and the right subtrees can be generated recursively.



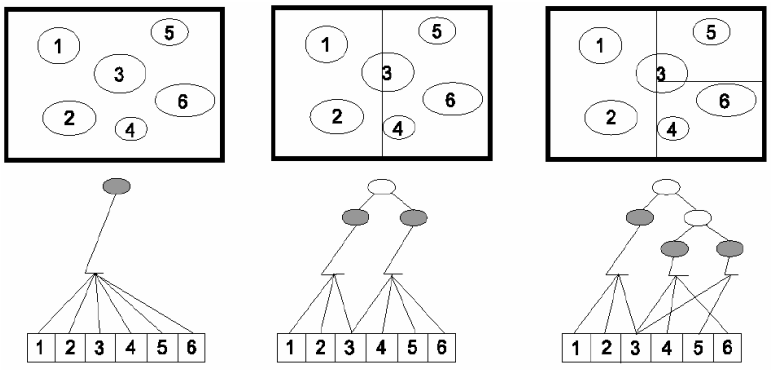
Operations with BSP trees include transformations, Points, plane equation and normal vector have to be transformed, and combinations, here perform combination with B-Rep, then generate the BSP tree. Combining BSP trees directly is faster.

PAINTER'S ALGORITHM

BSP trees are very good for fast rendering, one option is presented by the painter's algorithm.

```
IF eye is in front of a (in A+)
     draw all polygons of A-;
     draw a;
     draw all polygons of A+
ELSE
     draw all polygons of A+;
     (draw a);
     draw all polygons of A-;
```

KD TREE



The **kD tree** is a special case of a BSP tree and has only axis-aligned partitioning planes, the split direction is chosen, for example the longest side or to balance the number of objects in left and right sub-tree. A 1-D tree is a binary tree.

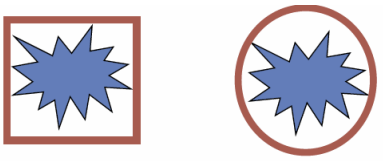
BINTREE

A bintree is a 3D tree, the separation plane is chosen for optimized (irregular) subdivision, and this leaves fewer nodes than the octree.

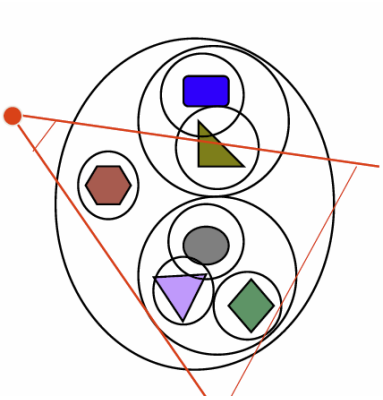
TOPOLOGICAL DATA STRUCTURES

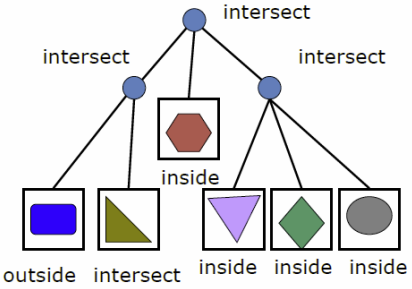
Topological data structures can be used for example for indoor environments, like for precomputing room-to-room visibility or the wayfinding of game characters. It can be represented by an **adjacency graph**.

BOTTOM-UP BOUNDING VOLUMES



These are bounding volumes, which are proxy shapes enclosing complex objects and are typically simple shapes to enable quick testing for intersection.





They also exists as hierarchical Bottom-Up BV's, where a hierarchy is present based on the bounding volumes. This can be used for view-frustum culling, as polygons that are outside of the view frustum can be eliminated.

SCENE GRAPH

BVH is used most often in real-time graphics as it is easy to implement and simple to understand, but it only contains geometric objects, this can be extended by a **scene graph**, which holds lights, transforms textures and more.

"… the gaming business on PC is growing by about a billion dollars every year."

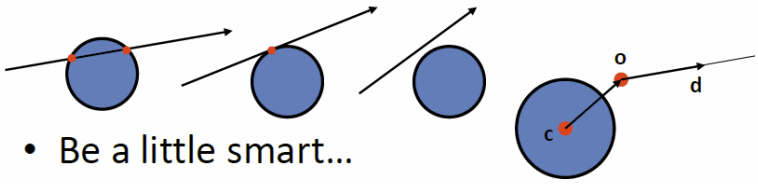
Richard Huddy | Gaming Scientist at AMD

INTERSECTION | COLLISION

Intersection testing is a component of fundamental importance as it tries to find, if and where a ray hits an object. It needs to be fast and is also needed for speed-up techniques and collision detection and there are various intersection techniques, with those many test can be derived easily, but often tricks are necessary to make them fast.

- Analytical
- Geometrical
- Separating axis theorem
- Dynamic tests

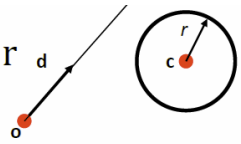
ANALYTICAL



- Be a little smart…

$(\mathbf{o}-\mathbf{c})\cdot\mathbf{d}>0$ ray points away from sphere

$(\mathbf{o}-\mathbf{c})\cdot(\mathbf{o}-\mathbf{c})-r^2<0$ o in sphere



Sphere center: $\mathbf{c}$, and radius r

Ray: $\mathbf{r}(t)=\mathbf{o}+t\mathbf{d}$

Sphere formula: $\|\mathbf{p}-\mathbf{c}\|=r$

Replace $\mathbf{p}$ by $\mathbf{r}(t)$, and square it:

$$(\mathbf{r}(t)-\mathbf{c})\cdot(\mathbf{r}(t)-\mathbf{c})-r^2=0$$

$$(\mathbf{o}+t\mathbf{d}-\mathbf{c})\cdot(\mathbf{o}+t\mathbf{d}-\mathbf{c})-r^2=0$$

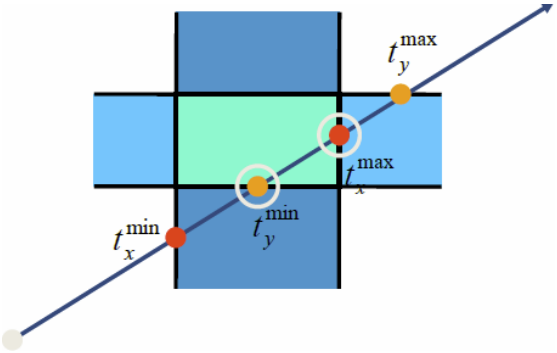
$$(\mathbf{d}\cdot\mathbf{d})t^2+2((\mathbf{o}-\mathbf{c})\cdot\mathbf{d})t+(\mathbf{o}-\mathbf{c})\cdot(\mathbf{o}-\mathbf{c})-r^2=0$$

$$t^2+2((\mathbf{o}-\mathbf{c})\cdot\mathbf{d})t+(\mathbf{o}-\mathbf{c})\cdot(\mathbf{o}-\mathbf{c})-r^2=0 \quad \|\mathbf{d}\|=1$$

GEOMETRICAL

Boxes and spheres are often used as bounding volumes, a **slab** is the volume between two parallel planes and a **box** is the logical intersection of 2 slabs in 2D and 3 slabs in 3D.

The **maximum** of t^{min} and t^{max} is kept and if $t^{min} < t^{max}$, then there is an **intersection**. A special case occurs, if the ray is parallel to the slab.

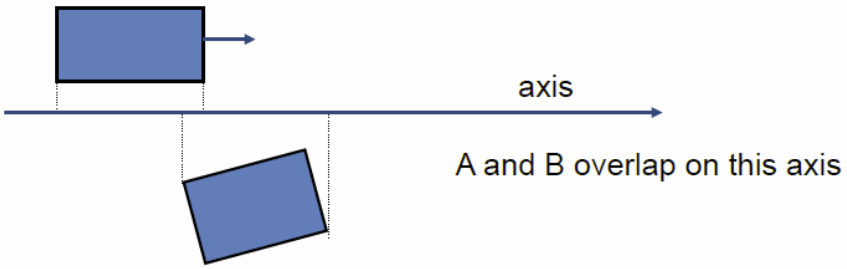


SEPARATING AXIS THEOREM (SAT)

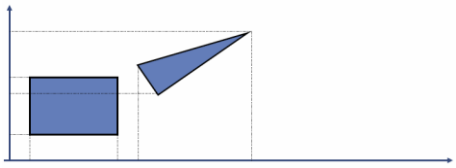
The **Separating Axis Theorem** states the following

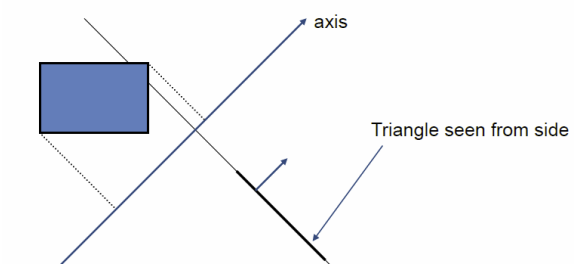
Two convex polyhedral, A and B, are disjoint, if any of the following axis separate the objects:

1. *An Axis orthogonal to a face of A*
2. *An Axis orthogonal to a face of B*
3. *An Axis formed from the cross product of one edge from each of A and B*



Given this example, the box is axis-aligned, therefore start out by testing the axis that are orthogonal to the faces of the box, in this case that means testing x, y and z.





Assuming they overlapped on x, y and z we must continue testing and now test all the axis orthogonal to the face of the triangle.

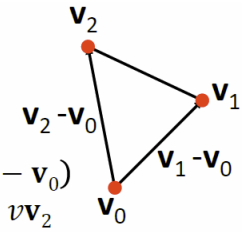
If still no separating axis has been found test the cross product of an edge of the box and an edge from the triangle and then do this for all such combinations. If there is **at least one separating axis**, then the **objects don't collide**, otherwise they overlap.

RULES OF THUMB FOR INTERSECTION TESTING

Try to do **acceptance and rejection tests** as **early** as possible to make a fast exit, **postpone expensive calculations** as long as possible, use **dimension reduction** and **share computations between objects** if possible.

RAY / TRIANGLE

- Ray: $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$
- Triangle vertices: $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2$
- A point in the triangle:
- $\boldsymbol{t}(u,v) = \mathbf{v}_0 + u(\mathbf{v}_1 - \mathbf{v}_0) + v(\mathbf{v}_2 - \mathbf{v}_0)$
  $= (1-u-v)\mathbf{v}_0 + u\mathbf{v}_1 + v\mathbf{v}_2$
- $u, v \geq 0$, and $u + v \leq 1$
- Set $\mathbf{t}(u,v) = \mathbf{r}(t)$, and solve!



$$\begin{pmatrix} | & | & | \\ -\mathbf{d} & \mathbf{v}_1 - \mathbf{v}_0 & \mathbf{v}_2 - \mathbf{v}_0 \\ | & | & | \end{pmatrix} \begin{pmatrix} t \\ u \\ v \end{pmatrix} = \begin{pmatrix} | \\ \mathbf{o} - \mathbf{v}_0 \\ | \end{pmatrix}$$

$\mathbf{e}_1 = \mathbf{v}_1 - \mathbf{v}_0 \quad \mathbf{e}_2 = \mathbf{v}_2 - \mathbf{v}_0 \quad \mathbf{s} = \mathbf{o} - \mathbf{v}_0$

- Solve with Cramer's rule:

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{\det(-\mathbf{d}, \mathbf{e}_1, \mathbf{e}_2)} \begin{pmatrix} \det(\mathbf{s}, \mathbf{e}_1, \mathbf{e}_2) \\ \det(-\mathbf{d}, \mathbf{s}, \mathbf{e}_2) \\ \det(-\mathbf{d}, \mathbf{e}_1, \mathbf{s}) \end{pmatrix}$$

$$x_i = \frac{\det(A_i)}{\det(A)}$$
for Ax=b
(A_i = A with b at column i)

Use this fact : $\det(\mathbf{a}, \mathbf{b}, \mathbf{c}) = (\mathbf{a} \times \mathbf{b}) \cdot \mathbf{c} = -(\mathbf{a} \times \mathbf{c}) \cdot \mathbf{b}$

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{(\mathbf{d} \times \mathbf{e}_2) \cdot \mathbf{e}_1} \begin{pmatrix} (\mathbf{s} \times \mathbf{e}_1) \cdot \mathbf{e}_2 \\ (\mathbf{d} \times \mathbf{e}_2) \cdot \mathbf{s} \\ (\mathbf{s} \times \mathbf{e}_1) \cdot \mathbf{d} \end{pmatrix}$$

- Share factors to speed up computations

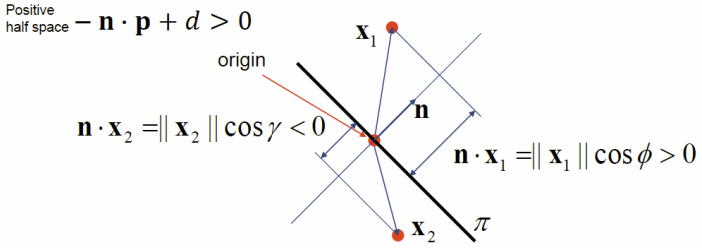
The goal is always to computer as little as possible and then test.

POINT / PLANE

- Plane: $\mathbf{n} \cdot \mathbf{p} + d = 0$
- Insert a point x into plane equation:

Negative half space $-\ \mathbf{n} \cdot \mathbf{p} + d < 0$

Positive half space $-\ \mathbf{n} \cdot \mathbf{p} + d > 0$



Sphere/Plane
AABB/Plane

Plane: $\pi : \mathbf{n} \cdot \mathbf{p} + d = 0$
Sphere: $\mathbf{c}$ r
Box: $\mathbf{b}^{min}$ $\mathbf{b}^{max}$

Sphere: compute $f(\mathbf{c}) = \mathbf{n} \cdot \mathbf{c} + d$

- $f(\mathbf{c})$ is the signed distance ($\mathbf{n}$ normalized)
- $abs(f(\mathbf{c})) > r$ no collision
- $abs(f(\mathbf{c})) = r$ sphere touches the plane
- $abs(f(\mathbf{c})) < r$ sphere intersects plane

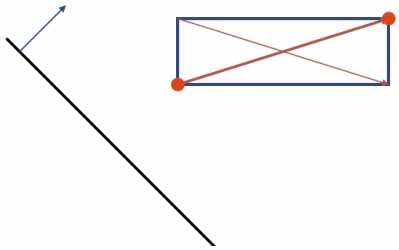
Box: insert all 8 corners

- If all f's have the same sign, then all points are on the same side, and no collision

AABB/Plane

Plane: $\pi : \mathbf{n} \cdot \mathbf{p} + d = 0$
Sphere: $\mathbf{c}$ r
Box: $\mathbf{b}^{min}$ $\mathbf{b}^{max}$

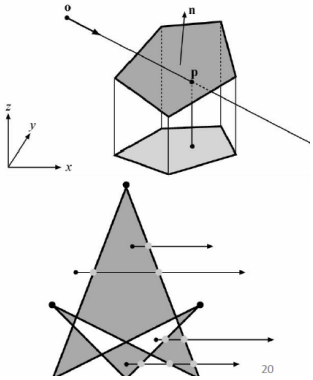
- The smart way (shown in 2D)
- Find diagonal that is most closely aligned with plane normal



Need only test the red points

Ray/Polygon: Very Briefly

- Intersect ray with polygon plane
- Project from 3D to 2D
- How?
- Find $\max(|n_x|,|n_y|,|n_z|)$
- Skip that coordinate
- Count crossings in 2D



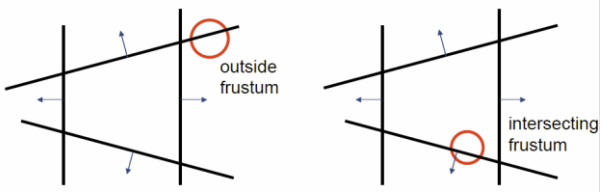
r Schmalstieg Real-Time Graphics 20

VOLUME/VOLUME TESTS

Volume/Volume tests are used in **collision detection**, for sphere/sphere the squared distances between the sphere centers can be calculated and then compared to the squared sum of the radii of the two spheres. For Axis-aligned bounded boxes (AABB) the **test is done simply in 1D** and for oriented bounding boxes **SAT** can be used.

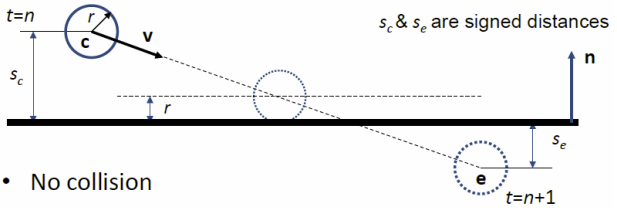
VIEW FRUSTUM TESTING

The view frustum is composed of six planes, these can be created from the projection matrix and objects are then **tested against all six frustum planes**. All positive half spaces are defined as *outside frustum* and are not dealt with here. If the object is in positive half space and the distance from the plane is smaller than *r*, than there is no intersection. It the object is intersecting a plane or in negative half space, then continue and if not outside after all six planes, then the object is either inside or intersecting. This is **not an exact test, but not incorrect**, as spheres, that are reported to be inside, can be outside but not vice versa.



DYNAMIC INTERSECTION TESTING

Testing is often done every rendered frame, meaning at discrete time intervals and this can possibly result in unwanted *quantum effects*. **Dynamic testing** deals with this problem by looking at the time interval more closely, this is more expensive.



- No collision
  - Must be on the same side of the plane ($s_c s_e$>0)
  - Must be $|s_c|>r$ and $|s_e|>r$
- Otherwise, sphere can move $|s_c|-r$
- Time of collision: $t_{cd} = n + \frac{s_c - r}{s_c - s_e}$
- Response: reflect **v** around **n**, move (1-t_{cd})**r**

DYNAMIC SEPARATING AXIS THEOREM

With **SAT**, one axis at a time is tested to find an overlap. This is the same with **DSAT**, but here the *projection on the axis needs to be adjusted, so that the interval moves on the axis as well*. The same axis as with SAT need to be tested and the same overlap/disjoint criteria hold.

COLLISION DETECTION

Collision detection is not really a computer graphics topic, but necessary for realism in games, movies and also needed to avoid the previously mentioned *quantum effects* (objects passing through objects). It uses **spatial data structures** and uses **intersection testing**. The major parts include **collision detection** (is there a collision), **collision determination** (where is it) and **collision response**. Rays can be used for simple applications but BVH is used to test two complex objects against each other. Approaches for collision detection include

- **Ray Tracing:** This is simple and fast, but not very accurate.
- **Bound volume hierarchies:** More accurate and can compute exact results, but more complicated and slower
- **Efficient CD for several hundreds of objects**

COLLISION DETECTION WITH RAYS

The idea is to **approximate** a complex objects with a set of rays. For example, to do collision detection for a car, rays can be attached at each wheel and the closest intersection distance between ray and road geometry can be computed. If this distance is 0, the car is on the road, if it's greater than 0 it is flying above the road and if it is smaller, it is ploughing deep in the road, these values can be used to compute a simple collision response.

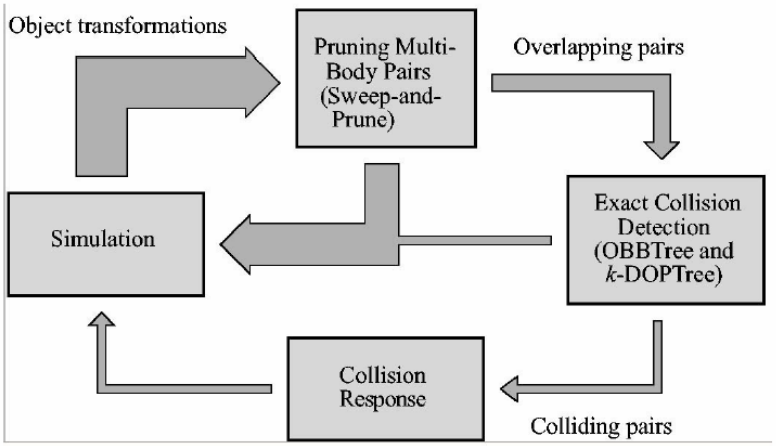
Now the car is simplified via rays, but not the road, it can be simplified by turning to **spatial data structures** like a BVH or BSP tree.

DIMENSIONALITY REDUCTION

Sometimes 3D can be turned into 2D operations, in an example of a *maze* and a human walking through it, the human can be approximated by a circle and the lines of a maze can be moved outwards by the circle radius, then only the center of the circla needs to be tested against the moved walls.

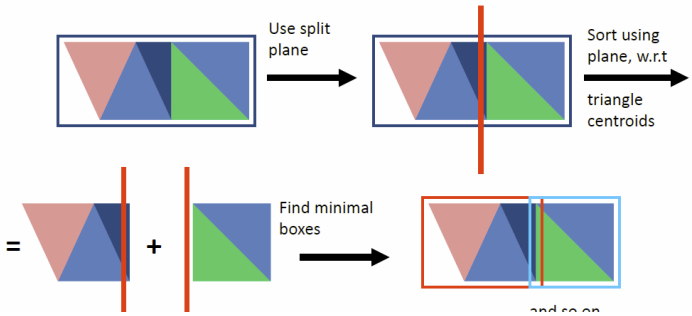
Here, the BV of each object is tested against the BV's of all the other objects, this may work for smaller sets, but requires way to many tests, the need for smarter and scalable methods arises.

If an accurate result is needed, separate BVHs for the objects are used and each BHV is tested against the other BVH's for overlap. When the triangles overlap, the also compute the exact intersection if needed.



BVH BUILDING EXAMPLE

- Can split on triangle level as well



Pseudo Code for BVH against BVH

```
FindFirstHitCD(A, B)
returns ({TRUE, FALSE});
1 :  if(isLeaf(A) and isLeaf(B))
2 :     for each triangle pair T_A ∈ A_c and T_B ∈ B_c
3 :        if(overlap(T_A, T_B)) return TRUE;
4 :  else if(isNotLeaf(A) and isNotLeaf(B))
5 :     if(Volume(A) > Volume(B))
6 :        for each child C_A ∈ A_c
7 :           FindFirstHitCD(C_A, B)
8 :     else
9 :        for each child C_B ∈ B_c
10 :          FindFirstHitCD(A, C_B)
11 : else if(isLeaf(A) and isNotLeaf(B))
12 :    for each child C_B ∈ B_c
13 :       FindFirstHitCD(C_B, A)
14 : else
15 :    for each child C_A ∈ A_c
16 :       FindFirstHitCD(C_A, B)
17 : return FALSE;
```

Pseudocode deals with 4 cases:

1) Leaf against leaf node
2) Internal node against internal node
3) Internal against leaf
4) Leaf against internal

When looking at the pseudo code, it terminates, when the 1st colliding triangle pair is found. It is also simple to modify the code to continue traversal and put each pair of triangles in a list. It is also reasonable simple to include rotations for objects as well.

TRADE-OFFS

There are several choices available for BV, for example **AABB, OBB, k-DOP, sphere**… In general, the tighter the BV, the slower the test but a less tight BV gives more triangle-triangle tests in the end

Cost function: $t = n_v c_v + n_p c_p + n_u c_u$

n_v : number of BV/BV overlap tests
c_v : cost for a BV/BV overlap test
n_p : number of primitive pairs tested for overlap
c_p : cost for testing whether two primitives overlap
n_u : number of BVs updated due to the model's motion
c_u : cost for updating a BV

This system is called a **first-level CD** and is executed, because we want to execute the **second-level CD** less frequently.

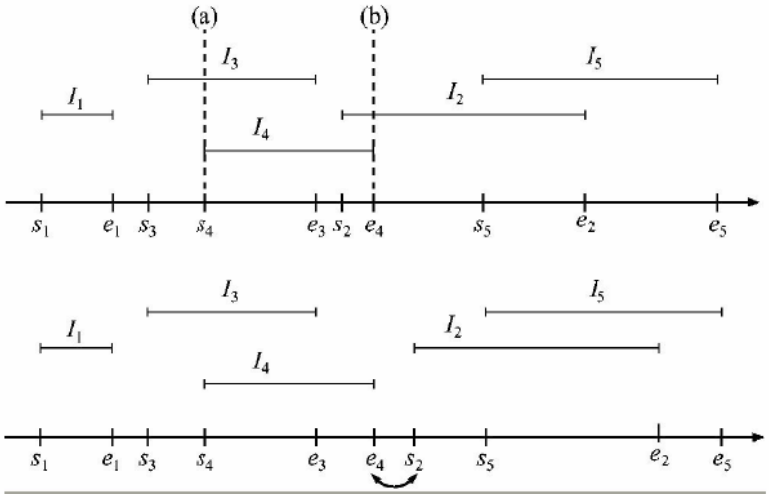
SWEEP-AND-PRUNGE ALGORITHM

- Assume objects may translate and rotate
- Then we can find a minimal bbox, which is guaranteed to contain object for all rotations
- Do collision overlap three times
  - Once for *x*, *y* and *z*-axes
- Each cube on this axis is an interval from s_i to e_i (i = index of bbox)

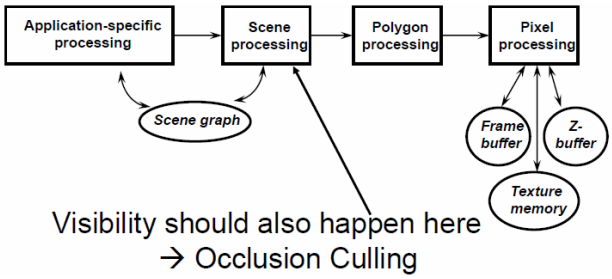
- Sort all s_i and e_i into a list
- Traverse list from start to end
- When an s is encounted, mark corresponding interval as active in an **`active_interval_list`**
- When an e is encountered, delete the interval in **`active_interval_list`**
- All intervals in **`active_interval_list`** are overlapping!

- Now sorting is expensive: $O(n \cdot \log n)$
- But, exploit frame-to-frame coherency!
- The list is not expected to change much
- Therefore, "resort" with bubble-sort or insertion-sort
- Expected: $O(n)$

- Keep a Boolean for each pair of intervals
- Invert when sort order changes
- If all Boolean for all three axes are true → overlap

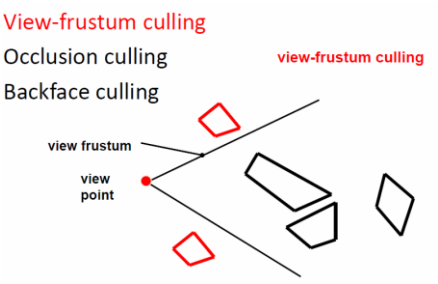


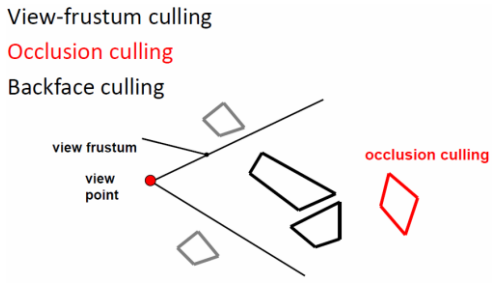
VISIBILITY

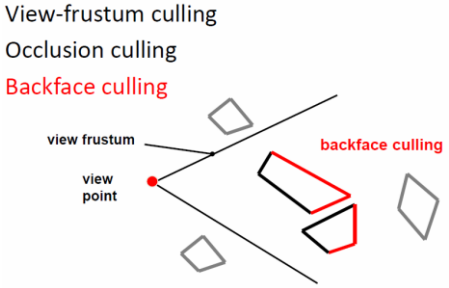


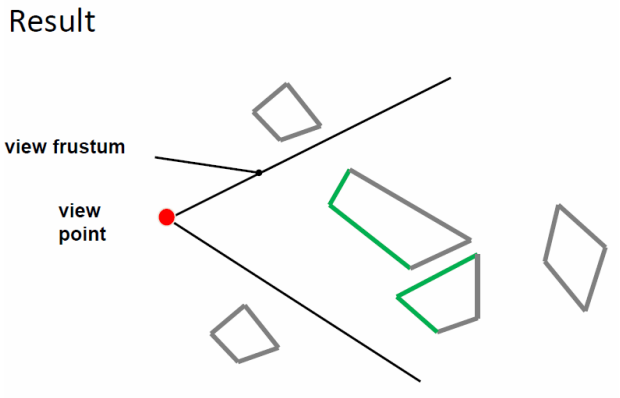
In large models with millions of polygons, some form of acceleration is necessary to enable real-time performance. The **Z-Buffer** alone is not enough, as it does **not eliminate depth-complexity** and also does **not eliminate vertex processing of occluded polygons**. Visibility is researched in many areas, including Computer Graphics, Computer Vision, Robotics … and used in *occlusion culling*, for *shadows and global illumination* (which objects are seen by light source), in *hidden-surface removal* (which objects are unseen by user), in *viewpoint selection* (determine viewpoints that can see entire scene) and *image-based rendering* (determine set of viewpoints that see entire scene).

VISIBILITY CULLING

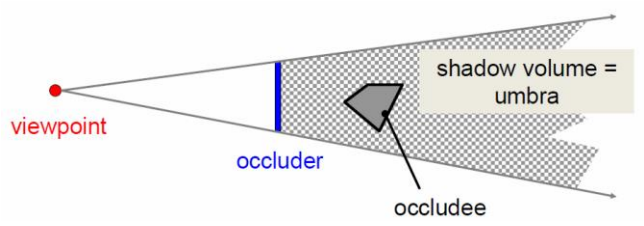








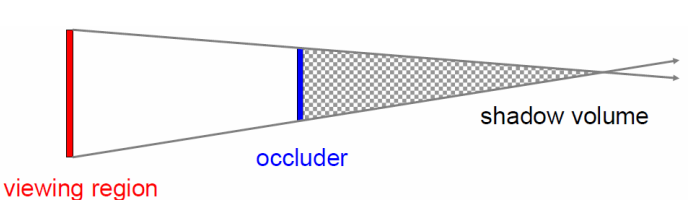
VISIBILITY FROM A POINT

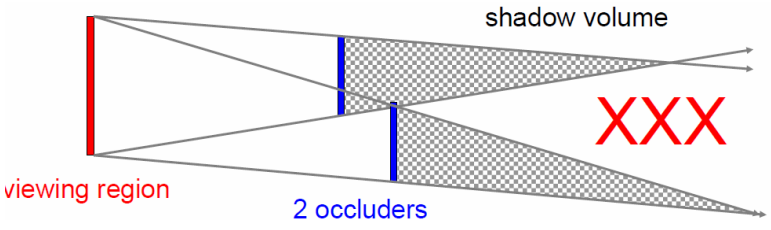


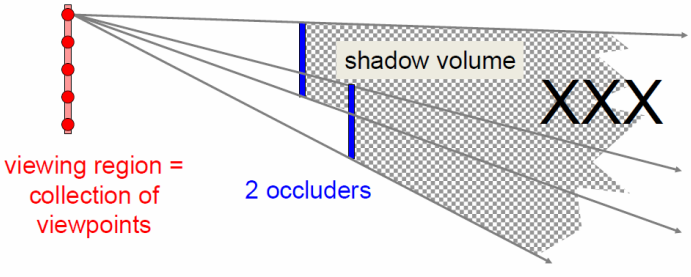
The complete shadow volume for a number of occluders is given as the **union** of the individual shadow volumes, this is called **occluder fusion**, which describes the effect of multiple occluders

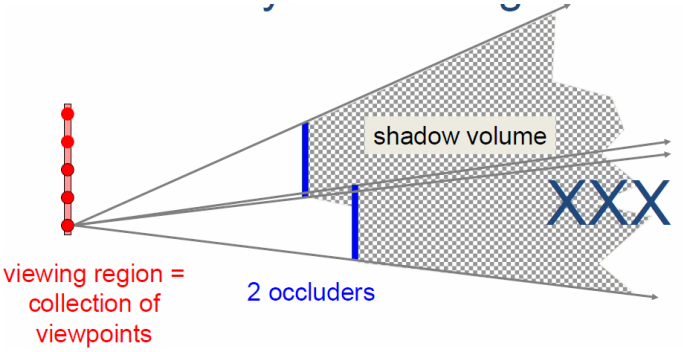
A simple algorithm can be used for point visibility, an empty **Shadow volume data structure (SVDS)** is used and for each occluder the shadow volume SV is calculated and added to the SVDS and each objects is tested against the SVDS and culled if occluded.

VISIBILITY FROM A REGION







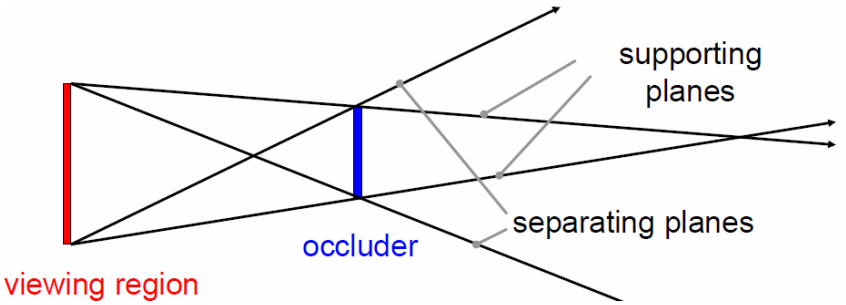


The given Area **XXX** is always occluded, here the shadow volume is more than the union of the individual shadow volumes.

BOUNDING AND SEPARATING PLANES

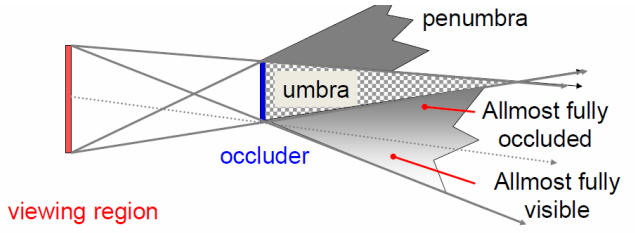
The **separating planes** of two convec polyhedral are formed by the edge of the first polyhed + a vertex of the second polyhed, whereas the polyhedral lie on opposite sides of the plane.

The **supporting planes** have both polyhedral on the same side.



UMBRA AND PENUMBRA

Umbra describes the full shadow while the **penumbra** describes the half-shadow. Because of that, umbra is a simple in/out classification while the penumbra additionally encode which parts of the viewing region are visible.



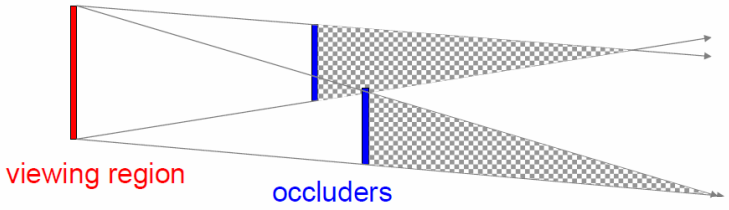
OCCLUSION CULLING FROM A REGION

The umbra can be calculated as the regions that are fully occluded, therefore, the umbra is the sum of all umbras of the individual occluders + all regions where the penumbra merges to umbra. But the question remains how to deal with the penumbra.

Idea 1: Ignore Penumbra

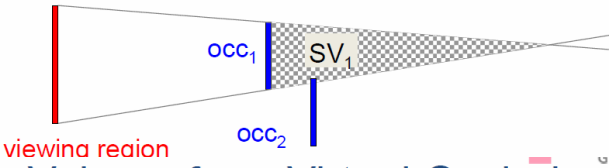
- Shadow volume data structure (SVDS) = empty
- For each occluder occ_i
  - Calculate shadow volume SV_i
  - Add SV_i to SVDS
- Test the scene against the SVDS

→ *Problem: little occlusion success*



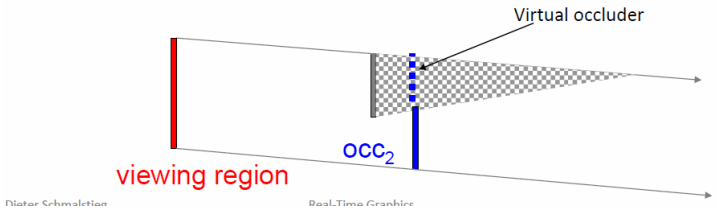
Idea 2: Detect Overlapping Umbra

- Shadow volume data structure = empty
- Front-to back: for each occluder occ_i
  - Expand Occluder inside existing shadow volume as far as possible
  - Calculate shadow volume SV_i
  - Add SV_i to shadow volume data structure
- Test the scene against the SVDS



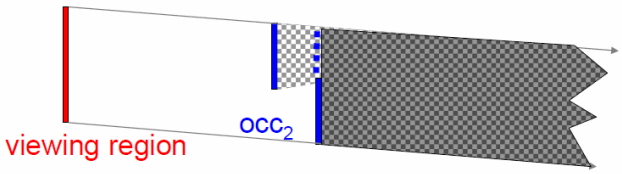
Virtual Occluders

- Shadow volume data structure = empty
- front-to back: for each occluder occ_i
  - **Expand Occluder inside existing shadow volume as far as possible**
  - Calculate shadow volume SV_i
  - Add SV_i to shadow volume data structure
- Test the scene against the SVDS



Shadow Volume from Virtual Occluder

- Shadow volume data structure = empty
- front-to back: for each occluder occ_i
  - Expand Occluder inside existing shadow volume as far as possible
  - **Calculate shadow volume SV_i**
  - Add SV_i to shadow volume data structure
- Test the scene against the SVDS



The 3rd idea is to calculate everything by looking at the shape of the surface that bounds the umbra, but this is really difficult to compute analytically and is done in practice with heuristics or (sub-) samples.

POTENTIAL VISIBLE SET (PVS)

The exact visible set is unknown as it is computationally infeasible, but the potentially visible set (PVS) is the set of objects that could be visible, this can be precomputed with varying detail, the exact hidden surface removal can be done by z-buffer.

OCCLUSION CULLING IN PRACTICE

Two hierarchical spatial data structures are used, first the **scene data structure (SDS)** is built from the objects and then the **shadow volume data structure (SVDS)** is generated from the occluders and the virtual occluders, after the the SDS can by culled using the SVDS.

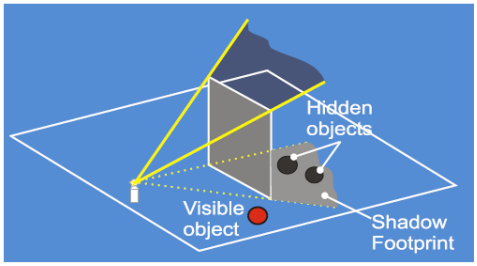
OCCLUSION CULLING – GENERAL IDEA

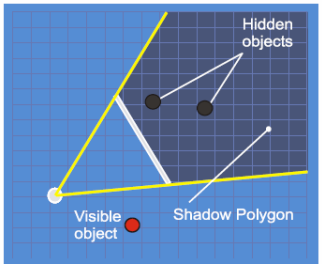
The general idea is to **traverse the SDS** top-down / front-back and test each node against the SVDS for visibility. If a node is invisible, it is skipped, otherwise traversal either continues or the objects are marked visible and the new occluders are inserted into the SVDS.

Occlusion culling can also be accelerated, one way is to use **2.5D occlusion culling**, where 2.5D visibility algorithms are used, the SDS is still 3D but the SVDS is only 2.5D, other include occluder selection, lazy update…

OCCLUDER SHADOWS

Often shadow frusta of occluders can be defined by one polygon, now the bird's eye view of the shadows can be rendered and used to cull invisible objects.





By doing this, all objects in the scene are entered into a regular grid and all occluder polygons are rendered into a bitmap, this can then be used for the occlusion culling.

OCCLUDER SELECTION

It is typically costly to use all scene polygons as occluders, therefore the idea is to **select only a subset of the polygons** that are close to the view point, occlude a large area and are facing the view point.

LAZY UPDATE OF SVDS

Using the z-buffer as SVDS is a popular approach but costly and it is also not trivial to update the hierarchy of the SVDS. The idea is now to do **lazy updates**, meaning insert many occluders into the SVDS and then updating the SVDS, this is sometimes done only once for the whole algorithm.

APPROXIMATE FRONT-TO-BACK SORTING

An exact front-to-back sorting is expensive, it is possible to use an approximation of that (but don't calculate incorrect occlusion, especially for visibility from a region).

HARDWARE OCCLUSION QUERY

A Z-Pyramid is a hierarchical z-buffer, where the lowest level is a full-resolution z-buffer and the higher levels represent the maximum depth of the four pixels "underneath" it. But this cannot be implemented in hardware.

It is possible to use the standard z-buffer instead and use only the lowest level of the hierarchy, here the SDS may consist of whatever but the SVDS is the z-buffer. All of this is only efficient with **hardware occlusion query**.

It provides an interface to issue **multiple queries at once** before asking for the result of any one, the result is the **pixel count** (number of pixels that pass). The application can then overlap the time it takes for the queries to return with other work, increasing the parallelism between CPU and GPU.

PVS PRECOMPUTING FOR REGIONS

The goal is to divide the view space into view cells and to calculate the PVS for each view cell and store those on disk.

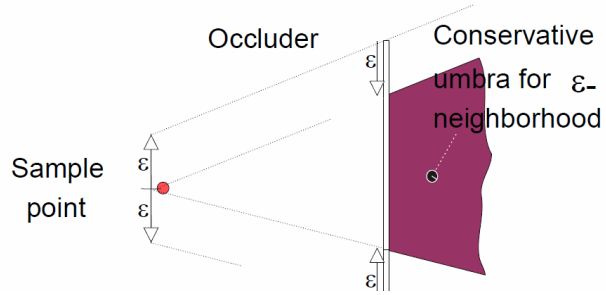
CELLS AND PORTALS

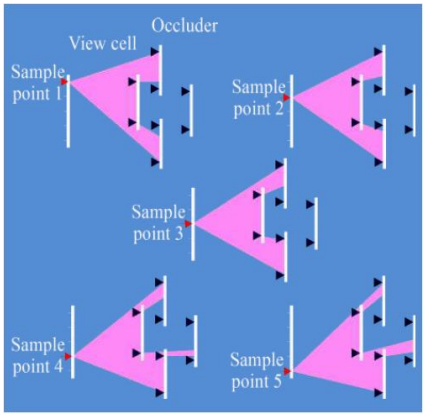
The idea is to build an **adjacency graph**, where the nodes are the cells and the edges are the portals, now the sightlines through an oriented portal sequence is tested via a depth search in the adjacency graph.

Screen-Space portals means projecting the portal sequence to the screen and intersecting the screen space boxes with the portal sequence and then do a depth search, which terminates when the intersection becomes 0.

OCCLUDER SHRINKING

This is a conservative occlusion culling for a region this is done by shrinking all occluders and then for each view cell and each sample point the PVS is calculated, then the union of all PVS is calculated.





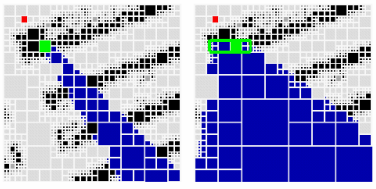
SHAFTS: OCCLUSION TRACKING

Axis-aligned boxes are used as view cells and occluders, a hidden region is therefore constructed from visual events, and this is the shaft. The Voxels are classified as hidden in the discretization.

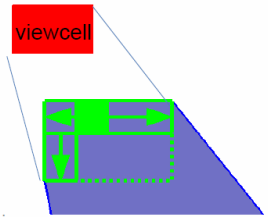
BLOCKER EXTENTION

A hierarchical data structure is used and the boundaries are rasterized, opaque voxels equal occluders.

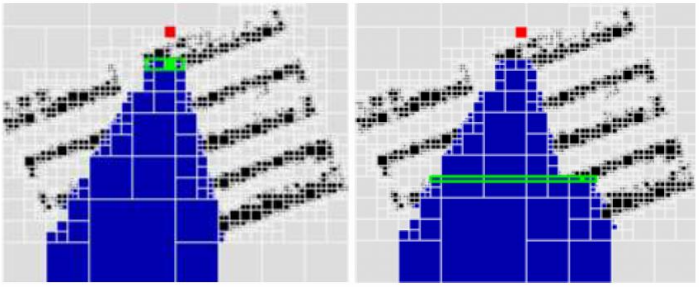
Extend into adjacent opaque space



L-shape preserves box-shape

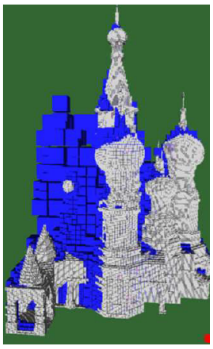


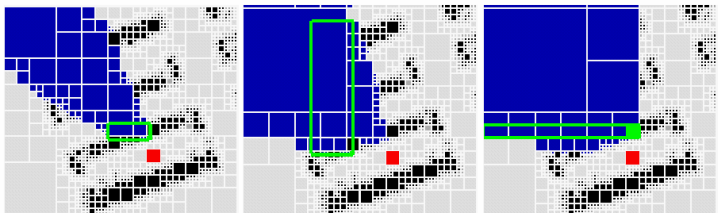
- Treat hidden space as opaque
- Virtual occluder idea



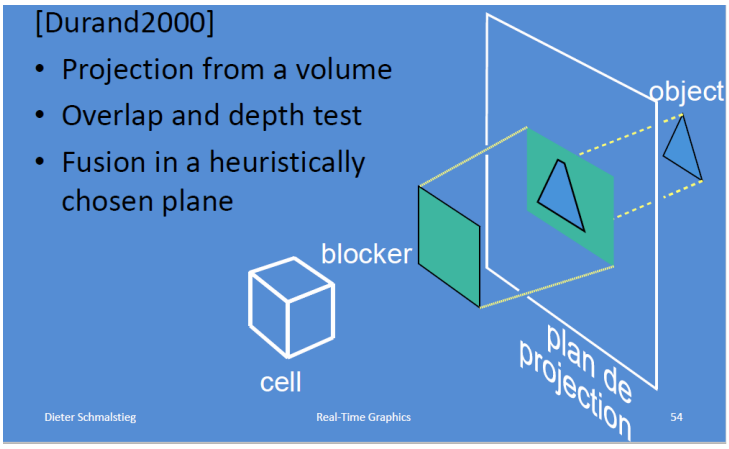
Greedy occlusion

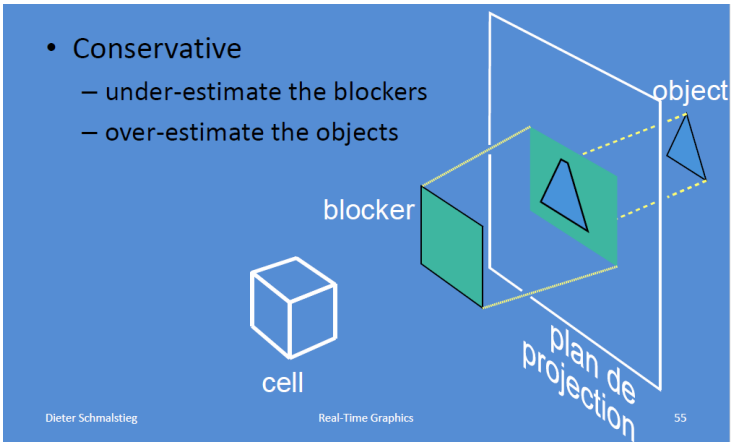
- Iterate over opaque voxels
- Greedily search for large virtual occluders
- Mark space hidden by them

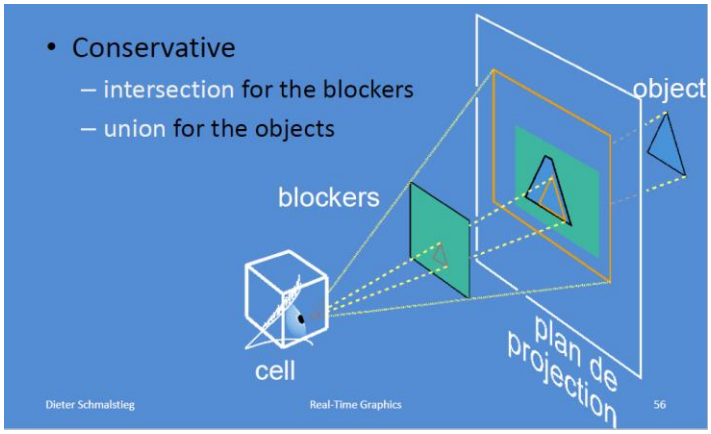




EXTENDED PROJECTIONS







LEVEL OF DETAIL

Even after dealing with visibility, the problem remains, that models may contain too many polygons. The **idea** is now to simply the amount of detail to render small or distant objects, this is known as **level of detail (LOD)**, and other words for the same phenomenon are multiresolution modeling, polygonal simplification, geometric simplification, mesh reduction....

TRADITIONAL APPROACH

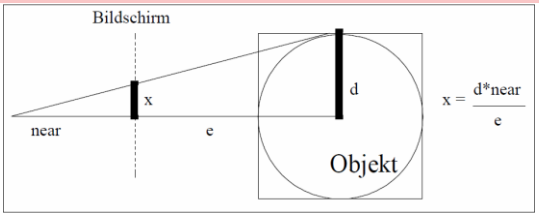
The traditional approach is to create **levels of detail** for each object in a preprocessing step, at runtime coarser LODs can be used.

Issues with this system can be found in

- **Runtime system**
  - LOD framework: Which LODs are eligible
  - LOD selection: Criteria for which LODs are selected
  - LOD switching: How to avoid artifacts when switching
- **LOD generation**
  - Simplification methods: How to reduce polygons
  - Error measures: Which polygons to reduce

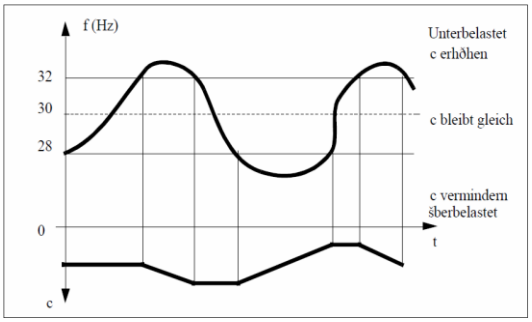
LOD SELECTION

STATIC LOD SELECTION



Here, the selection of the LODs happens according to the size of the object on the image, here it is not possible to control the frame rate.

REACTIVE LOD SELECTION



Here the object size is multiplied with a factor *c*, if the frame rate is too low, *c* is decreased, otherwise increased. This results in a roughly constant frame rate. But problems can occur, if complex objects suddenly become visible, this requires a hysteresis.

PREDICTIVE LOD SELECTION

The goal is to reach the vest possible image quality, therefore it tries to maximize the **benefit** (contribution of an object to the image quality) but while keeping the **cost** (time for drawing object with a given LOD) smaller than the frame time. This is an **optimization problem**.

LOD SWITCHING

There are different types of **switching between LOD**'s, there is **hard switching** (simple, but popping artifacts), **blending** (works well for all types, but has problems with transparences, shadows and temporarily increases the rendering load) and **geomorphing** (best quality but requires geometric correspondence between LODs).

LOD CREATION

The goal is to simplify the given models via **simplification methods = "operators"**, the question remains now what criteria should be used to guide simplification. Visual or perceptual criteria are hard to select, geometric criteria are more common for this task.

LOD BY SUBDIVISION SURFACES

Curved surfaces can be defines by repeated subdivision steps on a polygonal model, those subdivision rules create new vertices, edges and faces based on neighboring features and can be **computed in the geometry shader**.

LOD BY SHADING AND RENDERING

Textures and impostors can be used instead of geometry to reduce the rendering cost.

LOD BY GEOMETRIC SIMPLIFICATION

The polygonal geometry of small or distant objects is simplified, but this does not change the rasterization, as the fragment shader load remains roughly identical.

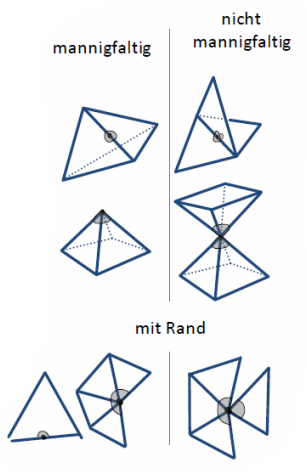
POLYGONAL MESHES

Meshes are defined by their **geometry** and **topology**, by using **triangle meshes**, a surface approximation of arbitrary accuracy is possible via a simple mathematical description and there exists efficient hardware support. The requirements on meshes include the ability to search for object details, wrap around a certain space and certain requirements for geometric algorithms. The information is encoded in two categories, the **topology** describes how the triangles are connected, therefore the inherent structure of the surface and independent of the actual representation in 3D, and the **geometry** that explains the actual location in 3D space.

MANIFOLD MESHES

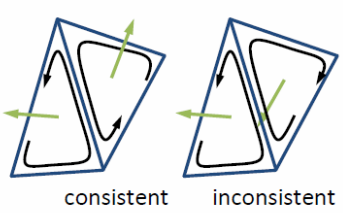
Manifoldness is a **strong requirement**, as there are no *special points*, every edge is adjacent to exactly two triangles and every vertex must be part of a loop of triangles.

Manifolds **with border** have weaker requirements to accommodate borders, whereas every edge is adjacent to one or two triangles.



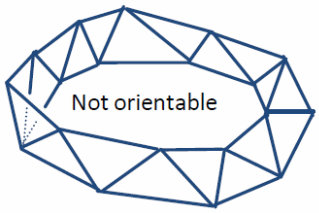
TOPOLOGICAL VALIDITY

A **consistent orientation** is important to decide, which side is front/outside or back/inside, the winding order of a polygon describes what is outside. In triangle meshes, adjacent triangles must agree about inside and outside, but this is **not always possible**.



GEOMETRIC VALIDITY

Self-penetration is undesirable, but this is hard to ensure, as far apart portions of objects may intersect.



SIMPLIFICATION FOR LOD

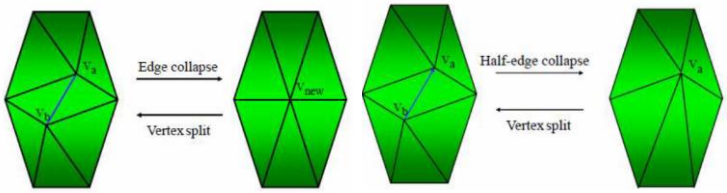
The goal is to simply the geometry (and possibly the topology) and also reduce the genus (number of holes) and work with non-manifold meshes.

SIMPLIFICATION OPERATORS

There are two forms of simplification, **local geometry simplification** (iteratively reducing the number of geometry primitives, also simplifying the topology and reducing the number of holes) and **global geometry simplification**.

LOCAL GEOMETRY SIMPLIFICATION

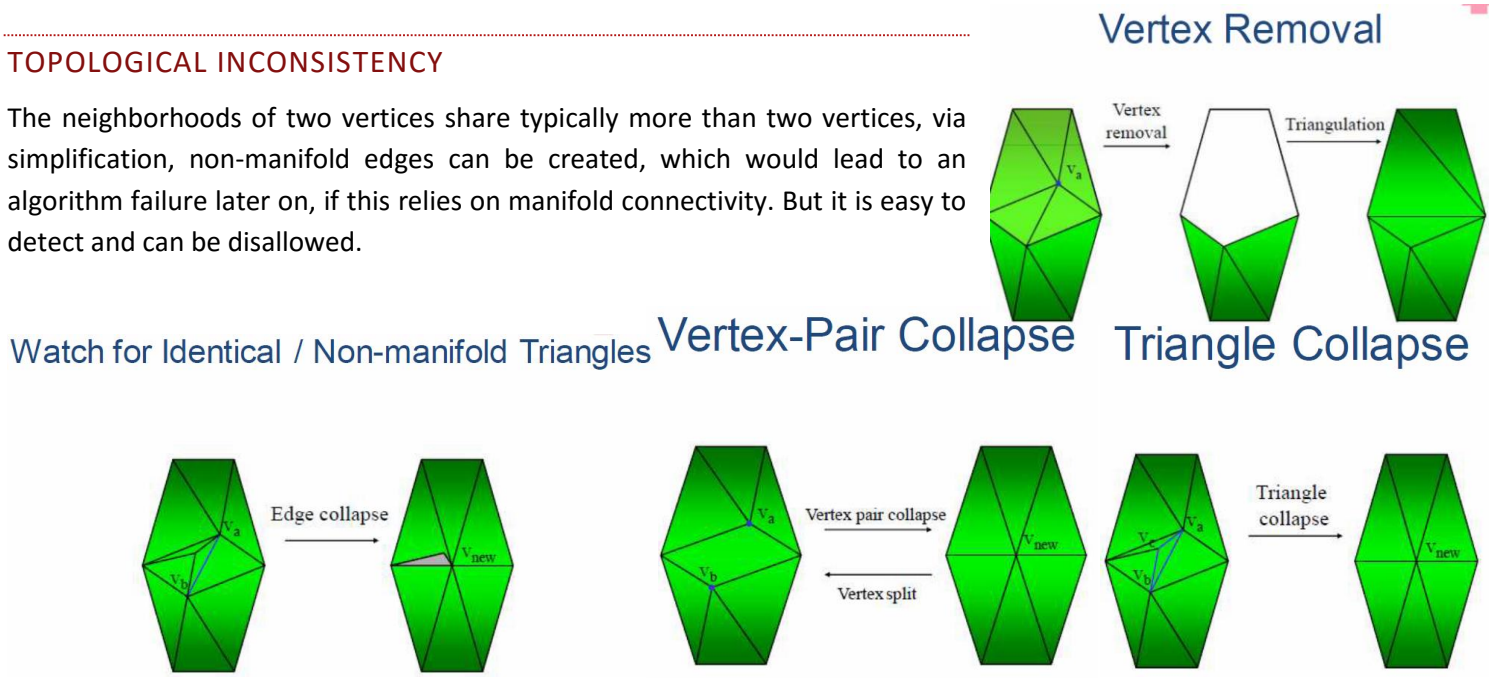
When reducing the local geometry, edges, vertex-pairs, triangles and or cells may collapse, it is also possible to remove vertices and do more general geometric replacements.



Mesh fold-overs pose problems, which ca be avoided by calculating the adjacent face normals and then testing, if they would flip after simplification. If so, this particular simplification can be weighted heavier or disallowed.

TOPOLOGICAL INCONSISTENCY

The neighborhoods of two vertices share typically more than two vertices, via simplification, non-manifold edges can be created, which would lead to an algorithm failure later on, if this relies on manifold connectivity. But it is easy to detect and can be disallowed.



GENERAL GEOMETRIC REPLACEMENT

The idea is to replace a subset of adjacent triangles by a simplified set with the same boundary, this is also called **multi-triangulation** and can in general encode edge collapses, vertex removals and edge flips.

In general, pure geometric simplification is not enough.

Comparison

- Edge collapse and triangle collapse
  - Simplest to implement
  - Support geometric morphing across levels of detail
  - Support non-manifold geometry
- Full-edge vs. half-edge collapses
  - Full edge represents better simplifications
  - Half-edge is more efficient in incremental encoding
- Cell collapse
  - Simple, robust
  - Varies with rotation/translation of grid
- Vertex removal vs edge collapse
  - Hole retriangulation is not as simple as edge collapse
  - Smaller number of triangles affected in vertex removal

SIMPLIFICATION FRAMEWORKS

Simplification frameworks describe how the overall algorithms work, there exist **local simplification frameworks** (variants include non-optimizing, greedy and lazy) and **global simplification frameworks** (sample and reconstruct, use adaptive subdivision).

LOCAL SIMPLIFICATION FRAMEWORKS - VARIANTS

Nonoptimizing

- Simple
- Cell clustering (independent operations)

```
for each level of the hierarchy
    for each possible independent operation
        perform operation
```

Greedy

- Perform operations in locally optimized order
- Cost function
- Keeps error low (by starting with “small changes”)

```
compute costs for each possible operation
insert them into queue
while the queue is not empty
    extract head of queue (i.e., element with smallest error)
    perform operation
    for each neighbor
        re-compute cost
        notify queue of changed value
```

Lazy

- Fewer cost evaluations

```
compute costs for each possible operation
insert them into queue
set „dirty“ flags to false
while the queue is not empty
    extract head of queue (i.e., element with smallest error)
    if head is dirty
        re-compute cost
        set „dirty“ flag to false
        re-insert into queue
    else
        perform operation
        for each neighbor
            set „dirty“ flag to true
```

GLOBAL GEOMETRY SIMPLIFICATION

Global geometry simplification is available in two variants.

SAMPLE AND RECONSTRUCT

Here, the surface is scattered randomly with sample points, which can repel each other. Then, the number of sample points is reduced and the surface is reconstructed.

ADAPTIVE SUBDIVISION

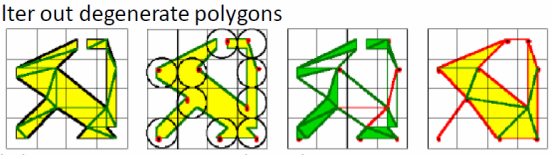
A very simply *base model* is created that represents the model and the faces of this model are selectively subdivided until the fidelity criterion is met. This can be used in **multiresolution modeling**.

VERTEX CLUSTERING

The **resolution of the grid** determines the degree of simplification, this is very fast and robust (topology-insensitive), but it is difficult to specify the degree of simplification, it only has low fidelity and the underlying grid creates a sensitivity to the model orientation.

Example: Vertex Clustering

- Rossignac and Borrel, 1992
- Operator: cell collapse
- Apply a uniform 3D grid to the object
- Collapse all vertices in each grid cell to single most important vertex, defined by
  - Curvature (1 / maximum edge angle)
  - Size of polygons (edge length)
- Filter out degenerate polygons



SIMPLIFICATION OBJECTIVES

There are generally two simplification objectives, a **fidelity-based simplification** (user specifies quality, leads to a constant approximation error) and a **budget-based simplification** (user specifies desired primitive count, leads to constant approximation time), ideally both should be used.

ERROR MEASURES

There are various error measures that can used like **vertex-vertex distance, vertex-plane distance, point-surface distance, surface-surface distance**.

Squared Distance at a Vertex

$$\Delta(v) = \sum_{p \in planes(v)} (p^T v)^2$$

$$= \sum_{p \in planes(v)} (v^T p)(p^T v)$$

$$= \sum_{p \in planes(v)} v^T (pp^T) v$$

$$= v^T \left(\sum_{p \in planes(v)} pp^T \right) v$$

QUADRIC ERROR METRIC

To calculate a **quadratic error metric**, the vertex-plane distance is looked at and the goal is to minimize the distance to all planes at a certain vertex.

Quadric Derivation (cont'd)

pp^T is simply the plane equation squared:

$$pp^T = \begin{bmatrix} A^2 & AB & AC & AD \\ AB & B^2 & BC & BD \\ AC & BC & C^2 & CD \\ AD & BD & CD & D^2 \end{bmatrix}$$

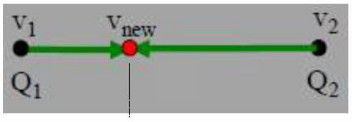
The pp^T sum at a vertex v is a matrix, Q:

$$\Delta(v) = v^T (Q) v$$

Using Quadrics

Construct a quadric Q for every vertex

The *edge quadric*:



Q' =Q1 +Q2

Sort based on edge cost

- Suppose we contract to v_{new}:
- Edge cost = v_{new}^T **Q'** v_{new}

v_{new}'s new quadric is simply **Q'**

OPTIMAL VERTEX PLACEMENT

Each vertex has a quadratic error metric Q associated with it, this error is zero for original vertices and nonzero for vertices created by merge operations. By **minimizing Q**, the optimal coordinates for placing a new vertex can be calculated.

BOUNDARY PRESERVATION

To preserve important boundaries, edges are labeled as *normal* or as *discontinuity*. For each face with discontinuity, a plane is formed perpendicularly intersecting the discontinuous edge. Planes then are converted into quadrics and can be weighted more heavily with respect to the error value.

Quadric Error metric is fast and delivers good fidelity even with drastic reduction, it is able to merge objects (aggregation) and handles non-manifold surfaces.

But it can introduce non-manifold surfaces, the algorithm parameters vary with the model density and are therefore hard to choose. Furthermore, it needs extension to handle color (7x7 matrices).

IMAGE-DRIVEN SIMPLIFICATION

The error is measured by rendering the different images and comparing the result, attributes and shading error is captures as well as the texture content.

APPEARANCE-PRESERVING SIMPLIFICATION

The reduction is drastic with this variant, but the lost geometry is simulated using bump maps.

LOD REPRESENTATION FRAMEWORKS

DISCRETE LOD

This is the traditional approach, it decouples simplification and rendering, the LOD creation itself does not need to address the real-time rendering constraints, the run-time rendering engine simply picks the LODs. It also fits modern graphics hardware very well as it is easy to compile into triangle strips and cache-aware vertex arrays, that render much faster than immediate-mode triangles on today's hardware.

But sometimes discrete LOD is not really suitable for drastic simplification and there are also better approaches concerning fidelity in theory.

CONTINUOUS LOD

This is a departure from the traditional approach, where individual levels of detail are created in a preprocess, here a **data structure is created** from which the desired level of detail can be **extracted at run time**. By allowing better granularity also better fidelity follows, this is possible by specifying the LOD exactly and not choosing from a few pre-created options. Thus, objects use no more polygons than necessary, which frees up polygons for other objects, this results in a better resource utilization, leading to a better overall fidelity.

Also, as there is no switching between LOD that would introduce a *visual popping* effect, here the detail can be adjusted gradually and incrementally, therefore reducing the visual pops. It can even **geomorph** the fine-grained simplification operations over several frames to eliminate pops (this is done in the vertex shader).

It also supports **progressive transmission (streaming)** and leads to a **view-dependent LOD** as it uses the current view parameters to select the best representation for the current view, single objects may thus span several levels of detail.

Continuous LOD Algorithm

- "Progressive meshes"
- Iteratively apply local simplification operator
  - Until base mesh reached
- Entity = edge or vertex or triangle …

```
Sort all entities (by some metric)
repeat
  Apply local simplification operator:
    remove entity
    Fix-up topology
until (no entities left)
```

With this view-dependency, nearby portions of objects can be shown at a higher resolution than distant portions and also the silhouette can be shown at a higher resolution than the interior regions. With this, even better granularity can be provided and it enables drastic simplification of very large objects.

TERRAIN LOD

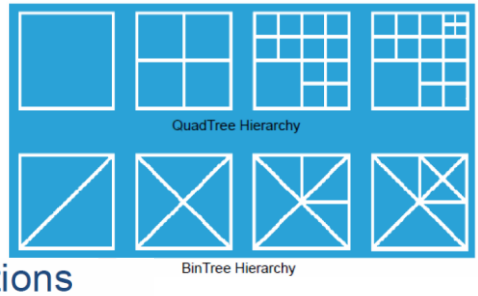
Terrain LOD has been around for a long time, here the geometry is more constrained and requires specialized solutions, as it is simultaneously very near and very far, and this requires a **progressive/view-dependent LOD**. It may pose problems with the dynamic modification if the terrain data and fast rotations of the camera.

REGULAR GRIDS

Regular grids provide a uniform array of height values that are simple to store, manipulate and easy to interpolate to find elevations. It is used most by implementers and allows for easy view culling and collision detection and requires less memory.

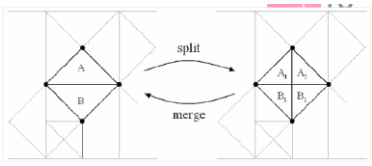
TINS

TINs (Triangulated Irregular Networks) need fewer polygons to attain the required accuracy and do higher sampling in bumpier regions and coarser sampling on flat ones, it is possible to model maxima, minima, ridges, valleys …



LOD HIERARCHY STRUCTURES

Bintrees



Terminology
- Binary triangle tree (bintree, bintritree, BTT)
- Right triangular irregular networks (RTIN)
- Longest edge bisection

Easier to avoid cracks and T-junctions

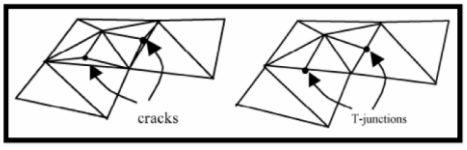
Neighbor is never more than 1 level away

Very popular "ROAM" algorithm

Cracks and T-Junctions

Avoid cracks:
- Force cracks into T-junctions / remove floating vertex
- Fill cracks with extra triangles

Avoid T-junctions:
- Continue to simplify …



We hope it gets as big as virtual reality. And we hope virtual reality gets as big as many people on the planet. Over time. This generation will take time to evolve – we want to be realistic about it – but we definitely see virtual reality as one of the simplest, most easy-to-use user experiences. If you've put on glasses before, you should be able to put on computer glasses in the future, and just be like 'oh, this is me, I know how to do this.' This isn't learning a new interface or a new controller. Ultimately this is just virtual vision."

Brendan Iribe | CEO at Oculus Rift | Talking about the Rift

HDR-RENDERING

Standard color with 24 bit enables 16.7 million colors, more than the human eye can discriminate, but the remaining 8 bit in intensity deliver only 256 different intensities, while humans can perceive 3 orders of magnitude more contrast than a monitor can produce. The dynamic range is given as the ratio of the highest to the lowest luminance.

With **High Dynamic Range Rendering (HDR-Rendering)**, intensities are represented as floating point to avoid clamping and round-off errors. In a final step, **tone mapping** to 24 bit occurs which tries to preserve the maximum amount of information for displaying on a LDR device, this mapping is possible perceptually driven.

With HDRR, the bright things are even brighter, dark things are even darker and details can be found in both.

TONEMAPPING

Because of the fact, that output devices only have a limited dynamic range, there needs to happen a mapping from HDR to LDR (floating point to fixed point), this mapping should be optimized for image details, contrast and quality in general. The parameters necessary for such a compression are different for every image, this is a so-called **dynamic compression**.

OPERATORS

There are two types of operators:

- Global: perform same operation for each pixel
  - Pro: + Quick
    + Preserve global contrast
  - Con: - May over-compress contrast
- Local: work on local neighbourhood of pixel
  - Pro: + Extreme values do not degrade whole image
    + Good local contrast
  - Con: - Slow
    - Can produce halos

GLOBAL

The given compression curve needs to bring everything in a certain range and leave dark areas alone, this implies the following mathematical properties

- Asymptote at 1.0f (=255)
- Derivative of 1 at 0

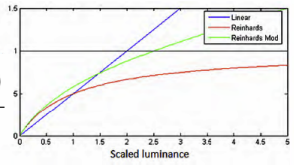
Reinhard Operator

- Global operator

$$L_{scaled} = \frac{a \cdot L_w}{\bar{L}_w}$$

a … Key
$\bar{L}_w$ … Average luminance
L_w … Pixel luminace

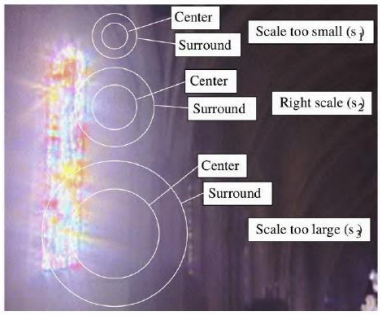
- Original $Color = \frac{L_{scaled}}{1 + L_{scaled}}$
- Modified $Color = \frac{L_{scaled} \cdot \left(1 + \frac{L_{scaled}}{L_{white}^2}\right)}{1 + L_{scaled}}$



- Key *a is* set by user or some predefined curve $a(l_a)$ dependent on average luminance l_a
- Calculations need to be done in linear color space
  - Numerically challenging

Local Operator (Robertson et al.)

- Gaussian average over local neighbourhood
- Different Scaling for each pixel



OFFLINE VS REAL-TIME TONEMAPPING

Offline tone mapping requires complex perceptual methods that can simulate the photoreceptor cells of the human eye, **real-time tone mapping** is much simpler und uses usually a global operator, it should at least consider accommodation effects.

Realtime: Log Compression

E.g., CryEngine 2

Logarithm L_d of HDR $L_d = \frac{\log_x(L_w + 1)}{\log_x(L_{max} + 1)}$

Pros:

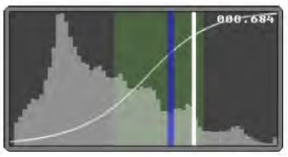
- Fast
- Emulates human perception
- Change of log base can simulate adaptation

Cons:

- Exaggerates extremely dark/bright areas

Realtime: S-Curve Compression

- E.g., Unreal Engine 4
- Compression based on brightness histogram of image
- Pros:
  - Reactive tone mapping
  - Blend to simulate accommodation
- Cons:
  - Creating histogram is costly



Radiance Mapping Algorithm

Combine multiple images with different exposure

Camera has non-linear response function: $Z=f(E \cdot t)$

- Light intensity E, exposure time t
- We want to recover f^{-1}

Given a set of images $Z_{ij} = f(E_i t_j)$ with known t_j

- $g(Z_{ij}) = \ln f^{-1}(Z_{ij}) = \ln E_i + \ln t_j$

Minimize: $\Sigma_{ij} (g(Z_{ij}) - \ln E_i - \ln t_j)^2 + \lambda \cdot \Sigma g''(z)^2$

- Regularization (smoothness term) using g''

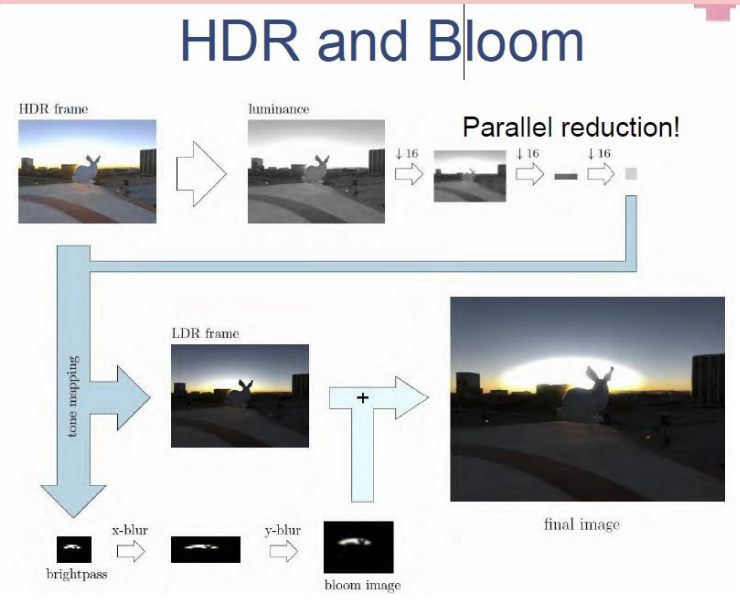
Final step: combine images

- $\ln E_i = \Sigma_j w(Z_{ij})(g(Z_{ij}) - \ln t_j) / \Sigma_j w(Z_{ij})$
- w ... higher weight for exposure close to middle of f

LIGHT BLOOM

The idea here is, that light spills over from bright to dark areas, this is **not an HDR technique** but is suggestive of *lots of light*, when used together with HDR, and the color of very bright light is retained as it is not clamped against white.

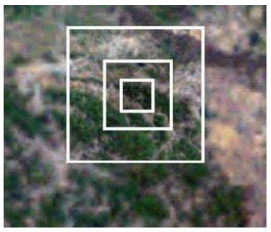
But this effect should not be overused.



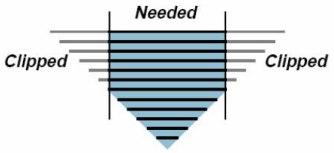
VIRTUAL TEXTURES

The problem is always, that lots of details require lots of memory, high details are needed for close objects and low details for far away objects.

Clipmap

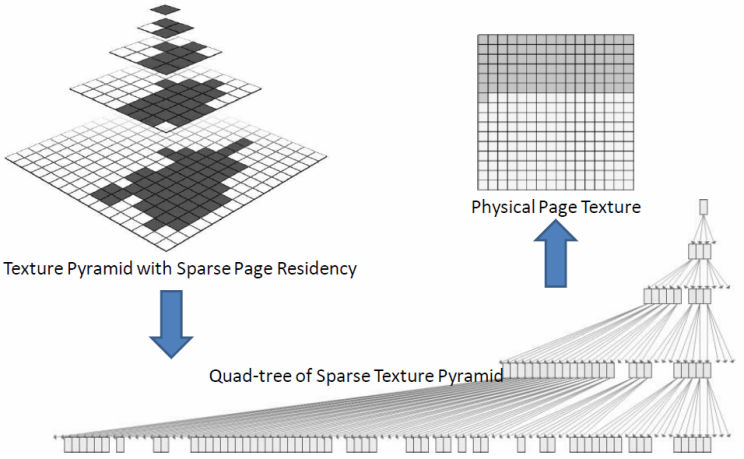


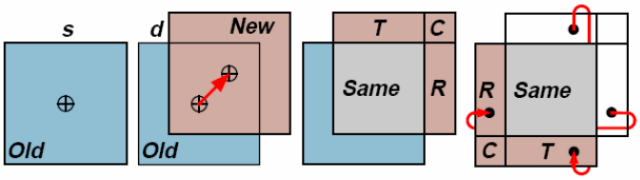
What part of texture map is needed?



Also not all parts of a texture map are needed, each mipmap level only saves small parts of the texture that surrounds the views, and this is called a **clip map**.

Tiled Virtual Textures





WRAP-AROUND UPDATES

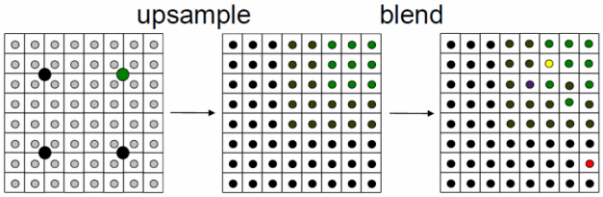
A texture level is updated by **invalidating** the **old region**, **updating** with the new region, if required also invalidate/validate entire levels.

Texture LOD Transition

High latency between first need and availability

Visible snap happens when texture LOD changes

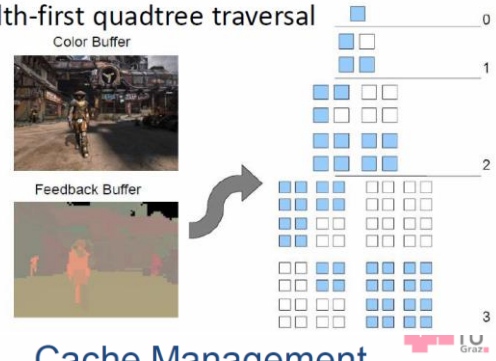
Use trilinear filtering (or similar) to blend in detail



Feedback Buffer

Render texture request to low-res buffer

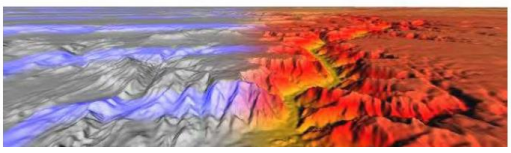
Breadth-first quadtree traversal



Geometry Clipmaps

Height map lookup from textures

Tessellation can be done in CPU vs direct texture lookup in vertex shaders



Cache Management

No blocking allowed

Cache handles hits, schedules misses to load in background

Physical pages organized as quad-tree per virtual texture

Linked lists for
- Free
- LRU
- locked pages