Manuel Pagliuca

A.Y. 2020/2021
Master's Degree
Computer Science

Real Time Graphics Programming

Sommario

RENDERING PIPELINE **6**
- IMAGE-ORIENTED 6
- OBJECT-ORIENTED 6

REAL-TIME RENDERING PIPELINE **8**
- APPLICATION STAGE 9
- GEOMETRY PROCESSING STAGE 9
  - *1. Vertex Shading* *10*
  - *2. Projection* *10*
  - *3. Clipping* *11*
  - *4. Screen Mapping* *11*
- RASTERIZATION STAGE 11
- FRAGMENT PROCESSING 12
  - *Z-buffer algorithm* *13*

PIPELINE PERFORMANCE **15**

GPU PIPELINES **17**
- VERTEX SHADER 18
- TESSELLATION SHADER 18
- GEOMETRY SHADER 19
- FRAGMENT SHADER (OR PIXEL SHADER) 19
- DEFERRED RENDERING 20

SHADER MODEL **22**
- UNIFIED SHADER MODEL 22
- UNIFIED ARCHITECTURE 22
- GPGPU 22

POLYGON MESHES **23**
- EULER'S POLYHEDRON FORMULA 24
- MESH PRODUCTION TECHNIQUES 25
  - *High poly vs Low poly* *25*
- MAIN 3D FILE FORMATS 26
- DATA STRUCTURE FOR POLYGON MESH 28
  - *Indexed mesh* *28*
  - *Half-edge* *29*
- MESH ELABORATION 30
  - *Tessellation* *31*
  - *Consolidation* *31*
  - *Simplification* *32*
- TERRAIN RENDERING (VIEW-DEPENDANT) 35

TRANSFORMATIONS AND PROJECTIONS **36**
- SCALE TRANSFORMATION 37
- ROTATION TRANSFORMATION 37
- TRANSLATION 38

HOMOGENEUS COORDINATES ... 39
INTERPRETATION OF TRANSFORMATIONS ... 43
AFFINE TRANSFORMATIONS IN THE 3D ... 43
EULER ANGLES ... 44
AXIS/ANGLE ROTATION ... 45
QUATERNIONS ... 45
PERSPECTIVE ... 47
CANONICAL CONFIGURATION FOR PARALLEL PROJECTION ... 49
CANONICAL CONFIGURATION FOR PERSPECTIVE PROJECTION ... 50
VIEW TRANSFORMATION ... 52
LOOKAT ... **ERRORE. IL SEGNALIBRO NON È DEFINITO.**
VIEW-FRUSTUM ... 54
THE CANONICAL VIEW VOLUME ... 55
WINDOW TO VIEWPORT TRANSFORMATION ... 57

SPATIAL DATA STRUCTURES ... 60

BOUNDING VOLUME HIERARCHIES ... 60
UNIFORM GRID ... 63
OCTREES ... 64
BINARY SPACE PARTITIONING TREES ... 65
SCENE GRAPH ... 67

API FOR GRAPHICS PROGRAMMING ... 70

DIRECT3D ... 71
OPENGL ... 71
DIRECT3D VS OPENGL ... 71
OPENGL EVOLUTION ... 71
VULKAN ... 73
VULKAN VS OPENGL ... 74
METAL AND FRAGMENTATION ISSUE ... 74
LIMITATIONS OF USING APIS FOR GRAPHICS PROGRAMMING ... 74

CULLING AND CLIPPING ... 76

BACKFACE CULLING ... 76
HIERARCHICAL VIEWFRUSTUM CULLING ... 77
PORTAL CULLING ... 78
OCCLUSION CULLING ... 79
CLIPPING ... 81

RASTERIZATION ... 84

ANTIALIASING ... 87
FSAA ... 88
MSAA ... 88
MFAA ... 88
MLAA ... 89
SMAA ... 89
FXAA ... 89

INTERSECTION TESTS PICKING ... 90

RAY ... 90
SPHERE ... 90
Implicit notation ... 90

Explicit notation ... *90*
INTERSECTION TESTS, RULES OF THUMB ... 91
RAY/SPHERE INTERSECTION : MATHEMATICAL SOLUTION ... 91
RAY/SPHERE INTERSECTION : GEOMETRICAL SOLUTION ... 92
RAY-BOX INTERSECTION (OBB AND AABB) ... 93
RAY-POLYGON INTERSECTION ... 94
Crossing test ... *94*
PLANE-BOX INTERSECTION (OBB AND AABB) ... 95
PLANE-BOX INTERSECTION AABB ALTERNATIVE ... 95
PLANE-SPHERE INTERSECTION ... 95
SPHERE-SPHERE ITNERSECTION ... 96
AABB-AABB INTERSECTION ... 97
SPHERE-AABB INTERSECTION ... 97
SEPARATING AXIS THEOREM (SAT) ... 97
TRIANGLE-TRIANGLE INTERSECTION ... 98
VIEW FRUSTUM INTERSECTION ... 98
VIEWFRUSTUM TESTING ... 99
DYNAMIC INTERSECTION TESTS ... 99
Dynamic intersection tests : Sphere-Plane ... *99*
Dynamic intersection tests : Sphere-Sphere ... *100*
PICKING ... 101
Other techniques 1 ... *102*
Other techniques 2 ... *102*
shader-based techniques ... *102*

COLLISION DETECTION (COLLISION HANDLING) ... **103**

COLLISION DETECTION ... 103
Simple cases ... *103*
COLLISION DETECTION FOR COMPLEX SCENES ... 105
Broad phase ... *105*
Mid-phase ... *107*
Narrow-phase (optional) ... *108*
Collision Response ... *110*

PHYSICS SIMULATION ... **110**

ACCURACY ... 111
FORCES IN PHYSIC SIMULATION ... 113
Gravity ... *113*
Friction ... *113*
Linear momentum ... *113*
Impulse ... *113*
Angular momentum ... *113*
Torque ... *114*
CONSTRAINTS ... 114
EXAMPLE-1 COLLISION OF 2 OBJECTS ... 115
EXAMPLE-2 RAGDOLL PHYSICS ... 116
HOW MUCH REALISM IS NEEDED? ... 117
EXAMPLE-3 SPRING-MASS-DAMPER SYSTEMS ... 117
EXAMPLE-4 VEHICLE ... 119

LIGHT SOURCES IN COMPUTER GRAPHICS ... **121**

DIRECTIONAL ... 121

POINT LIGHT 121
SPOT LIGHT ..... ERRORE. IL SEGNALIBRO NON È DEFINITO.
AREA LIGHT 122
INVERSE-SQUARE LAW 123
MATERIALS 123
BRDF, BTDF, BSDF 124
BSSRDF 125
PHYSICALLY BASED BRDF 125
BRDF 126
ANISOTROPIC BRDF 127
WHERE DO I FOUND THE BRDF FOR MATERIALS ? 127
RENDERING EQUATION 128
ILLUMINATION MODELS 129
BRDF SIMPLIFICATION 130
EMPIRICAL BRDF 130
SIMPLIFIED MODELS OF LIGHT REFLECTION 130
Diffusive 130
Specular 131
Glossy 131
Lambert's cosine law 132
Phon reflection model 1975 133
The whole phong model 135
BLINN-PHONG MODEL 1977 136
PHYSICALLY BASED BRDF MODELS - GGX 137
MICROFACETS DISTRIBUTION (D) 140
Geometry attenuation (G) 141
EMPIRIC BRDF - WARD ANISOTROPIC MODEL 142
OTHER MODELS 142

TEXTURE MAPPING & PROCEDURAL TEXTURES 143

Texturing approach 143
SIMPLIFIED TEXTURING PIPELINE 145
Projector function 145
Projector function : mesh unwrap 145
Projector function : intermediate surface 146
Corresponder function 147
Texture values 148
Bump mapping 151
Normal mapping 153
Parallax mapping 154
Displacement mapping 155
PROCEDURAL TEXTURES 155
Issues of procedural texturing 157
Combination of functions 161
Antialiasing: analysis of fragment neighbourhood 162
Noise 163
Perline Noise 163
*Turbolence ..... **Errore. Il segnalibro non è definito.***
GLSL and noise 169
Layered shader 170

GLOBAL ILLUMINATION TECHNIQUES 170

HEMISPHERE LIGHTING 170
AMBIENT OCCLUSION 172
SSAO (SCREEN SPACE AMBIENT OCCLUSION) 173
IMAGE-BASED TECHNIQUES 174
Environment mapping 174
SHADOW MAP 177
PERCENTAGE CLOSER FILTERING (PCF) 178
LIGHT MAPS 178
SHADOW (DEFINITION) 179
PROJECTION-BASED SHADOWS 180
SHADOW VOLUME 180
IMAGE BASED LIGHTING (IBL) 181
CURRENT STATE OF REAL-TIME GLOBAL ILLUMINATION 183
SSR, SCREEN BASED REFLECTIONS 183
STOCHASTIC SCREEN BASED REFLECTIONS 185
IRRADIANCE VOLUMES 185
VIRTUAL POINT LIGHTS (VPL) 186
NVIDIA VXGI (VOXEL GLOBAL ILLUMINATION) 186
REAL-TIME RAYTRACING 188
Shaders in DXR 189
FINAL CONSIDERATIONS 191

Rendering Pipeline

The main function of the pipeline is to generate, or render, a two-dimensional image, given a virtual camera, three-dimensional objects, light sources, and more. The final result is a render (an image), It's like shooting a photo of a virtual world.

There are mainly two different approaches to the rendering pipeline

1. **Image-oriented**
2. **Object-oriented approach**

Image-oriented

We could also call it "offline rendering", the image is generated calculating the color for each pixel, this implies that the whole scene has to be loaded in memory. The process consists in send lot of rays from the camera to the scene and then calculate the various intersections with objects.

This approach gave a **photorealistic** look, for this it is used in **photorealistic rendering** technologies like ray tracing, photon mapping, surface scattering…

This approach is even used in Animation movies, each render of the movie could take hours for being drawn, cause of the intensive computations involved.

This kind of approach is done primarily for this non interactive media, that can push the hardware a lot since there isn't expected a real-time feedback from the user.

The rendering pipeline :

- Preprocessing
  - Loading of models
  - Creation of scene graph
  - Acceleration of data structures generation
- Rendering happens with raytracing technique.
- Post processing effects
  - AA
  - Gamma correction
  - …

Object-oriented

In Real-Time graphics we don't trace rays often like in the offline graphics paradigm, we use for some particular features. In the Object-oriented approach, each object is rendered. We can't render like we did for the image-oriented approach, because we have a different goal, the goal is the smoothness of interactivity between all this images. Rendering must be recalculated several times per second, at least 30fps. This approach is used for interactive application (the state changes).

The goal is everything must be **smooth** (**RTGP**), there is lot of tricks in this field to make the application run fast, because it is a priority! This is done in recent games, that looks really good but still using the interactive approach (no offline approach like Pixar movie).

Object Oriented Approach , Different phases:

- Preprocessing
  - Model loading
  - Scene graph creation
  - Collision detection

- IO management
- …
- **Culling**

All the data collected in the preprocessing phase is submitted to the Graphics HW.

- Rendering
  - For each vertex of each triangle
    - Apply transformations (part of the **T&L**)
    - Apply local illumination model (part of the **T&L**)
    - Clipping
    - Determine pixel composing the on-screen triangle
    - Assign final color to each pixel (respecting the depth order)

Real Time Graphics is not only videogame, it is the most known media but anytime I have a graphical application which is interactive (Unity, Editor, Maya, Blender, scientific visualization …) so where the graphical result changes interactively in real time when I press a button, that one is based on Real-Time Graphics.

Differences between the two rendering approaches

The **image-oriented** is much more expansive in terms of resources, this brings better results for photorealistic rendering and animation movies. This happens to the fact that the whole scene must be loaded in memory, and then use the ray tracing rendering technique for calculate the color of each pixels.

- Direct and indirect light
- Intersections
- Real physics equations
- …

The Object-oriented approach instead, discards all the geometry not involved in the generation of the single frame, because we have a different goal, the smoothness of the result since we are talking about an interactive application.

- Each triangle is elaborated independently.
- Local illumination.
- Essential's triangles are sent to GPU.
- Discard elements not used in the rendering stage (culling).

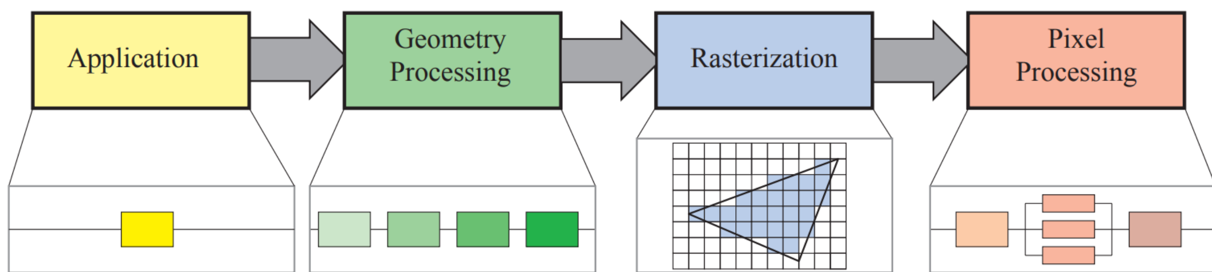
Real-time Rendering Pipeline

In the Real-time Rendering Pipeline, we manage and elaborate each object **separately**, **every** 3D model in the scene is **composed by triangles**, every object has got elaborated independently. We got a large preprocessing phase, but also we do lot of other operations which are related to the interactivity of the application (input events), it is something that must be detect during the rendering.

The pipeline stages execute in parallel, but they are stalled until the slowest stage has finished its task. A coarse division of the real-time rendering pipeline into four main stages—application, geometry processing, rasterization, and pixel processing. Each of these stages is usually a pipeline in itself, which means that it consists of several substages.

We differentiate between the **functional stages** shown here and the **structure** of **their implementation**.

- A **functional stage** has a certain task to **perform** but does not specify the way that task is executed in the pipeline.
- A **given implementation** may combine **two functional stages** into one unit or execute using programmable cores, while it divides another, more time-consuming, functional stage into several hardware units



The **rendering speed** may be expressed in frames per second (**FPS**), that is, the number of images rendered per second. It can also be represented using **Hertz** (**Hz**), which is simply the notation for 1/seconds.

Overview of the pipeline stages :

- **Application stage**, the application stage is driven by the application and is therefore typically implemented in software running on general-purpose CPUs. These CPUs commonly include multiple cores that are capable of processing multiple threads of execution in parallel.
  - Some tasks performed on the CPU : collision detection, global acceleration, animation, physics simulation.
- **Geometry Processing stage**, which deals with transforms, projections, and all other types of geometry handling. This stage computes what is to be drawn, how it should be drawn, and where it should be drawn. The geometry stage is typically performed on a graphics processing unit (GPU) that contains many programmable cores as well as fixed-operation hardware.
- The **Rasterization stage**, typically takes as input three vertices, forming a triangle, and finds all pixels that are considered inside that triangle, then forwards these to the next stage.

- The **Pixel processing** stage, it may also perform per-pixel operations such as blending the newly computed color with a previous color.

The images are stored inside the **"Frame Buffer",** it's a portion of memory on the GPU where I save the final pixels of my render before showing them to screen (32-bit image). Inside this memory is contained the information about the color of every pixel, in nowadays usually are used 32 bits per pixel (24-bit true color pixel + the alpha channel).

To avoid a partial update of the frame content, the *flickering* effect, it's used the **double buffering** (with Vulkan recently we moved to the triple buffering) technique that involves two buffers called back-buffer (which contains the new image) and front-buffer (which contains the current image, the thing you are seeing on the screen).

- Rendering in back buffer the image
- Swap back buffer and front-buffer
- Show the new frame

Back in the past the **frame buffer** was placed in the main **RAM**, now it is on the GPU.

Application Stage

The developer has **full control** over what happens in the application stage, since it usually executes on the **CPU**. Therefore, the developer can entirely determine the implementation and can later modify it in order to improve performance.

At the end of the Application Stage, the **geometry** to be **rendered** is **fed** to the **Geometry Processing Stage**. These are the rendering primitives, i.e., points, lines, and triangles, that might eventually end up on the screen (or whatever output device is being used). This is the most important task of the application stage.

All this said, some application work can be performed by the GPU, using a separate mode called a ***compute shader***. This mode treats the GPU as a highly parallel general processor, ignoring its special functionality meant specifically for rendering graphics.

A consequence of the software-based implementation of this stage is that it is not divided into substages, However, to increase performance, this stage is often executed in parallel on several processor cores.

One process commonly implemented in this stage is collision detection. The application stage is also the place to take care of input from other sources, such as the keyboard, the mouse, or a head-mounted display. Depending on this input, several different kinds of actions may be taken.

Geometry Processing Stage

The geometry processing stage on the GPU is responsible for most of the per-triangle and per-vertex operations.

1. Vertex Shading

There are two main tasks of vertex shading, namely:

1. Compute the position for a vertex.
2. Evaluate whatever the programmer may like to have as vertex output data, such as a normal and texture coordinates.

In the past much of the shade of an object was computed by applying lights to each vertex's location and normal and storing only the resulting color at the vertex. These colors were then interpolated across the triangle. For this reason, this programmable vertex processing unit was named the "vertex shader".

The vertex shader is now a more general unit dedicated to setting up the data associated with each vertex.

The vertex position is computed starting from a set of coordinates where the model resides in its own **model space**, which simply means that it has not been transformed at all. Each model can be associated with a **model transform** so that it can be positioned and oriented. It is possible to have several model transforms (matrix operations) associated with a single model. *It is the vertices and the normals of the model that are transformed by the model transform.* The coordinates of an object are called **model coordinates**, and after the model transform has been applied to these coordinates, the model is said to be located in **world coordinates** or in **world space**.

As mentioned previously, only the models that the camera (or observer) sees are rendered. The camera has a location in world space and a direction, which are used to place and aim the camera. To facilitate projection and clipping, the camera and all the models are transformed with the **view transform**.

The purpose of the **view transform** is to place the camera at the origin and aim it, to make it look in the direction of the negative z-axis, with the y-axis pointing upward and the x-axis pointing to the right.

(The actual position and direction after the view transform has been applied are dependent on the underlying application programming interface (API).) The space thus delineated is called camera space, or more commonly, view space or eye space.

Next, we describe the second type of output from vertex shading. This operation of determining the effect of a light on a material is known as **shading**. It involves computing a **shading equation** at various points on the object. Typically, some of these computations are performed during geometry processing on a model's vertices, and **others may be performed during per-pixel processing**.

Vertex shading results (which can be colors, vectors, texture coordinates, along with any other kind of shading data) are then sent to the **rasterization** and **pixel processing stages** to be interpolated and used to compute the shading of the surface.

2. Projection

The aim of this stage is to convert 3D coordinates to 2D coordinates.

As part of vertex shading, rendering systems perform projection and then clipping, which transforms the view volume into a **unit cube** with its extreme points at $(-1, -1, -1)$ and $(1, 1, 1)$. The unit cube is called the **canonical view volume**. Projection is done first, and on the GPU it is done by the vertex shader.

There are two commonly used projection methods, namely orthographic and perspective projection. Note that projection is expressed as a **matrix** and so it may sometimes be concatenated with the rest of the geometry transform.

- The main characteristic of **orthographic projection** is that parallel lines remain parallel after the transform. This transformation is a combination of a translation and a scaling
- The **perspective projection** is a bit more complex. In this type of projection, the farther away an object lies from the camera, the smaller it appears after projection. In addition, parallel lines may converge at the horizon. The perspective transform thus mimics the way we perceive objects' size. Geometrically, the view volume, called a **frustum**, is a truncated pyramid with rectangular base. The frustum is transformed 18 2. The Graphics Rendering Pipeline into the unit cube as well.

Both orthographic and perspective transforms can be constructed with 4 × 4 matrices and after either transform, the models are said to be in **clip coordinates** (this occurs before division by *w*).

During this phase the z-coordinates of the objects is stored inside the *Z-buffer.*

3. Clipping

Only the primitives wholly or partially inside the view volume need to be passed on to the rasterization stage which then draws them on the screen. Primitives entirely outside the view volume are not passed on further, since they are not rendered. The use of a **projection matrix** means that the transformed primitives are clipped against the unit cube. Only the primitives that are partially inside the view volume that require clipping.

4. Screen Mapping

Only the primitives inside the view volume are passed on to the screen mapping stage, and the coordinates are still three-dimensional when entering this stage. The x and y coordinates of each primitive are transformed to form **screen coordinates** by adapting the primitives coordinates to the screen aspect ratio (this is called view-port transform).

Rasterization Stage

Given the transformed and projected vertices with their associated shading data, the goal of the next stage is to find all pixels that are inside the primitive, and then the rasterization colors all these pixels. This is the big difference in respect of image-oriented approach, where the pixel color is chosen since the beginning and not at the end of a pipeline (we didn't talk about pixels since now).

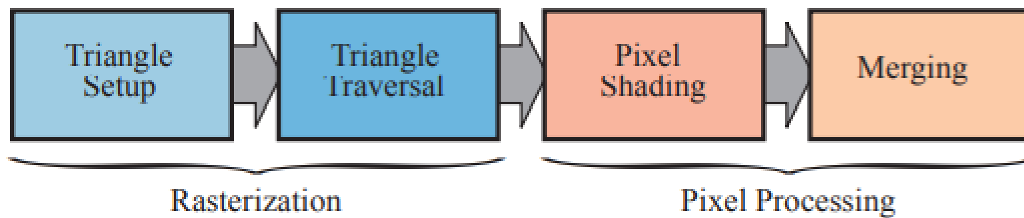
Rasterization, also called *scan conversion,* is thus the conversion from two-dimensional vertices in screen space(each with a z-value (**depth value**) and various shading information associated with each vertex, this is the triangle traversal) into pixels on the screen.

So, we have these vertices in continuous domain, and we have to represent it on a discrete domain (the pixels). This is the rasterization, there is different techniques that differ in base how the GPU's pipeline was settled up.

Attention, the rasterization produces **fragments**, this because the rasterization doesn't create the final pixels. The rasterization takes the primitive vertices from something that could maybe be visible (still not culled or clipped, remember the vertex processes doesn't care about how the object is placed, they just calculate the vertices for that), and calculated the *potential pixels,* and the scan conversion is applied to every triangle of every object.

Rasterization can also be thought of as a synchronization point between geometry processing and pixel processing, since it is here that triangles are formed from three vertices and eventually sent down to pixel processing.

It is split up into two functional substages: **triangle setup** (also called primitive assembly) and **triangle traversal**.



1. **Triangle Setup**
   In this stage the differentials, edge equations, and other data for the triangle are computed, some of this data will be used in the next step the **Triangle traversal.**

2. **Triangle Traversal**
   In this stage of the rasterization is where each pixel that has its center covered by the triangle is checked and a fragment is generated for the part of the pixel that overlaps the triangle. Finding which samples or pixels are inside a triangle is often called triangle traversal. Each triangle fragment's properties are generated using data interpolated among the three triangle vertices, these properties include the fragment's depth, as well as any shading data from the geometry stage.
   - **Bresnam's algorithm** : considering the distance between two points **P** and **Q**, I then consider the distance between the point of the line and the center of the pixels, then I search for the closest pixel to the "line" and I assigned that pixel to the line. In some cases, I can color two pixels (2-pixel centers are perfectly distant from the line). This technique is applied to all edges to determine a primitive like a triangle , and then the intern of the primitive will be filled.

Fragment Processing

At this point, all the fragments are considered, the Pixel Processing (or we should call it fragment processing) is divided into *pixel shading* and *merging*. Pixel processing is the stage where per-pixel or per-sample computations and operations are performed on pixels or samples that are inside a primitive.

1. **Fragment Shading (or pixel shading)**
   Any per-pixel shading computations are performed here, using the interpolated shading data as input. the pixel shading stage is executed by programmable GPU cores. To that end, the programmer supplies a program for the pixel shader which can contain any desired computations. A large variety of techniques can be employed here, like texturing, color interpolation, per-fragment illumination, …

2. **Merging**
   Final operations to determine the final color of pixel, the information for each pixel is stored in the *color buffer.* It is the responsibility of the merging stage to combine the fragment color produced by the pixel shading stage with the color currently stored in the buffer. This stage is also called **ROP** "**raster operations (pipeline**)" or "**render output unit**".

   Unlike the shading stage, the GPU subunit that performs this stage is typically **not fully programmable**. However, it is **highly configurable**, enabling various effects. This stage is also responsible for resolving visibility. This means that when the whole scene has been

rendered, the color buffer should contain the colors of the primitives in the scene that are visible from the point of view of the camera.

For most or even all graphics hardware, this is done with the *Z-buffer* (also called depth buffer) algorithm. A *Z-buffer* is the same size and shape as the *color buffer*, and for each pixel it stores the z-value to the currently closest primitive, this is done in the projection stage.

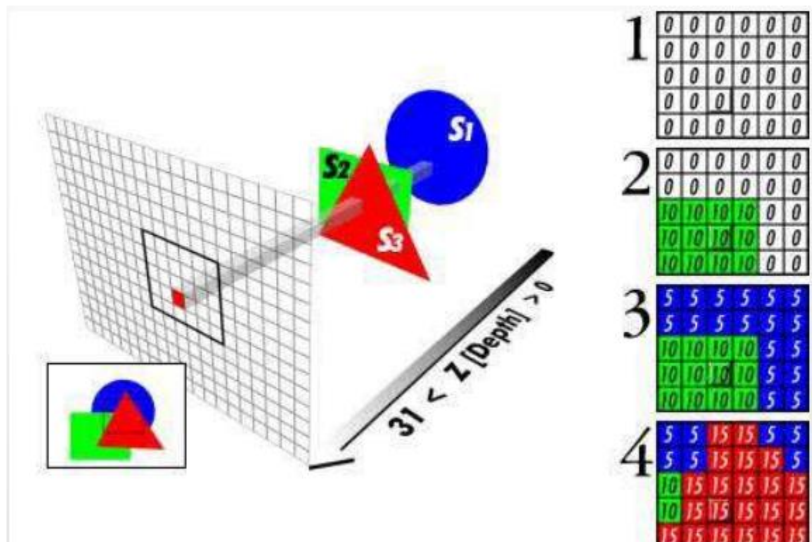
This means that when a primitive is being rendered to a certain pixel, the z-value on that primitive at that pixel is being computed and compared to the contents of the Z-buffer at the same pixel. If the new z-value is smaller than the z-value in the Z-buffer, then the primitive that is being rendered is closer to the camera than the primitive that was previously closest to the camera at that pixel.

Therefore, the z-value and the color of that pixel are updated with the z-value and color from the primitive that is being drawn. If the computed z-value is greater than the z-value in the Z-buffer, then the color buffer and the Z-buffer are left untouched.

Z-buffer algorithm

There is an auxiliary buffer initially set at 0, for all the fragments we independently do the following operations (processing order of the fragments is irrelevant) :

- Check the depth value in the Z-buffer
- If the fragment depth > depth inside the Z-buffer (> o < depends on pipeline implementation), this means that the current fragment is closer than previous fragments. Then we update the Z-buffer and assigning fragment color to final pixel.



- **The Z-buffer Issues with transparency**
  The Z-buffer stores only a single depth at each point on the screen, so it cannot be used for partially transparent primitives. These must be rendered after all opaque primitives, and in back-to-front order, or using a separate order-independent algorithm. Transparency is one of the major weaknesses of the basic Z-buffer (one of the solutions is to do a rough approximation (there is a chapter in the book)).

Other Buffers :

- **Alpha channel**

- **Stencil buffer** is an offscreen buffer used to record the locations of the rendered primitive. The stencil buffer can be a powerful tool for generating some special effects. All these functions at the end of the pipeline are called raster operations (ROP) or blend operations.
- **Accumulation buffer**, used for effects like motion blur, soft shadows, etc.
- The **framebuffer** generally consists of all the buffers on a system.
- To avoid allowing the human viewer to see the primitives as they are being rasterized and sent to the screen, **double buffering** is used. This means that the rendering of a scene takes place off screen, in a back buffer. Once the scene has been rendered in the back buffer, the contents of the back buffer are swapped with the contents of the front buffer that was previously displayed on the screen. The swapping often occurs during vertical retrace, a time when it is safe to do so.

Pipeline performance

As we know we have a sequence of operations, some of them are on the **CPU** (for the *application stage*) and others are on the **GPU**. We may need to establish some measures of performance of the pipeline.

- **The speed of the rendering pipeline** is called "**rendering speed**" or "**throughput**" , this is given by the **slowest** pipeline stage, in other situations this stage is called "**bottleneck**".

- **FPS** means ***frame per second***, it represents the number of images you can render in one second (can change, isn't a fix value).

- **Hertz** is a measure for frequency, and it is expressed as 1/seconds, this is more linked to displays for the output devices since it is an update frequency (and it is a fixed value).

- **Fill rate**, is the number of pixels that a GPU can write to memory in 1 second, the unite measure is Gigapixel/s (at the moment @2021)

Fill rate = (byte per pixel) x (dim framebuffer) x (update frequency)

Anyway, there is not a univocal method to determine the fill rate, it is used by card manufacturer to test the GPU conditions, extremely high fill rate is required for real-time graphics, many of those high results are measured in laboratory conditions.

The role of fill rate was more important back in the past, because it was used as "marketing indicator" of performance by video card manufacturers, currently its role is less considered (to measure the performance of the GPU but isn't the only one to look at).

GPU, **Graphics Processing Unit** is a hardware specifically designed to accelerate graphics rendering, the first commercial GPU was distributed in the **90'** from **Silicon Graphics** was leader in professional fields, mainly used by playing industries, military industries, CAD,… for doing graphics operations.

Then since 96' we had the first version of GPU commercially available for PC, the **3Dfx Voodoo 1**.

Today exist two kinds of GPU for PC, one with a dedicated hardware and dedicated cards, the other is integrated in the motherboards.

Videogames are the most computational expensive application that we can have, they still push the **HW** to the limit because we want to use better algorithms for better rendering.

The GPU has its own processor and memory, it is connected to the PC through the bus.

Different types of bottlenecks

- If it is during the geometric stage ***Transform-limited application***
- If it is during the rasterization stage ***Fill-limited application***
- If it is due to the **CPU** ***CPU-limited application***
- If it is due to the **BUS** (bandwidth of the BUS) ***Bandwidth-limited application***

There are different kind of measures of the GPU's, picking the case for GeForce Titan RTX

- Computational Operations :11.34 TFLOPS
- Bandwidth : 672 GB/s
- 180+mln of cards
- The diffusion of the GPU doubled each 6 months, Moore's law cubed

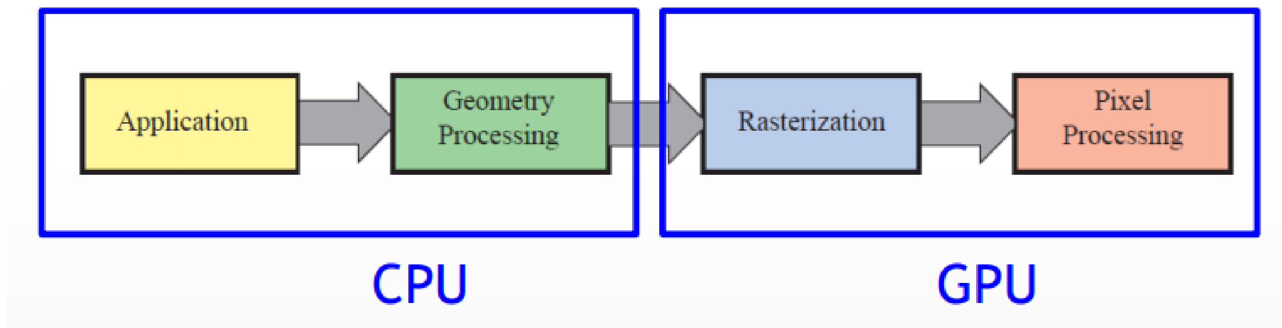
There are also different words respect to the design, people think that a GPU is like a computer inside a computer, but a GPU is designed mainly to do **Graphics Rendering**, so the internal structure of this computational machine is different than the design of a PC, it is made to treat Linear Algebra and Matrix Operations.

- CPU
    - Limited amount of cache
    - Control Unit
    - **Limited amount** of ALU, only 7% of the CPU is dedicated to the ALU.
    - DRAM
- GPU
    - **Insane amount** of ALU's, more than 50% of the GPU is dedicated to the ALU.
    - DRAM

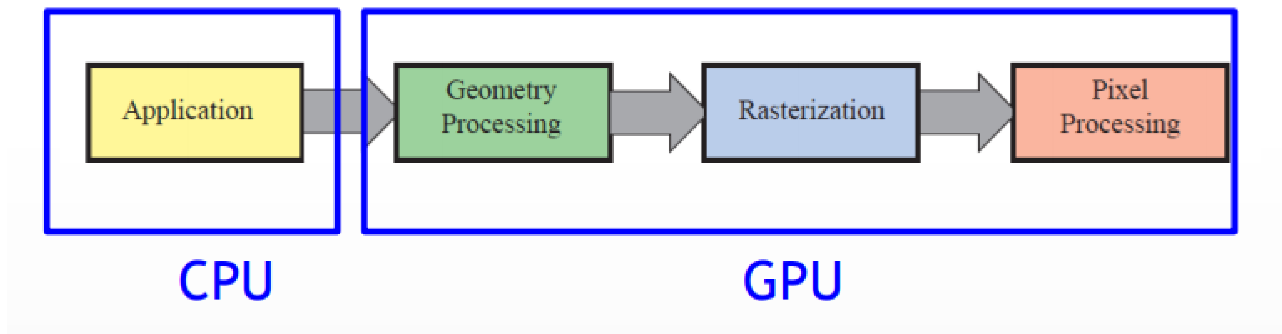
They are really different !

GPU Pipelines

In the **96**' the GPU was more like a **black box**, you couldn't develop shader or edit parameters, even the geometry processing was mainly on the CPU, this is why at that time we rely in back-to-front ordering.

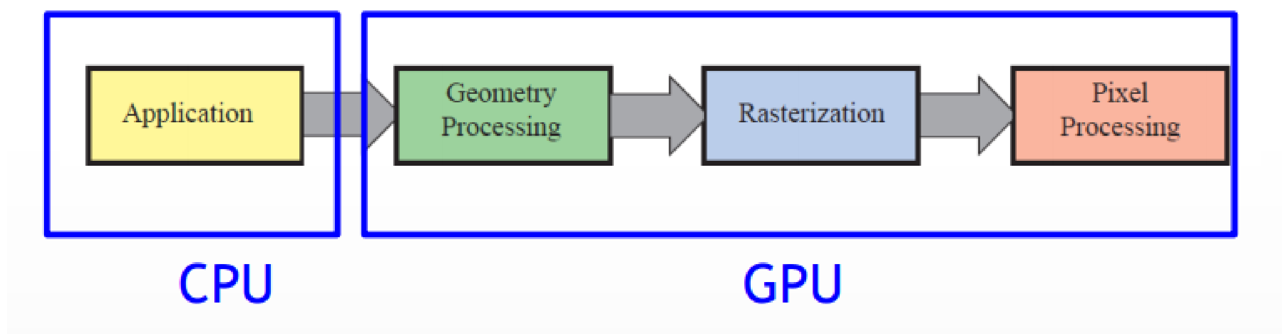


With the going on of researches we arrive to the situation where the geometry processing moved to the GPU, and CPU got totally divided from the GPU stages.



These stages weren't programmable it was called **fixed-function pipeline,** the only possibility was to enable or disable the Z-buffer test, for the rest were no control over the stages.

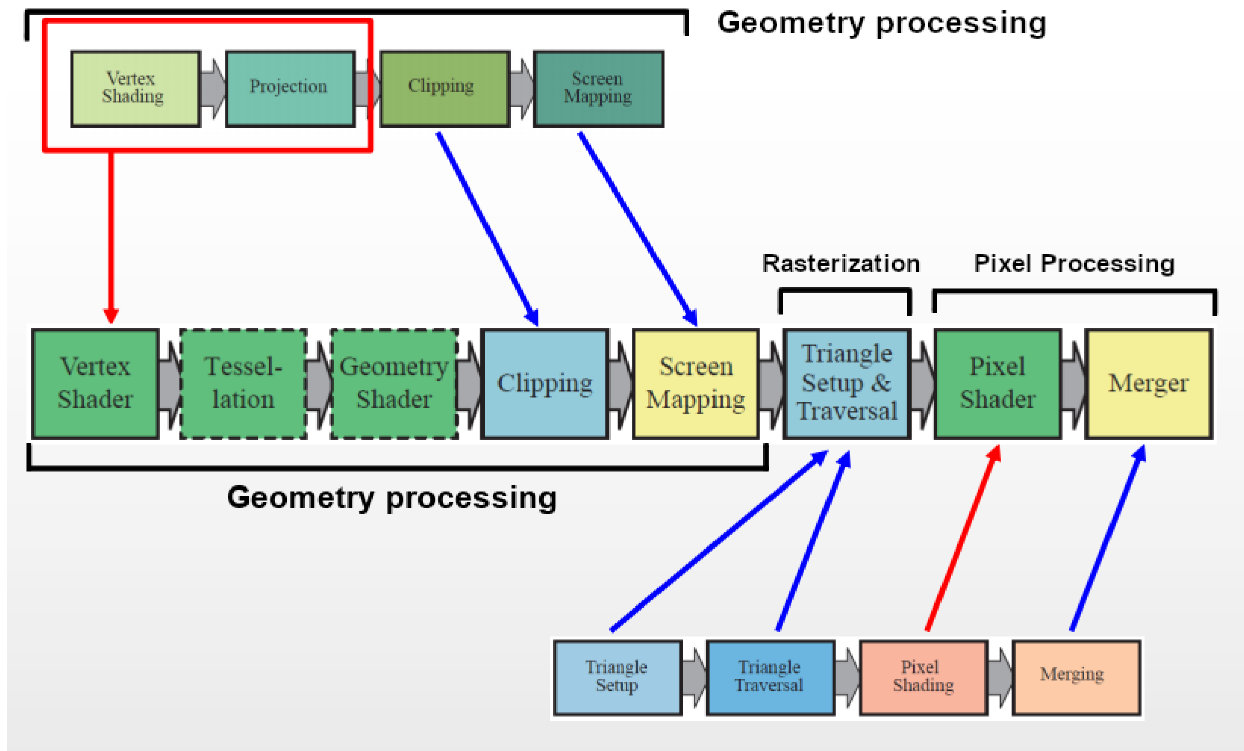
In the last 15 years we developed the **programmable pipeline** (still same distribution of stages between CPU and GPU).



- The crucial stages are programmable by ***shaders***, where we can tell the GPU how we want to treat the vertices by code.
- There is still part of the GPU where they aren't programmable, there are still fixed operations.

The trend Is slow, but the direction is to be going in direction to a **completely programmable pipeline**.

Given a reorganization of the stages we have seen before, we could update the scheme like this



The green squares are the programmable stages by shaders (the dotted are the optional shaders), the blue squares (Clipping and Triangle Setup + Triangle Traversal) are the fixed stages, the yellow squares (Screen Mapping and Merger) are fixed but it's possible to set some parameters.

Vertex Shader

This is the first part of the pipeline where I can develop code. A VS can modifies create ignores values associate to the vertices :

- Color
- Normal
- Texture coordinates
- Position

It can't destroy vertices (it is possible with tessellation shader).

Each vertex is elaborated **independently ,** they are elaborated in parallel by the GPU.

Tessellation Shader

It is an optional stage and is mainly used to render curved surfaces :

- **Beziers surfaces**
- **NURBS**

Before tessellation the curved surfaces were obtained only with offline rendering. This tessellation shader can change dynamically the number of meshes involved in the figure giving a much higher level of detail. It is a computationally expensive shader (less then geometry shader). It reduces the

memory involved because the surfaces are computed dynamically, no bottleneck between the CPU and GPU.

Geometry Shader

Its an optional stage, it is used to elaborate a subset of vertices together, for example :

- Creation of a limited number of copies
- Edges selection for shadow generation techniques
- Extraction of parameters linked to vertices

For example, extract some parameters from the primitives elaborated and …

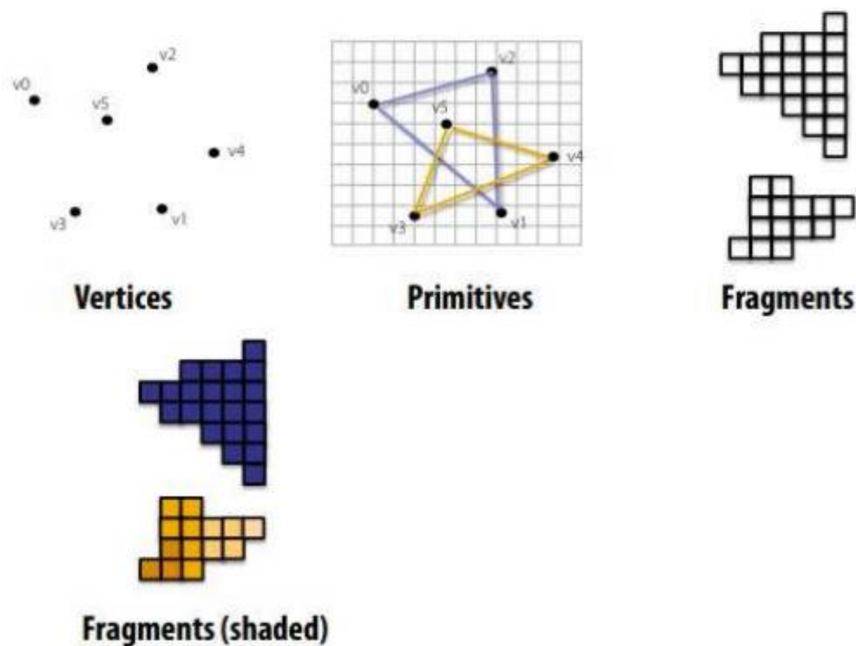
It is a computationally expensive shader.

Fragment Shader (or Pixel Shader)

It is a fully programmable stage over the fragment operations, it gives the color to assign to that fragment. But is also used for :

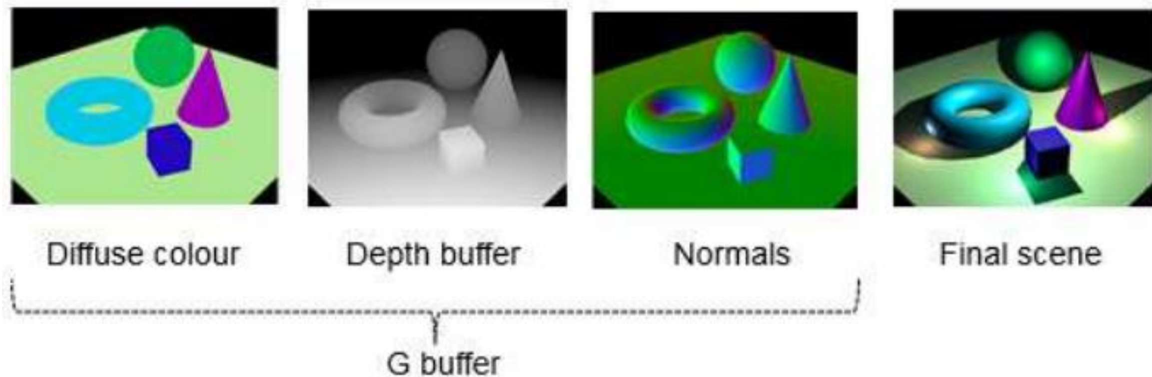
- Image Processing Techniques
- Gradient computations
- Advance texture techniques

Each fragment is elaborated independently, the standard output of the fragment shader (or pixel shader) is the **fragment color**. After the fragment shader the Z-buffer test will occur (using the Z-buffer algorithm).



Deferred Rendering

It is possible to use the Fragment/Pixel Shader for **MRT, Multiple Render Targets**, it means that the pixel shader results are saved in auxiliary buffers. Usually, the result is saved on the **frame buffer** (after the Z-buffer) but is also possible to save the results of the computation not directly on the frame buffer but on auxiliary buffers.



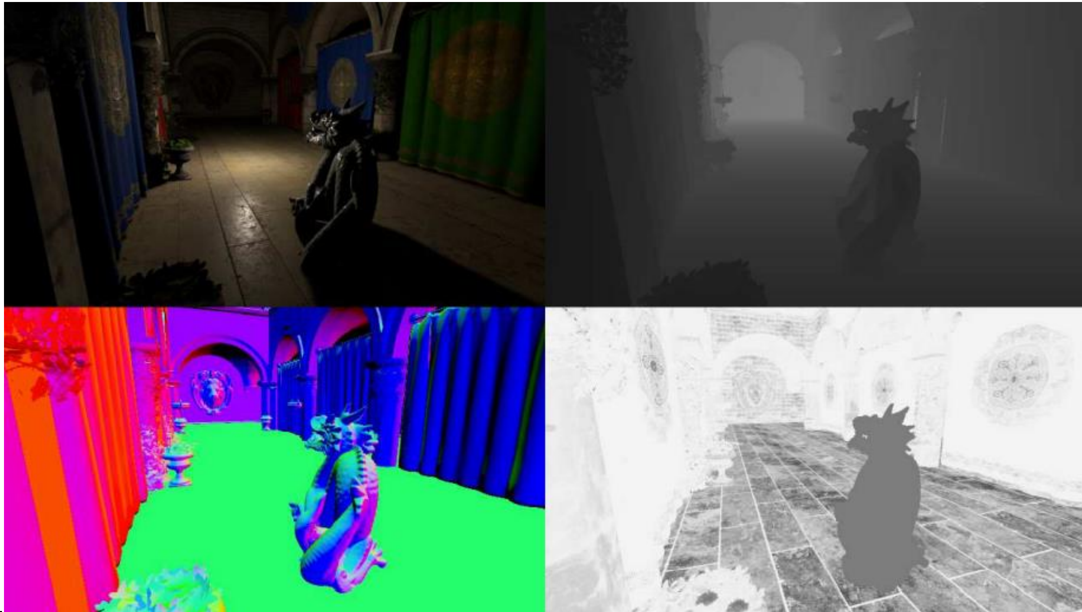
It's like we have an area of temporary memory like the **frame buffer**, and we assign for a certain time the results in this buffer instead of the classic frame buffer.

This is at the basis of a technique called **deferred rendering**, so for do that *we need more than one buffer* and after we do the elaboration of Vertex Processor and Rasterizer, then the Fragment Processor doesn't save the final color of the Pixel, but it computes different components of the color (which usually are combined together for obtain the final color) it saves these components as images inside this buffers.

This process requires two rendering steps :

1. The first pass is called **geometry pass**, we render the scene and retrieve all kinds of geometrical information (vectors, normal, position) from the objects that we store in a collection of textures called the G-buffer.
2. The second pass is called **lighting pass**, we use the textures saved in auxiliary buffers (the G-buffer or MRT), then we iterate pixel by pixel over them applying the lighting (Fragment Shader). Instead of calculating each object from Vertex Shader to Fragment shader, essentially we calculate the Fragment Shader of all the G-buffer objects in the second pass.

The major advantage is that everything inside the MRT (or G-buffer) will be combined in fragment information that will go on screen, this is great because assure us that for each pixel the lighting will be calculated once (depth information stored as texture), this allows us the intensive use of lighting.



This works well because the standard pipeline does depth calculation **after** the Fragment Shader, so after the lighting calculation, this means that in the standard way the lights are calculated for every object in the scene even if the object is behind another.

Forward Rendering vs Deferred Rendering

Forward Rendering is the classical pipeline VS-GS-FS and save the resulting image directly in the frame buffer. This is the standard mode where the illumination is calculated per-vertex, per-fragment for each light and this involves lot of computation. Much of the calculations used per-fragment will be discarded during the **depth testing**.

This is sadly the truth the **standard depth testing** is applied after the fragment shader, only in the most recent architectures there is a specialized hardware to perform an optional **early depth testing**.

Deferred Rendering in this kind of pipeline VS – FS the geometry is processed without the lighting computation there are several render targets to being saved (colors, depth and normal). saves on **MRT** (Multiple render targets, different buffers)

The lighting will be calculated at the end in the *Fragment Shader* only the actually visible fragments. There will be more than one execution of the **pipeline** one for **MRT**, this means that everything has to be optimized (I'm using the GPU for just one frame).

Complexity Forward Rendering : O(*num_fragments * num_lights***)**

Complexity Deferred Rendering : O(*screen_resolution* num_lights***)**

Deferred Rendering issues

- It is needed recent hardware with **MRT.**
- An adequate bandwidth
- Issues with transparency and multiple materials
- No anti-aliasing

After the fragment shader there is the depth test.

Shader Model

Different graphics board of different generations may have some specific techniques or features, so to help subdivide these families they have introduced the term "**Shader Model"**.

The number of Shader Model is linked to the version of the Graphics API, higher the number of the **Shader Model** newer will be the GPU, the Shader Model is a number that is used to classify the characteristics of the video card.

Unified Shader Model

The Unified Shader Model or **Shader Model 4.0** is a Shader Model that use a common instruction set for handling all the three typology of shaders (vertex, geometry and pixel). It's an evolving technology and we are moving to this. It was introduced by DirectX 10
It offers many advantages :

- Simpler shader coding, since the **ISA** is reduced.
- More flexible use of the hardware, any shader unit can be used no difference in base of the type.

In this model, we have a common architecture (unified architecture), in the normal architecture we have transistors for Vertex, Pixel and Geometry shaders, different parts of the GPU are used for the different shaders. This also means that the shader programming languages (like GLSL, HLSL, …) have instructions or features that are usable only in a specific shader. The aim of the Shader Model 4.0 is to use a common architecture with common operations for all the shaders.

Unified Architecture

It's the architecture used for supporting the Shader Model 4.0, the aim of this architecture is to have a unified design where all the **computational units** can execute all the types of shaders.

- This means having a much more flexible architecture, where all the shaders are treated as threads.
- At the moment the Unified Shader Model is more advanced than the Unified Shader Architecture.

The idea of **unified architecture** is to have no more distinctions between processors, so that I can dynamically change their distributions of the processors as needed, resulting in a less sequential approach where all the computational units are the same and in the same place and then distributed in base of the processors needs.

GPGPU

In the last years they exploited the capabilities for the **GPU**, and they found out that this boards are capable of a very fast parallel computation at very low cost in respect of **cluster of machines**.

For do that there is high level API like **OpenCL** and **CUDA**, for allowing the programmers using the computational pipeline of the machine for GPGPU. They don't use the graphic pipeline, they are more a higher level of the rendering API, under the hood the computation is then performed on the graphics hardware. Usually, they use the **fragment processor** the main idea is to use the approach SIMD. They load the data on a matrix and use the fragment processor to elaborate this matrix of data.

Polygon Meshes

A **primitive** in computer graphics is a basic element, such lines, points or polygons, which can be combined to create a more complex graphical image.

A **polygon mesh** is a **lattice** that defines an object in the space, this object is composed by vertices, edges and faces. The faces could miss, these depends by the natures of the objects (wireframe).

Usually, they have some attributes with the vertices :

- **Color**
- **Normals**
- **Texture coordinates**

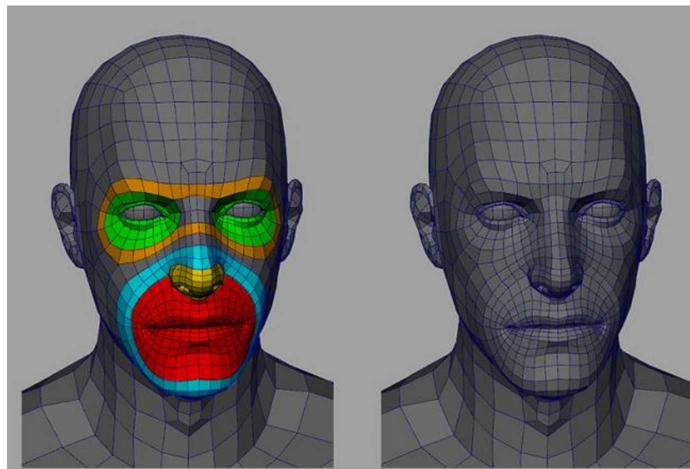
Usually, they are stored in the disk, and they are sent together with the vertices, in other cases no and they are computed at the moment in the application stage.

An important concept is the **connectivity among the primitives**, which is the topology of the mesh (which is a graph). It is really important because our mesh is made by these vertices and the connection between vertices.

We can see the topologies from both sizes:

- **Creation process**, we need to have a good topology this allow the model to be changed processed and modified easily, and it is easy to setup animation.
- **Developer process**, I need a good and efficient data structure to select the topology.

For the creation process a good approach is the "**edge loops"** or **"face loops"**, which Is a chain of connected edges (loops) following some important lines depicting the nature of the model. For some area of this objects the loops are used for depicting in a better way the details, for others area the vertices are distributed in a more uniform way.



They are usually two options for representing faces of a mesh, 3 or 4 vertices, no more then that because is computationally inefficient.

Triangles :

- Very good quality
- They are surely **coplanar.**
- Easy intersection test
- Very easy to computer the normal
- They are more difficult to create complex meshes.

Quads :

- They aren't surely coplanar.
- They are easter to create correct topology.

In the end a quad is made by two triangles, and it will be converted to these triangles (by the modeler or by the API, like during the tessellation stages) so in the end you have triangles (this is what the GPU will use).

Euler's polyhedron formula

Put in relation number of vertices and faces.

V = Vertices

E = Edges

F = Faces

g = surface genus

$$V - E + F = 2 - 2g$$

If the surface has no holes ($g = 0$), it is called "closed", and F are triangles which means that is made by three edges (multiply by 3) and each of these edges is shared by two faces (divide by 2), we get :

$$V - \frac{3}{2}F + F = V - \frac{1}{2}F = 2$$

The **most important attribute** of the mesh is the **normal**, which is a versor perpendicular to the surface.

Mesh production techniques

- **Box modeling** is a technique in 3D modeling where a primitive shape (such as a box, cylinder, sphere, etc.) is used to make the basic shape of the final model.
- **Digital sculpting**, which is heavily used for movies, is the use of software that offers tools to push, pull, smooth, grab, pinch or otherwise manipulate a digital object as if it were made of a real-life substance such as clay.
- **Procedural modeling**
  - **Height map**, this by using a noise calculated on an image (DEM .- digital elevation)or procedurally
- **Using parametric surfaces**, we use the tessellation shader performing the evaluation.
  - **Bezier**
  - **NURBS**
- **3D Scanning**, not used actively where we acquire the points corresponding to the real object and then we have to heavily elaborate these vertices. After that we do triangulation in order to create the final mesh.

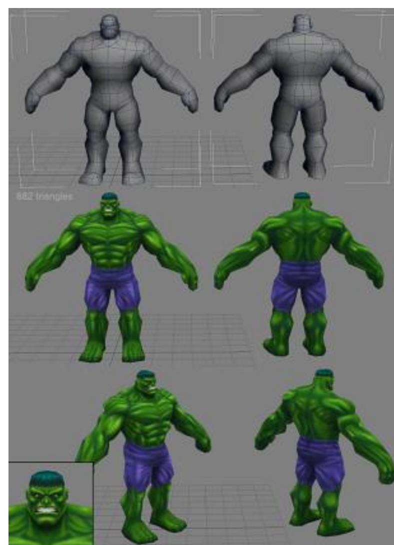
High poly vs Low poly

This is one of the main differences in graphics between online and offline rendering. In the **offline rendering** i want the very final level of detail to be perfectly detailed, so I don’t care about the number of polygons, where in the online rendering I care of the graphics but mainly about the performance of this graphics in my game.

Same model but different topology.

In case of **low poly**, we compromise the mesh complexity for achieve higher performance, this processing It’s involved on games with more cartoonish effect and fantasy.

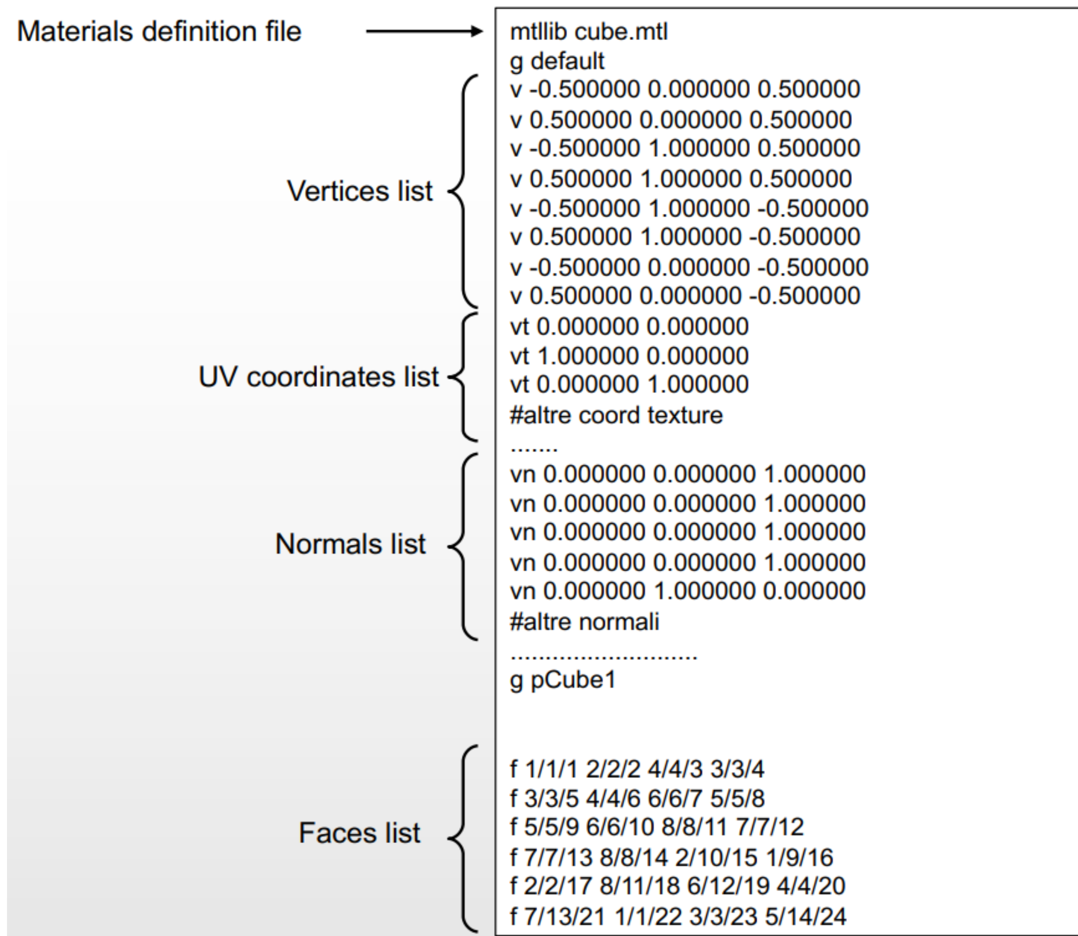
There is a little trick, where it is possible to use a low-poly mesh with a detailed texture for achieve a good-looking effect.



Main 3D file formats

There is not a standard, but anyway there is a trend to use something more uniform among the different possible application :

- **OBJ** format made by Wavefront (now fused with alias, they made Maya), it is a text-based format .. It is a really notorious format. Used for describing :
    - Geometry
    - It is possible to use an external file .mtl to describe materials. Inside the .obj file there will be a reference to the .mtl

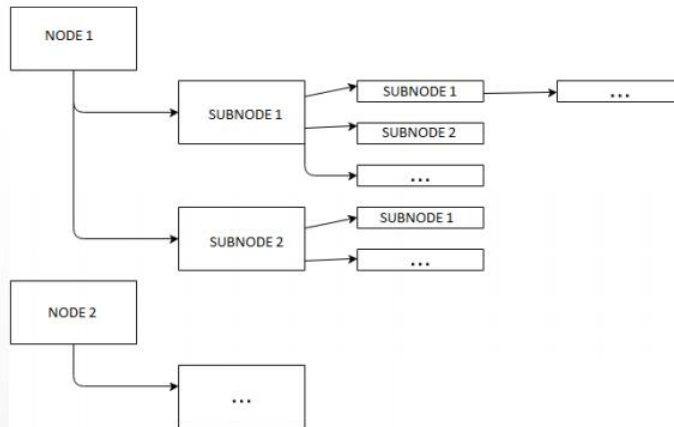


It is a simple file to parse in a program to retrieve the data about the object. The bottom part of the list contains how the faces are composed of the object 1/1/1 means the first vertex + first uv coordinate + first normal vector.

OBJ is really diffused and still a lot used and well supported, the problem of **.obj** is given by the fact that it is good only for static meshes (this is the major limitation).

- **FBX** is another quite diffuse format, initially created for motion capture data (used in movie for moving virtual character) it is currently owned by Autodesk and isn't open as format. There is an SDK in C++ and Python for support the format in engines for other software. It is more capable of OBJ because he can store animation and world scenes (not only single model like with **.obj**). A loader for an .fbx file is more complex than one for .obj.

FBX file structure



"Objects" node: structure

```
Objects:  { <---- beginning of node Objects
   Model: "model name", "Mesh" { <---- beginning of node of the model
      […]
      Vertices: […]       <---- vertices
      PolygonVertexIndex: […]  <---- indices
      LayerElementNormal: { }  <---- node of the normals
      LayerElementUV: { }  <---- node of the UV coords
   } <---- end of node of the model
   Material: "material name", "" { } <---- node of the material
   […]
} <---- end of node Objects
```

- In the last years there is another format called **COLLADA** (Collaborative Design Activity) (.dae, Digital Asset Exchange) proposed by **Khronos Group.** The aim of this format is to store everything possible (geometry, shader, physical simulation, …) inside a single file based on XML, this for give higher flexibility. The main issue with **COLLADA** is the compatibility between *importers* and *exporters.*

- **glTF**, since COLLADA didn't get famous as expected the Khronos Group proposed another format for efficient transmission and loading of 3D Scenes, also called the "*JPEG of 3D*". This format works even for distributed graphics application like mobile games, multiplayer games, … there is a need of transferring models and assets among servers and clients in the most efficient and fast way. webGL born for minimize the assets of these models. They intended this format as more usable as possible, it supports different kind animations, support Physically Based Materials (PBR) and the approach was a bit more efficient respect of COLLADA, because there is lot of tools for managing the input/output of the data.

  Structure of the file :
  - **JSON** containing the description of the hierarchy of the file
  - .bin binary external file containing all the actual information
  - .png/.jpg for texture

  Having this component separated means that json can points to different files allocated in different servers. So, you can receive the .gltf and inside you will find the reference to the remaining files on other servers (lot of flexibility).The main idea of .glTF is to be the final

format used by tools of creation and engines or even other frameworks that need to work with the final model.

COLLADA has the role of exchange of assets, so keeping all the information to use in another production software, while **glTF** has really the one to use in the final part of the process (inside an API).

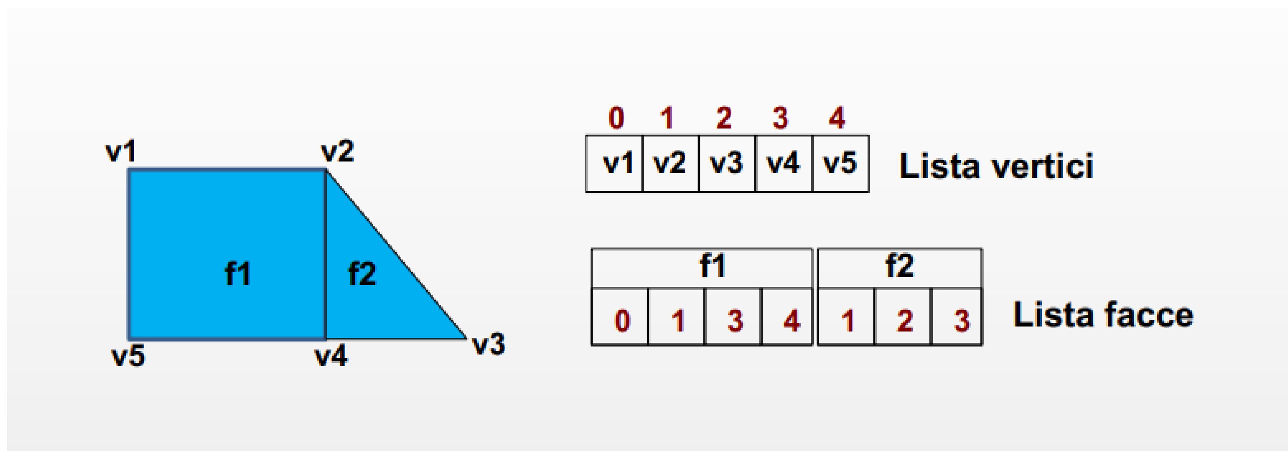
Data Structure for Polygon Mesh

We need two data structure for the Polygon Mesh:

1. For the mesh elaboration in the **Application Stage.**
2. Another data structure for the **GPU** per-vertex elaboration inside the **Geometry Stage.**

Indexed mesh

In some way similar how the **.obj** format file are expressed.



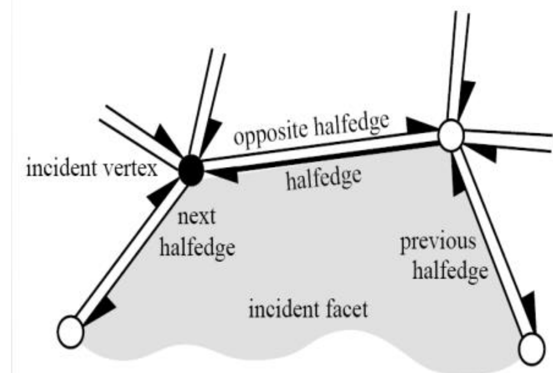
We have a list of vertices and a list of faces, where the index of the list of vertices are used to describe the faces of the mesh.

There is an option to this, where instead of a list of vertices you use and edge list (it is a bit more redundant).

This data structure is easy, but they aren't efficient for elaboration, because if you need to process one face or one edge you have to parse all the arrays. So usually, you need something more complex but more performant the **half-edge** or the **winged edge** (very similar, we are not going to see it).

Half-edge

You have a model, and you have edges and faces; you take one edge and you considered it not just as a single edge, but you consider it has made by two vectors pointing from and to the vertices in between.



So, in the data structure you maintain the information about :

- The vertex at the end of the half-edge.
- The next half-edge.
- The internal face of the half-edge.
- The two half-edges pointing in the opposite direction.

```
struct HE_vert
{
        float x;
        float y;
        float z;


        //half-edge from the vertex
        HE_edge* edge;
};
```

```
struct HE_face
{
       //a half-edge
       HE_edge* edge;
};
```

```
struct HE_edge
{
       //vertices  at the end of the half-edge
       HE_vert* vert;
       //"twin" half-edge
       HE_edge* pair;

       //internal face
       HE_face* face;

       //next half-edge
       HE_edge* next;
};
```

Why you do this ? In this situation if you have to elaborate one single primitive of the mesh, I can in a linear time assess to all the neighborhood of the mesh.

e.g., if I have the half-edge and I can cycle in all the edges of the face, I can do it iteratively accessing the pointers of next edges.

```
//task: to find vertices and faces connected to
an edge

HE_vert* vert1 = edge->vert;
HE_vert* vert2 = edge->pair->vert;

HE_face* face1 = edge->face;
HE_face* face2 = edge->pair->face;
```

```
//task: to find and elaborate all the edges
composing a face

HE_edge* edge = face->edge;
   do {

     // do something with edge
     ........
           edge = edge->next;

   } while (edge != face->edge);
```

face
opposite
inc. vertex
edge
next
face

This is a really effective data structure for elaboration, I use something like the half-edge in the Application Stage, but actually the GPU wants the **Indexed Mesh** data structure for do his own computation.

So usually what you do is :

1. Load a **half-edge.**
2. Do stuff on the mesh.
3. Convert the half-edge in **Indexed Mesh.**
4. Copy the information on the **GPU.**

Why we need to have a copy of the indexed mesh in the memory and then on the GPU?

The first version of **OpenGL** we didn't have the transfer of vertices data on the VRAM it was transferred directly by the central memory, it was called **immediate mode.** It was really slow, in particular for the bus transfer (was the most common bottleneck in the pipeline).

After OpenGL 3.0 they created the **Vertex Buffer Object** is an area in the **VRAM** which stored the buffer of vertices expressed in **Indexed Mesh** approach, ofc is faster since the data copied is maintained in the memory and will be read directly from the GPU. This kind of architecture has some weakness :

1. You keep a copy of the data between the Application Stage and Geometry Stage. But actually, for static meshes there is a caching mechanism for not transferring again for each frame.

Mesh Elaboration

One of the things that happens on the mesh on Application Stage are some kinds of preliminary elaboration in order to get ready for the GPU processing.

There are three possible set of operations :

1. Tessellation
2. Consolidation
3. Simplification

If needed, they are applied only once, after loading stage (performed by the auxiliary lib like GLM, or by specific classes), other cases, they may be dynamically applied at runtime.

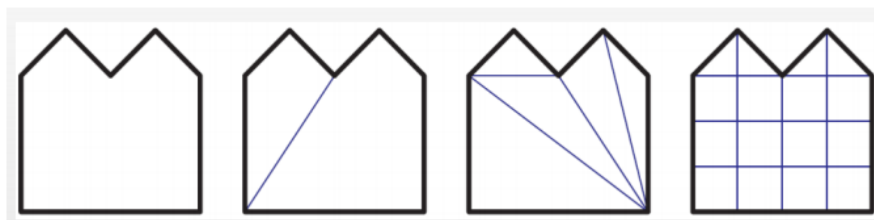
Tessellation

Usually condition to use the tessellation, is that the model has polygons with more than four edges (higher primitive order than triangle), and they should be subdivided.

So the operation is needed to create triangles in the end, starting with a higher primitive order instead of triangle. In the end you can do any kind of tessellation you want. In general tessellation means that I want to subdivide the mesh in a particular way, tessellation it's a triangulation (most common case). Other criteria are the convexity (must be), the maximum area preset and the regular generation of sub-polygons with a fixed scheme.

It isn't an easy process, and it can give some several possible issues with the model geometry. This occurs with non-planar polygons and warping quads, but this problems origin from the model itself.

This process can also be iterative, and with more iteration I can keep refine this result.
It got enabled in OpenGL after the 4.1 release, it uses the Tessellation shaders, it is used



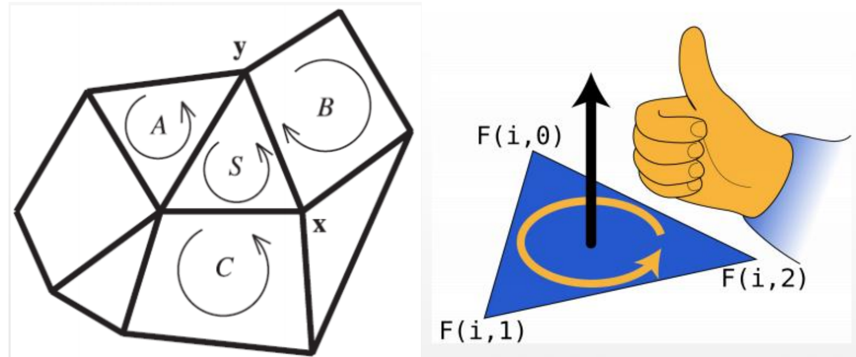
Consolidation

One procedure commonly applied to polygons is **merging**, which finds shared vertices among faces. Another operation is called **orientation**, where all polygons forming a surface are made to face the same direction. Orienting a mesh is important for several different algorithms, such as blackface culling, crease edge detection, and correct collision detection and response. Related to orientation is **vertex normal generation**, where surfaces are made to look smooth. We call all these types of techniques **consolidation algorithms**.

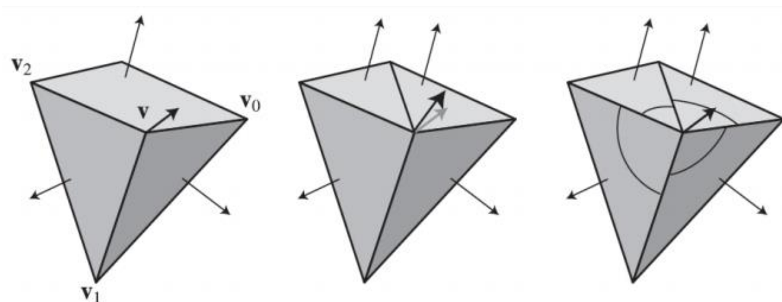
- **Merging,** I can do some optimization like merging the mesh, a model maybe can have different area disconnect, where they are seen as a single mesh but actually are two putted together. They visually look like they are sharing the edge, but actually in the memory distinct triangles are involved, this is waste of memory and information. You get what so called **Polygon soup**. In this stage before sending data on the **GPU** you try to merge the double vertices and you try to avoid the repetition. Sometimes two vertices are so closed that can be collapsed, this operation is called **welding**.

- **Orientation**, how the order of vertices composing the faces are distributed inside the data structure. The direction of the normal is given by the order of the vertices following the

Right-Hand Rule. Counterclockwise order you get the normal pointing up, if the order go clockwise the normal points down.



- **Normal smoothing**, the computation isn't difficult but it's tricky. You have the three normals for the faces and you average in other to get the v vector in the center of the image. The final vector is the vertex normal. On the left the bottom side is a quad, means that they are 2 triangles so 2 normals equals, so counting the other 2 edges we got 4 normals and the resulting average vector isn't correct because the two on the top got double weight. The correct way should consider this issue, the convenient technique consists in the use of the angle between the face and the normal as weight for the average.



Simplification

It is also called data reduction or decimation; the idea is that I have the mesh, but I want to reduce the number of the polygons (I want to use less vertices). I want to discard vertices, but I want the final mesh to maintain the appearance of the original one. In RTGP less polygons we have the faster the process will be, we may need to decimate a polygon because it's to heavy and I want to maintain high performance.

There are three approached to simplification :

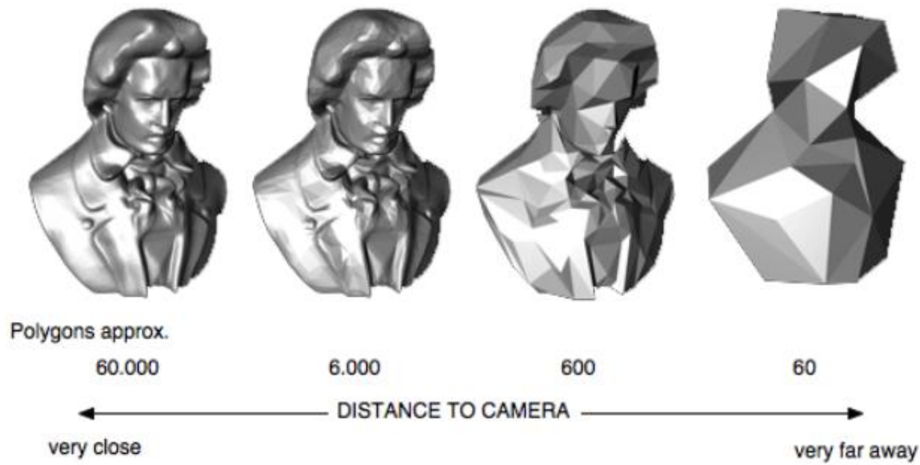
- A static approach, **Level of Detail** (**LOD**)
- A dynamic approach, **Edge Collapse**
- A view-dependent approach, **Terrain Rendering**

Level of Detail (LOD) (Static)

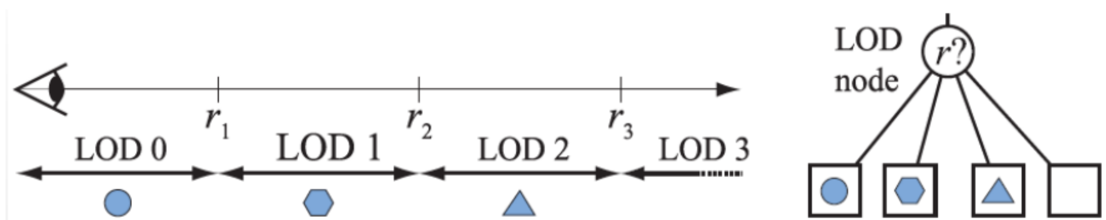
When I have a mesh very close to the camera I will need lot of details, but when the same model is far from the camera it may cover very few pixels in the final image. But in the end the same number of pixels are computed by the GPU, the idea in LOD is to improve performance I have a model composed by different version : the final one with lot of polygons and different version at different level of simplification when I need to use the model I choose the more appropriate version

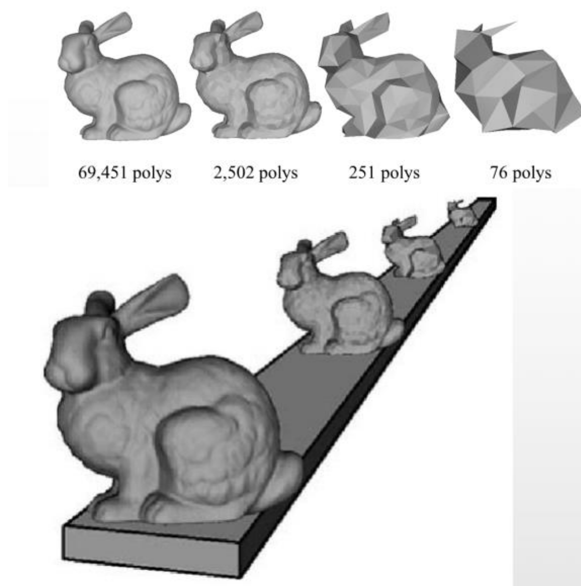
in base at the distance from the camera. The more distant is from the camera the simpler one I will choose.

I need to create the different versions then at the runtime I need tochoose one and I need to switch at runtime the new version with the one currently used; the generation is something that in the classical LOD is done before the opening of the application by the modeler.



In the old days there was a phenomenon called *popping*, the computational power was low and the **LOD** at a certain distance totally remove the meshes. So suddenly after the meshes get into the visible step, the mesh suddenly pop out!

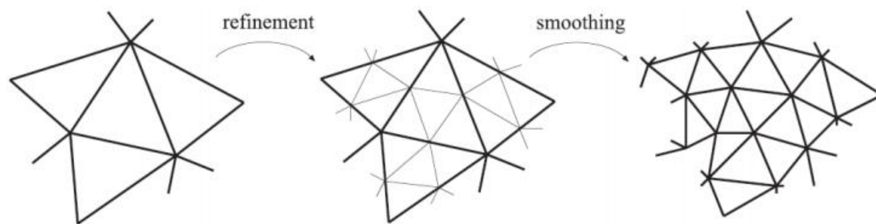




A fix was to do a blending approach to make a soft transition; this will make the computation more intense but usually the transition is short. Instead of blending another opportunity is to use the transparency, the model became transparent while it moves away, again this improves the *popping* effect, but performance is better only after the model disappear.

Subdivision surfaces

It is used mainly for high-poly models for animations, is an iterative or recursive technique.

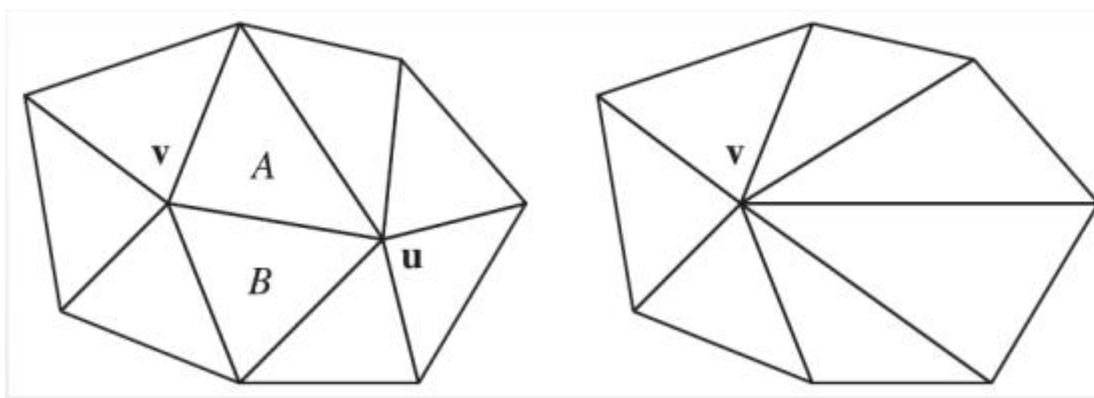


You have the first mesh then you create new vertices in the midpoints of the original triangle and then you create the edges and faces, and obviously you create the curvature (Icosphere…). There are different techniques the most famous is the **Catmull-Clark**, was one of the found of Pixar (the code is now open-source), at the moment the subdivision of surfaces is used mainly for film animation.

The selection of the level of detail is done by taking as reference the distance of the camera, this means that the space is subdivided in discrete steps and each step is assigned a version of the model.

Edge Collapsing (Dynamic)

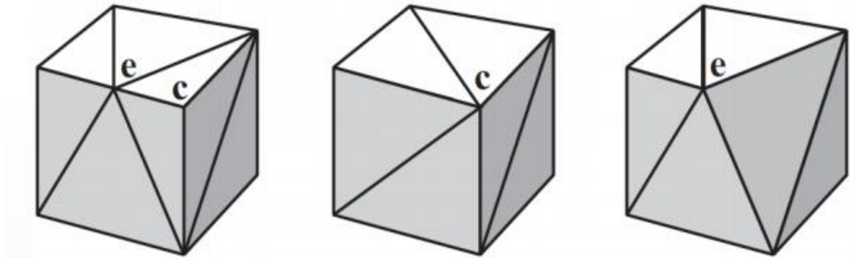
Even if there are this approach to soften the problem of the popping to have a better switching effect, in the last year considering the tessellation stages we are moving to a **dynamic simplification** of the model having the LOD created on the fly directly on the mesh. These techniques are based on **edge collapse** or **edge contraction** techniques. The idea is to delete the vertex u and this comport the disappearing of the face a and b.



The idea is based on analysis of the local neighborhood to determine best option for the vertex to collapse. Usually, you consider the vertex and you assign a cost to each of them, this means that I take the set of triangles share the vertex, each triangle has one plane, so iconsider the equatiuon of the plane and the cost function is the sum of the squared distances between the potential new location of the vertex and the planes of the triangles sharing **p.**

$$c(\boldsymbol{p}) = \sum_{i=1}^{m}(\boldsymbol{n}_i \cdot p + d_i)^2$$

Example with a cube of size 2, suppose that we have cost(e) = 0 and cost(c) = 1, in the first case means that we can collapse e into c, e doesn't move from the planes it shares. In the second case we collapse c into e, ci moves away the plane on the right face of the cube



This process of edge collapsing is also **reversable**. If you keep trace the collapsing that you have done, you can go back in the reverse order, without popping because we are dynamically change it, this is the **CLOD**, **Continuous Level of Detail**. The tessellation Shaders can be used to perform Dynamic Simplification.

Terrain Rendering (view-dependant)

The simplification is applied only to part of the model far from the camera, this is typical of outdoor scenes where I have large terrain.

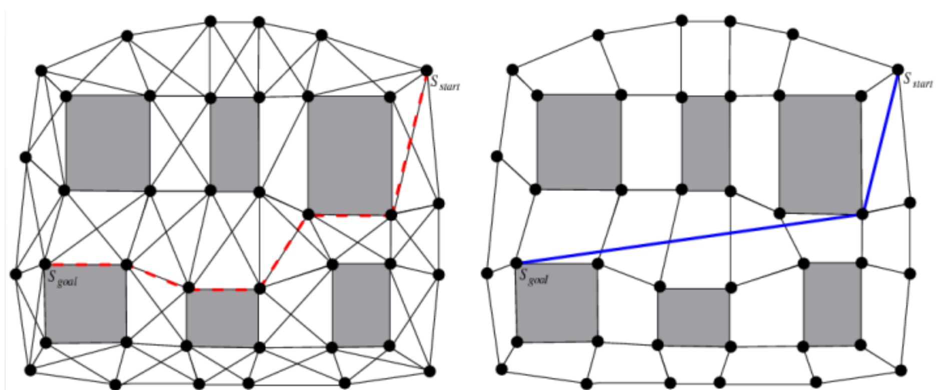
Other uses of meshes (Final Notes)

The mesh you have loaded in the **RAM** in the Application Stage aren't used only for rendering but even for other purposes in the Application stage like the physics simulation, AI , … needs this information in a videogame there is a need of lot of other activities.

The topology of objects\scene may be used not for rendering but for other purpose (mesh used as a graph).

For example, the **navigation mesh.**

A navigation mesh is a mesh exploited as a graph the purpose is to use it for pathfinding algorithms in the game. The AI for videogames needs the pathfinding of the mesh for determine a path, it can use a subset of original mesh, defined by the level designer or by automatic detection.



Transformations and Projections

They are applied in different stages, the transformations are used for :

- **Modeling**, this means the placement of models (or parts of complex models) in the scene.
- **Camera ,** this means the placement and orientation of the virtual camera and by giving a sense of perspective.
- **Animation,** the transformations are applied and changed in real-time to create the illusion of a smooth movement.

A **cartesian coordinate system** or a **frame**, is composed by :

- **1 Origin**
- 3-Dimensional **orthonormal basis**

Possible coordinates systems :

- **Local coordinates** (or **local space**), where the position is expressed with respect to the origin of the model, which usually is in the center of the object.
- **World coordinates** (or **world space**), from the moment I place the model inside the scene the coordinate system changes, now the coordinates of the model are expressed respect to the origin of the **World Scene**.

  In the moment I load the model and I put in my scene, in the first very moment before applying the transformation the model will be placed at the origin of the world then it is moved.

The real name (or nature) of transformations are call it **Linear Transformation**, a linear function which has these features :

$$f : V \rightarrow V$$

- $f(v + w) = f(v) + f(w), \quad \forall\ v$ and w in the domain of f
- $f(cv) = cf(v), \quad \forall$ scalar c

The transformations are not all linear like **rotation** and **scaling**, linear transformations usually are the ones who apply a transformation of vertices around the origin.

The **translation** isn't linear transformation since the transformation isn't defined around the origin.

A peculiar characteristic of a **linear transformation** is that can be expressed by an **invertible matrix**.

$$M = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

(e.g., a 2x2 matrix)

Scale Transformation

Then I apply the linear transformation matrix as a function (since it is) to a point that I want to change, and for do that you use the multiplication (M*x == M(x)).

The scaling matrix 2x2 is this :

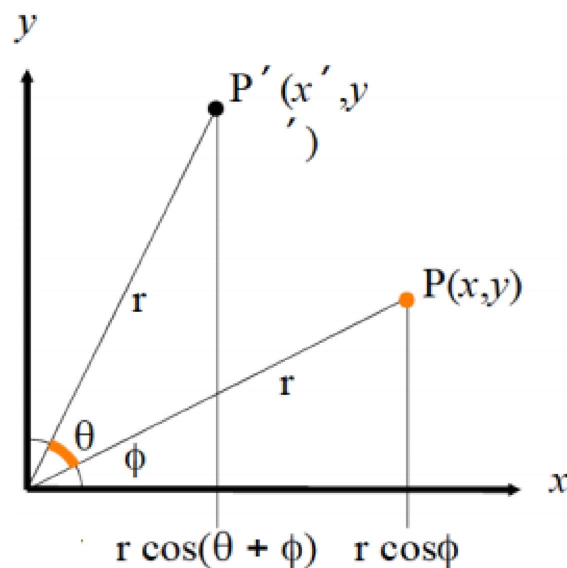
$$S = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

Properties of scaling matrix :

- Scaling Factor < 1 , the model gets closer to the origin.
- Scaling factor > 1, the model gets further from origin.
- Sx != Sy, this means that is a **non-uniform scaling**, means that angle isn't preserved (and so proportions).

Rotation Transformation

Is a different kind of transformation, we can express the coordinate x,y by using the polar coordinates (defined by magnitude and argument, where the magnitude gives us the distance from the origin and the argument will say the direction of our point).



Where the starting coordinates are expressed as :

$$x = rcos\phi$$
$$y = rsin\phi$$

And the rotated coordinates are :

$$x' = rcos(\theta + \phi)$$
$$= rcos\theta cos\phi - rsin\theta sin\phi$$
$$= rcos\theta \frac{x}{r} - rsin\theta \frac{y}{r}$$
$$= xcos\theta - ysin\theta$$

$$y' = rsin(\theta + \phi)$$
$$= rsin\theta cos\phi + rcos\theta sin\phi$$
$$= \cdots = xsin\theta + ycos\theta$$

Everything is expressed by using a 2x2 matrix :

$$P' = \boldsymbol{R} \cdot P$$

$$P = \begin{bmatrix} x \\ y \end{bmatrix}; \qquad P' = \begin{bmatrix} x' \\ y' \end{bmatrix}; \qquad \boldsymbol{R} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Translation

As said, the translation is not a **linear transformation**, this means that isn't possible to use a matrix for represent that, it is expressed as a vector sum.

$$p' = p + \boldsymbol{t}; \qquad \boldsymbol{t} = \begin{bmatrix} dx \\ dy \end{bmatrix}$$

Homogeneous Coordinates

Is a peculiar **coordinate system** used in the field of **projective geometry**, this field has an extra dimension to the others which is called w, this four-dimensional space is called **projective space** and coordinated in projective space are called **homogeneous coordinates**.

This new coordinate w determines the scaling of the other components

<u>Properties</u> of w

- The w coordinate must be **different** from 0.
- There is a proportion to be maintained with the w coordinates and the *x* and *y* coordinates (or *and z* in case of 3D transformation).

$$w' \neq 0, \qquad \frac{x'}{w'} = x, \qquad \frac{y'}{w'} = y$$

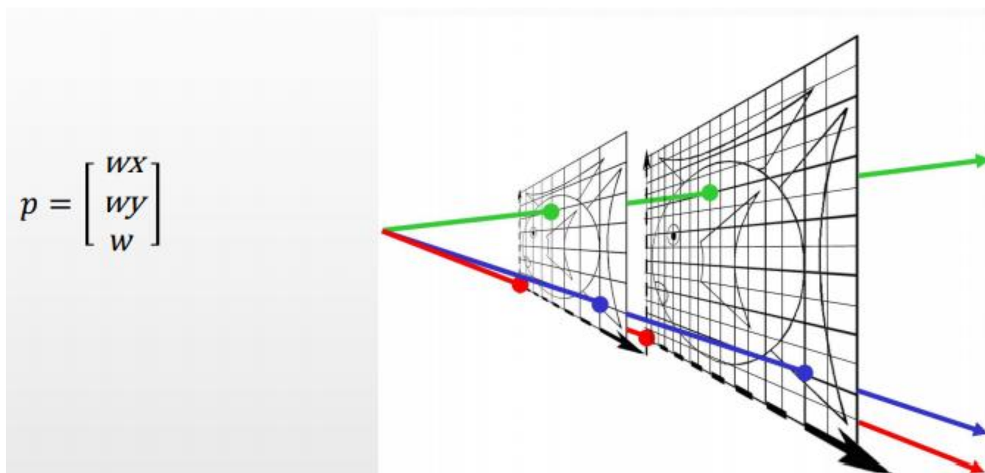
Inhomogeneous point : $P(x, y, z)$

Homogeneous point : $P(x', y', z', w')$

The main idea is this :

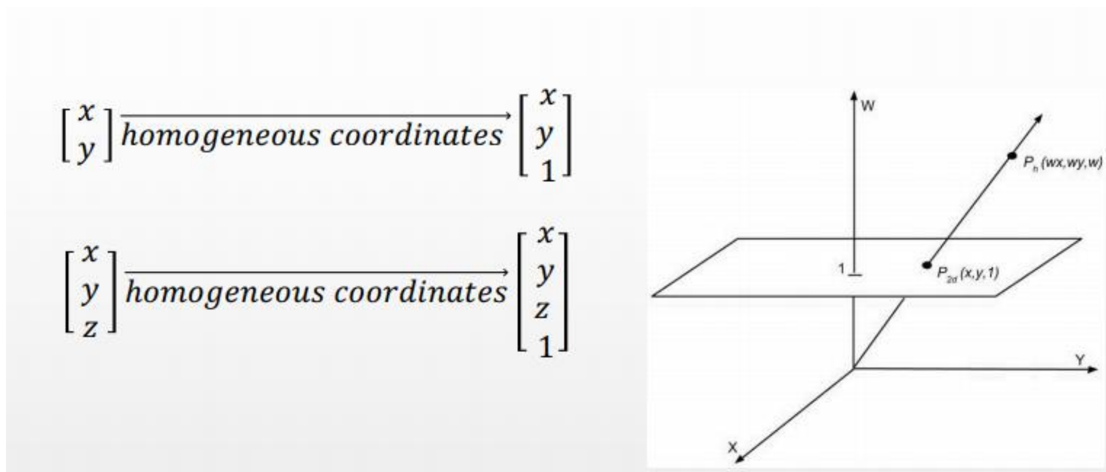
If we start from a 2D cartesian space and we move to his homogeneous 3D space, we get a set of parallel planes. This means that any point expressed on the 2D plane will be expressed in the remaining parallel planes of this projective space.

Then we can say that each point in the starting space (2D plane) has an infinite set of equivalents points in the homogeneous space. This points from different planes differs by a factor of *w*. So, we can say that the *w* is the <u>identifier</u> of a parallel plane in the projective space.



- When $w = 1$ we are in the **canonical form** of the space (x and y are multiplied by 1). When you scale the coordinates by 1, the coordinates don't shrink or grow, it just stays the same size. If $w > 1$ everything will look smaller, vice versa for $w < 1$.

- When $w = 0$ this is the convention used for defining a vector in the infinite space, by the way this will cause your program to crash because will try to divide by 0.



The transformations that use **homogeneous** coordinates are called **affine transformation**, where lines and planes are preserved and so as well the parallelism, angles are not necessarily preserved. An affine transformation is any composition of a **linear transform** with a **translation.**

The transformations are defined in such way with homogeneous coordinates, (so now they are affine transformations) :

Scaling

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Rotation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Translation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Inverse transformation

Scaling

$$\begin{bmatrix} 1/s_x & 0 & 0 \\ 0 & 1/s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Rotation

- $R^{-1} = R^T$

$$\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Translation

$$\begin{bmatrix} 1 & 0 & -d_x \\ 0 & 1 & -d_y \\ 0 & 0 & 1 \end{bmatrix}$$

In the case of scaling matrix , there is a particular scaling where the factor is -1, this is called **reflection** and it is done on the axes why got the scaling factor of -1.

Reflection across x axis

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Reflection across y axis

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Reflection across origin

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

There is another transformation which isn't really common called **shear** or **skew**, it tilts the model along one axis, it is a sort of deformation could be :

Horizontal shear $\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & sh_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$

Vertical shear $\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$

Horizontal + vertical shear $\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & sh_x & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$

The parameters of the inclination are expressed by using an angle of :

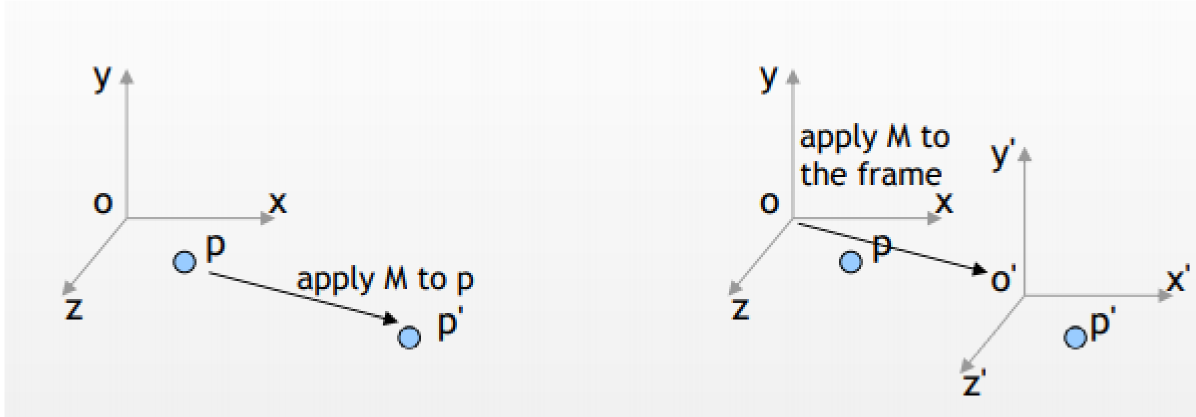
$$sh_x = \frac{1}{tan\theta}$$

It is very good to apply **composition** of transformation (functions), and it is really simple to do it with matrices you just have to **multiply** between them. But, you have to take care about the order of the product since matrices product is **<u>not</u> commutative.**

- The convention says that the order to follow is from the right to left, in base at what you have to do.
  - E.g. : If I want to Translate-Rotate-Scale, then I will have to multiply the matrices in this way : Scale*Rotate*Translate
- In the case that i want to invert an entire composition of transformations, then you will have to apply the inverse of each transformation of the compositions in the reverse order.
  - E.g., Translate-Rotate-Scale , then I will have to InverseScale-InverseRotate-InverseTranslate

Interpretation of transformations

There isn't a right one, but there is another interpretation of the transformations. We have considered the transformations to be applied to vertices but actually is possible to consider the transformations to be applied to the whole reference system.



Affine Transformations in the 3D

Scaling

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translation

$$\begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Shear

$$\begin{bmatrix} 1 & sh_{yx} & sh_{zx} & 0 \\ sh_{xy} & 1 & sh_{zy} & 0 \\ sh_{xz} & sh_{yz} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Euler Angles

Unfortunately, we don't have only one matrix for rotations, but we have three, one for every axis of rotation. This is following the **Euler Angles** mathematical concept.

$$R_{yz}(\psi) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\psi & -\sin\psi & 0 \\ 0 & \sin\psi & \cos\psi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_{zx}(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_{xy}(\phi) = \begin{bmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

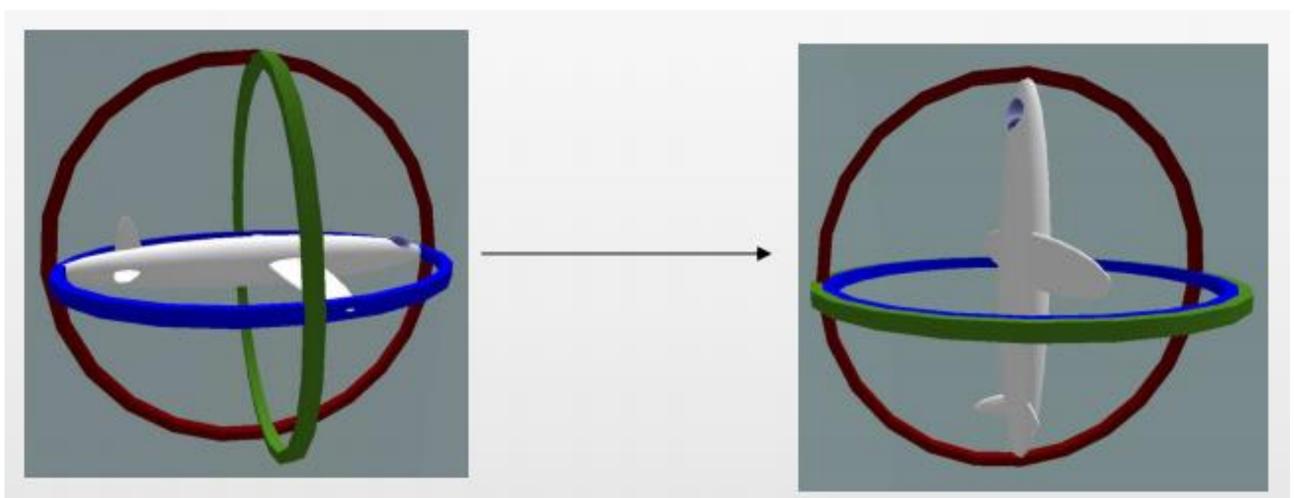
Using 3D rotations to combine the orientation of the object in the space is for sure simple to implement, unfortunately this approach as some issues :

1. We have a problem with the **interpolation**, e.g. We want to move an object in another orientation, usually this is based animation by using *keyframes.* The initial keyframe is setup with a rotation and position and the destination keyframe as well. Then the system will execute the interpolation between these two keyframes. The problem is that with Euler Angles there **isn't a unique interpolation** you don't have control about this. You can't know what the result will be, and probably is different about what do you expect.

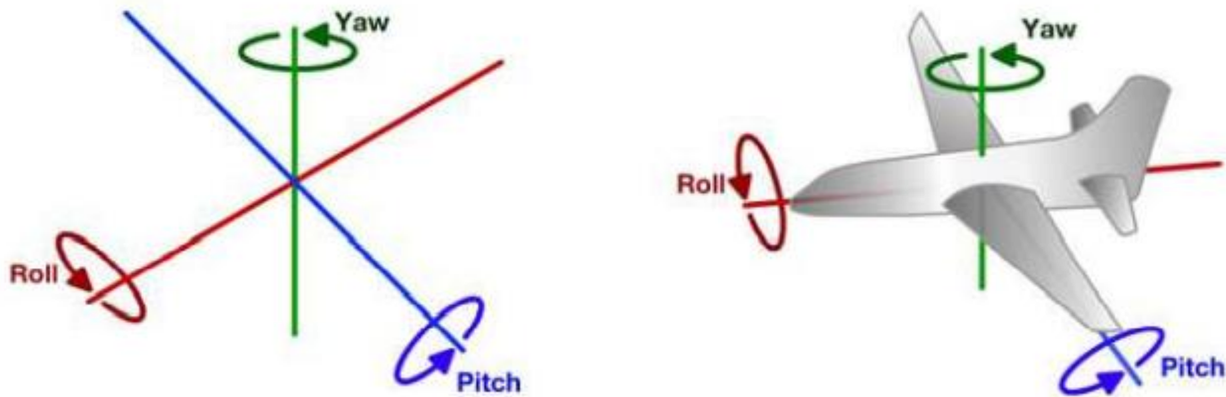
2. Another problematic is given to the fact that you can't invert the rotation after the compositions of matrices.

3. **Gimbal Lock**, this term comes from planes and how the planes flights, when you apply transformation there is a concept called **degrees of freedom** that means how many axes you can consider during the transformation. In our case three degrees of freedom, we can move free above 3 axes.

   If we rotate one axis about 90°, then what happens is that the two axes will collide and became the same, and now I lose one degree of freedom. If now I rotate one of this two axis I will obtain the same result !



Usually, API and engines use **Euler Angles** and matrices for describe rotations, these angles are called Yaw, Roll, Pitch (from planes flight).

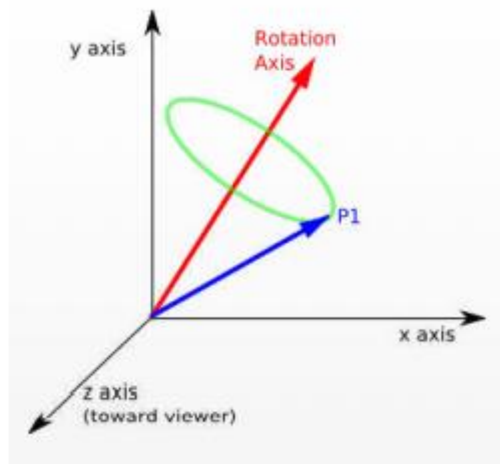


You can avoid the **Gimbal Lock** by **constraining** the rotation, e.g. You don't always all the three rotations, in the camera you can avoid the roll for example. And when you do the rotation you can avoid the full roam , going to 90° and go for a bit less like 89° for avoid the **Gimbal Lock**.

Axis/Angle Rotation

Is another way to describe the rotations instead of using Euler Angles. **Axis/angle rotation**, this is another theorem from Euler, any rotation is expressible by an axis and a vector that rotate by an angle around the axis.

This is more intuitive instead of using 3 separate matrices, and it is adequate for rigid body dynamics.
It doesn't suffer from **Gimbal Lock**, but by the way the issue with **interpolation** is still there.



Quaternions

The method which has no issues with interpolation **Gimbal Lock** involves the use of **quaternions**. From the mathematical point of view, is an extension of **complex numbers** and it extends the axes angle representation by using the complex numbers, it doesn't have any issues with gimbal lock and interpolation because it bases everything on two operations :

1. Conjugation
2. Multiplication

Proposed by Hamilton in 1843 in Dublin, he was a mathematician, and it was searching to find a concept to express a perfect rotation. He was walking to the University and was under the bridge where he got the illumination.

We define the quaternion in complex dimensional space :

$$a + bi + cj + dk$$

a, b, c, d are *real numbers* and usually *a* is a scalar and *b*, *c*, *d* are vector components (of a vector ***p***). So, we can say that the quaternions **q** is q = [a, p].

The imaginary part is made by the i, j, k

And there is a property which says that :

$$i^2 = j^2 = k^2 = ijk = -1$$

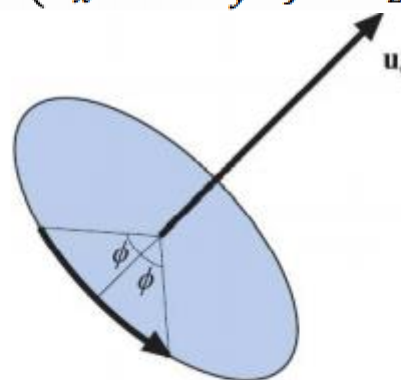
He has proposed an algebra for the quaternions.

- Addition
  - *q1 + q2 = [a1+a2 , p1+p2]*
- Multiplication (associative, not commutative)
  - *q1q2 = [a1a2 - p1p2 , p1xp2 + a1p2 + a2p1]*
  - multiplication of *i,j,k* follows specific rules
- Conjugate Quaternion
  - *q* = [a,-p]*
  - *oppure q* = [-a,p]*

	1	i	j	k
1	1	i	j	k
i	i	-1	k	-j
j	j	-k	-1	i
k	k	j	-i	-1

For rotations we consider the unit quaternion (where the length is = 1), given an axis u and angle phi we express the quaternions like this :

$$q = cos(\phi/2) + (u_x \cdot i + u_y \cdot j + u_z \cdot k) \cdot sin(\phi/2)$$

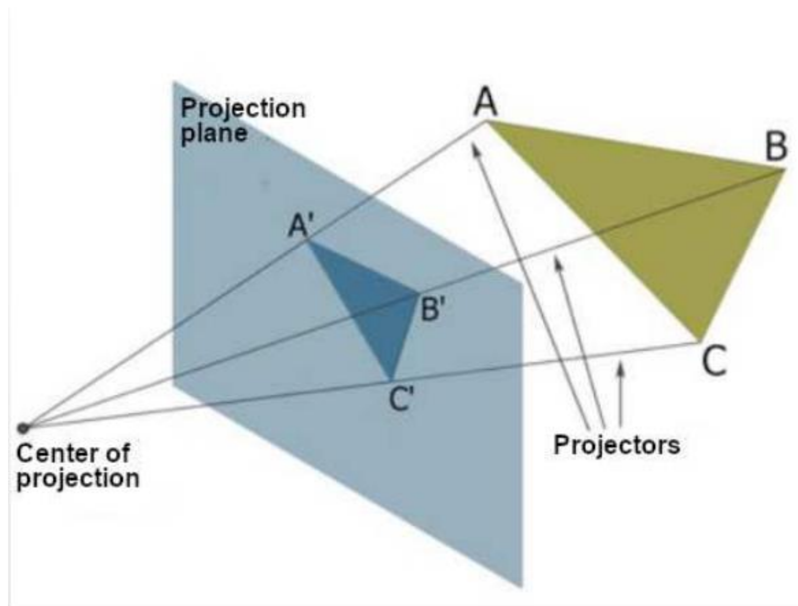


There is a method to convert a quaternion in a matrix, so you can use it quite simple (need to understand the overall concept of quaternion, we don't need the whole mathematical demonstration!).

Perspective projection

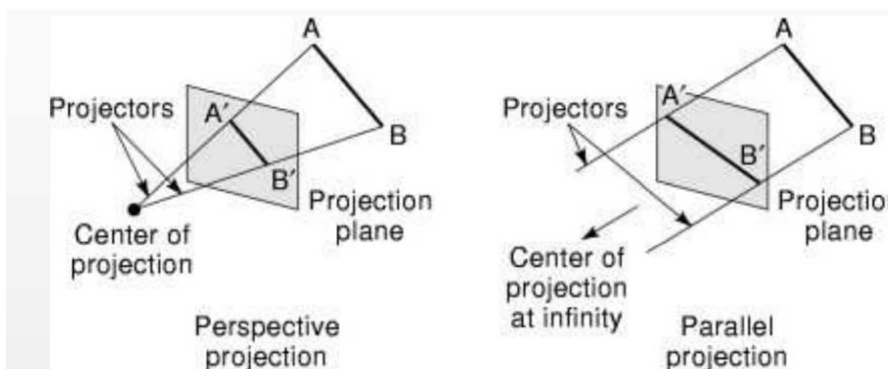
This is another transformation that is being applied, it is called perspective projection, this transformation isn't about the object itself. With projection we do the passage between 3D to 2D.

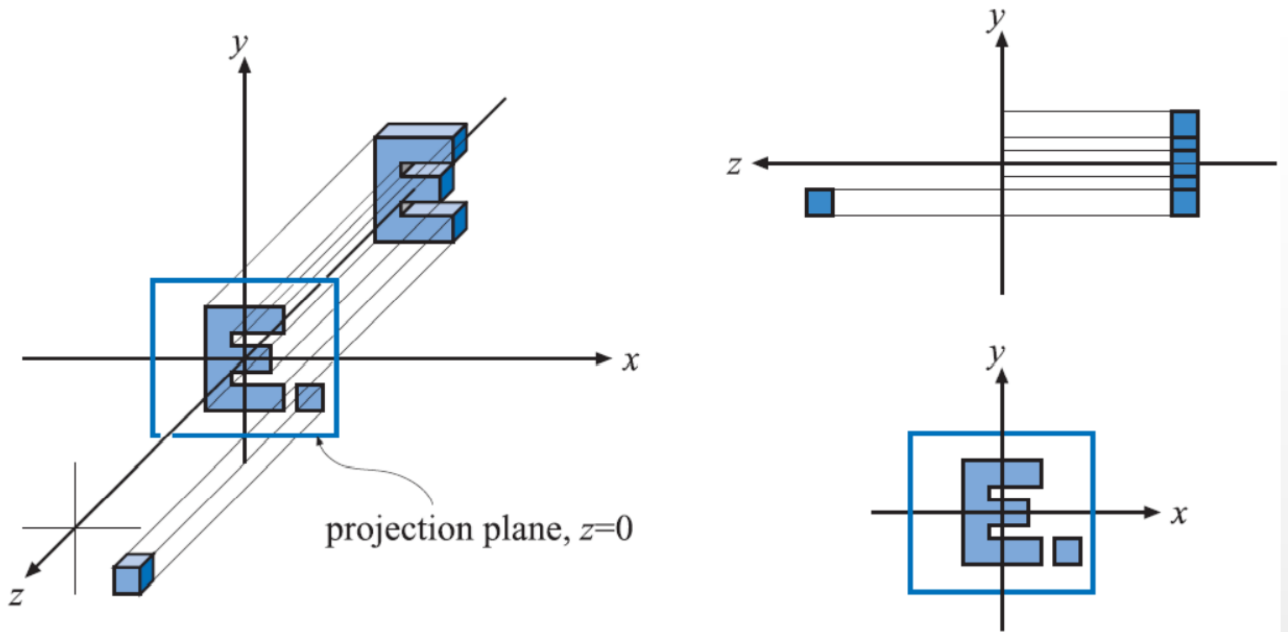
This will setup the position, the perspective and the zoom of the camera so that in the end from the shooting of the scene we will gain a 2D image of a 3D world.



To define a projection, we need one center of projection and a set of lines called projectors from the center of projection to the vertices and a projection place, the plane where I want to project the vertices.

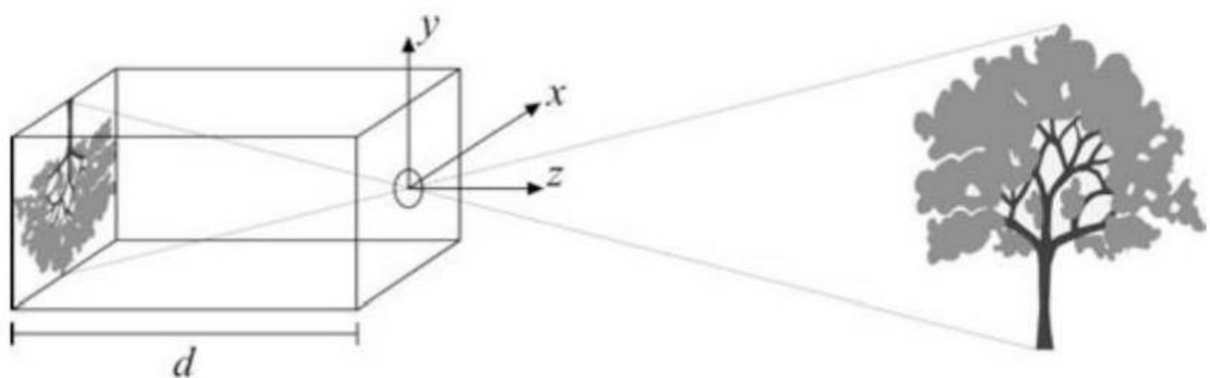
There is another kind of projection above the perspective, the so-called **parallel projection** (or orthogonal projection) where the projectors are parallel.





In Computer Graphics we use a simplified model for the camera, the biggest difference is that in real life we use a lens on the photograph. In the virtual camera from CG, we use this concept similar to the pinhole camera model.

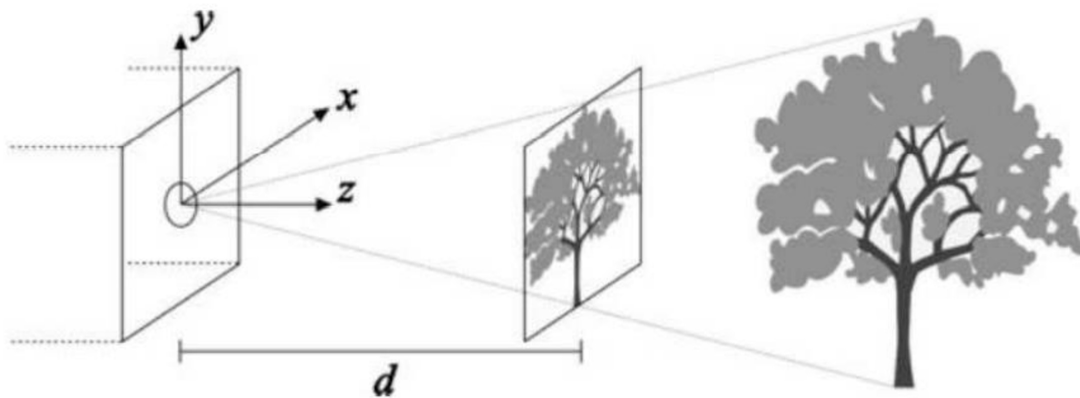
The idea is that the camera is a **pinhole-camera**, which is a box where is placed a hole on an edge (**stenopeic hole**) and the length from that edge to the opposite is a value *d*, which is **called focal** length, by changing that value it changes how the information is acquired.



On the real pinhole the image is saved on a film placed on the other side of the stenopeic hole, the image is saved as **upside-down** this due the ray-light orientation and direction that hits the film.

Without the lens we have some part of the image in focus and other part of the image not in focus, in Computer Graphics everything is completely sharp this to the fact we don't have a physical simulation of the lens (when there is a blur in the image usually it is added in post processing or made by an appropriate engine).

But actually, the real camera convention used is Computer Graphics is another one, which is different from the pinhole camera because the image plane is above the hole.



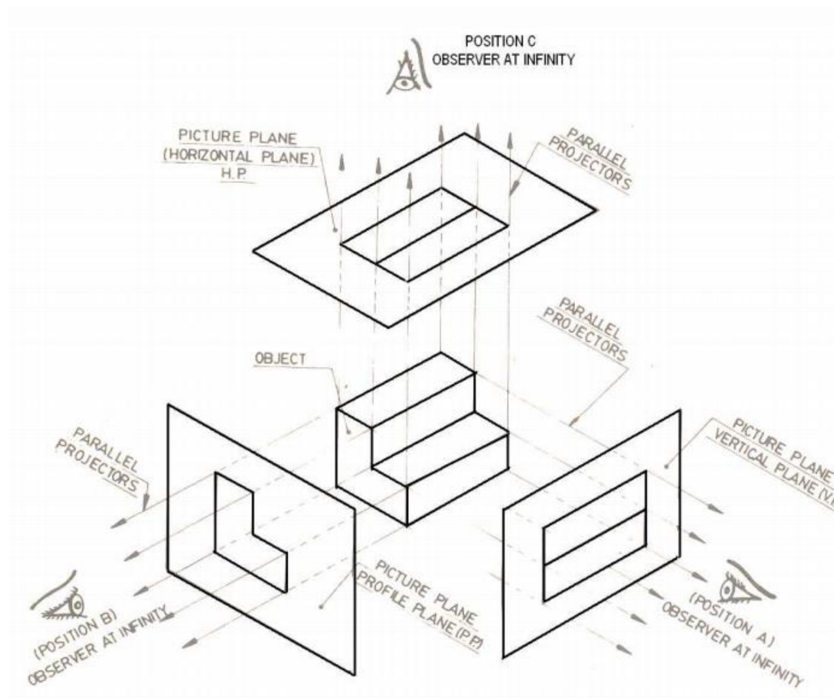
Usually, we use two kinds of projection in Computer Graphics :

- **Parallel projection**, where projectors are parallel to image plane.
- **Perspective projection**.

Canonical configuration for parallel projection

The first thing it has to be define is the **canonical configuration** or also called **basic configuration**.

The canonical configuration for **orthographic** (parallel) **projection** starts from a xy plane which is the **image plane** with projectors parallel to z-axis.



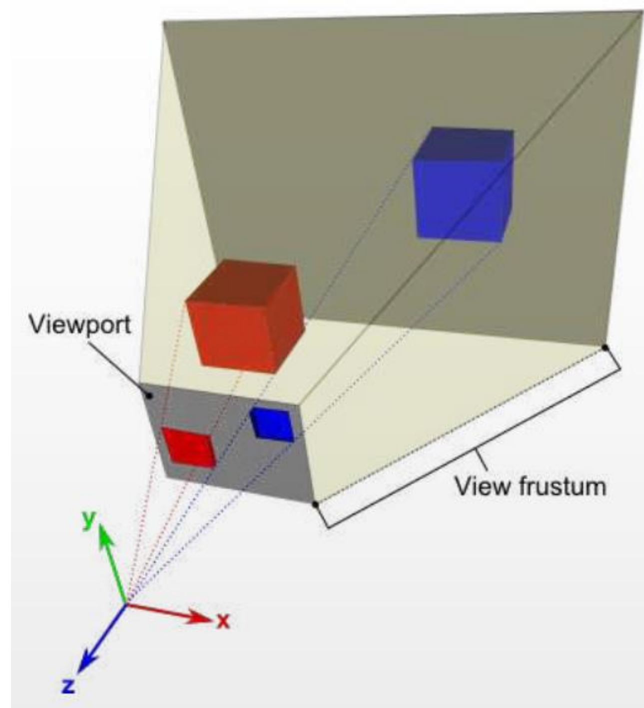
This is done by using a matrix, if you take your original point x, y, z after the parallel projection and multiply by the matrix, this will end by setting the z coordinate to zero.

$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

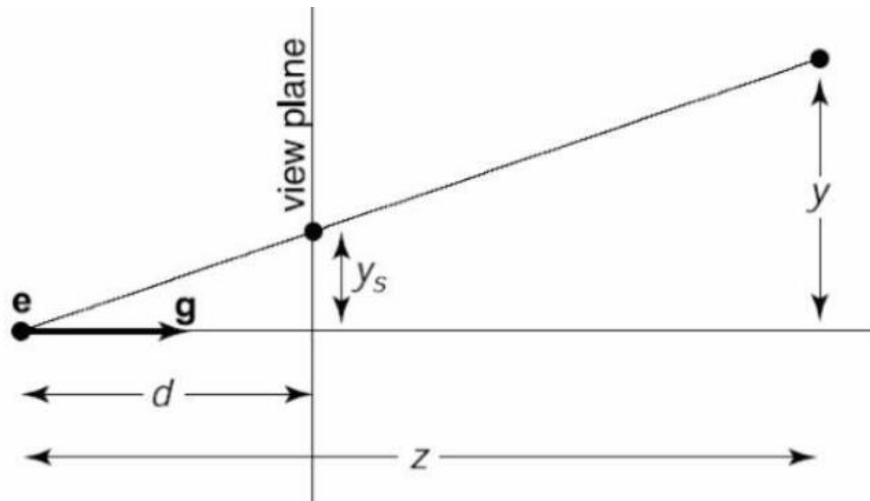
Canonical configuration for perspective projection

The canonical configuration for the perspective projection has the center of the projection in the origin, the main projection axis extends along the negative z-axis (left-handed OpenGL style coordinate system).

The **projection plane** (the first plane) is parallel to the xy **plane** and is set to a certain distance *-d* from the *z-axis* (since we are using negative z-axis), the *-d* distance is the **focal length** of the view-frustum.



This concept is also explainable by using the **similar triangles** :



- d is the **focal length.**
- zs is the z coordinate to project on the projection plane.
- ys the y coordinate to project on the projection plane.

For the similar triangles rule the proportion between $\boldsymbol{ys}$ and $\boldsymbol{y}$, and for $\boldsymbol{xs}$ and $\boldsymbol{x}$, there is a mantained proportion :

$$\frac{y_s}{d} = \frac{y}{z} \longrightarrow y_s = d\frac{y}{z}$$

$$\frac{x_s}{d} = \frac{x}{z} \longrightarrow x_s = d\frac{x}{z}$$

This means we can determine $\boldsymbol{ys}$ and $\boldsymbol{xs}$ by simply taking the ratio between ***y*** and ***z*** and we multiply for the **focal length** (same for the ***x***).

If we express this using a matrix we have something like this :

$$\begin{bmatrix} -d & 0 & 0 & 0 \\ 0 & -d & 0 & 0 \\ 0 & 0 & -d & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

As result we have a vector (should be vertical)

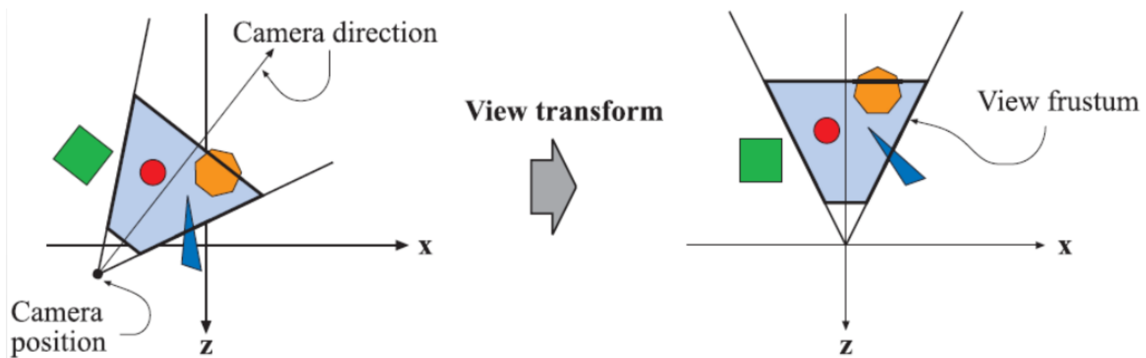
$$[-dx \quad -dy \quad -dz \quad z]$$

To go in the **canonical form** with homogeneous coordinates with *w = 1*, I have to dived everything by z (**perspective division**).

The problem is that I will never use a **canonical configuration** in my scene, usually my camera is placed away from the origin, and it is oriented in a different way, or in an FPS game I keep moving the camera by controlling the avatar.

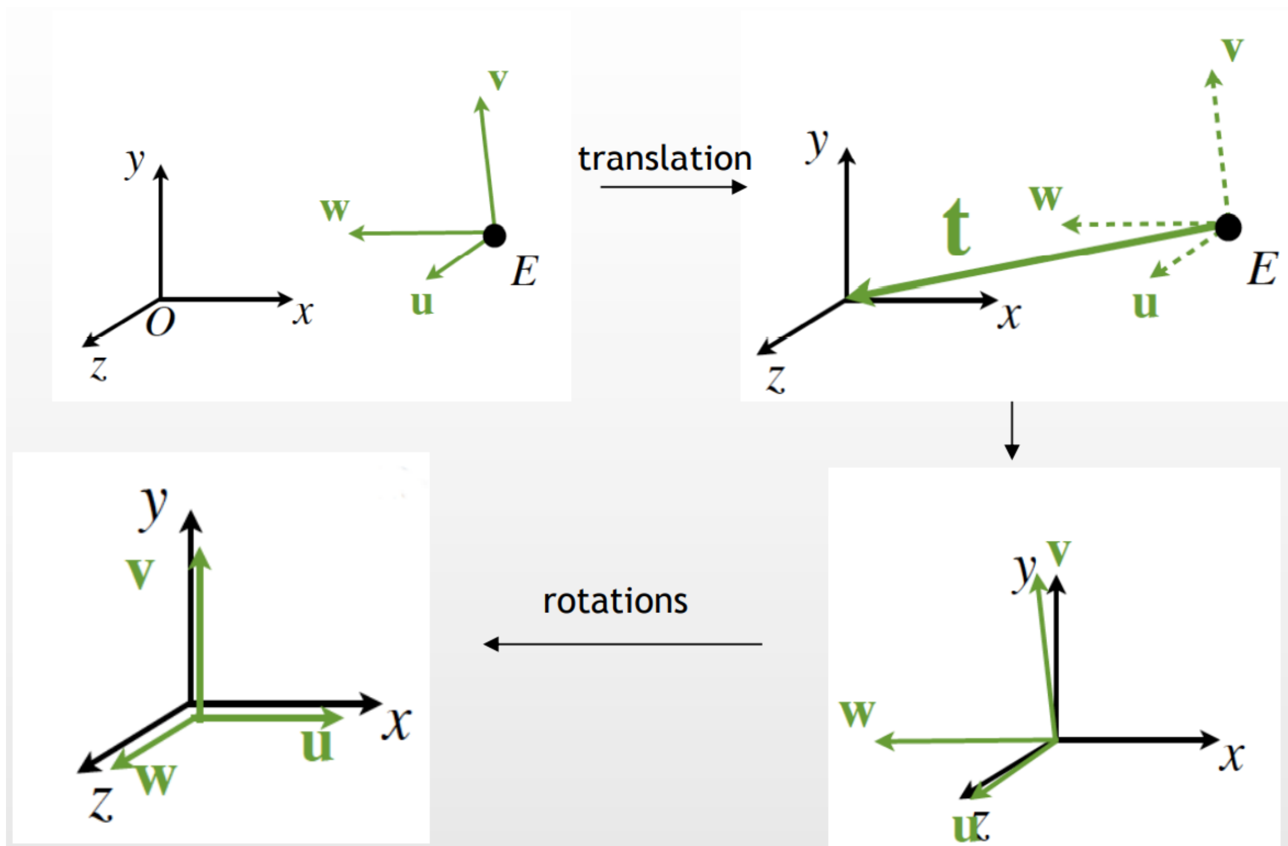
This means that the camera is placed somewhere else (**not** the origin), this means that the projection plane is **not parallel to the xy plane.**

The solution is given by moving whatever orientation the camera has it, in a way that I convert it to the **canonical configuration**, so I rotate the camera until the main axis are equals to the canonical configuration axis, this is called **View Transform**, it is a transformation, and it is done before the **projection transformation** (perspective division).



View Transformation

There are needed translations and rotations for have everything matched with the **canonical configuration**.



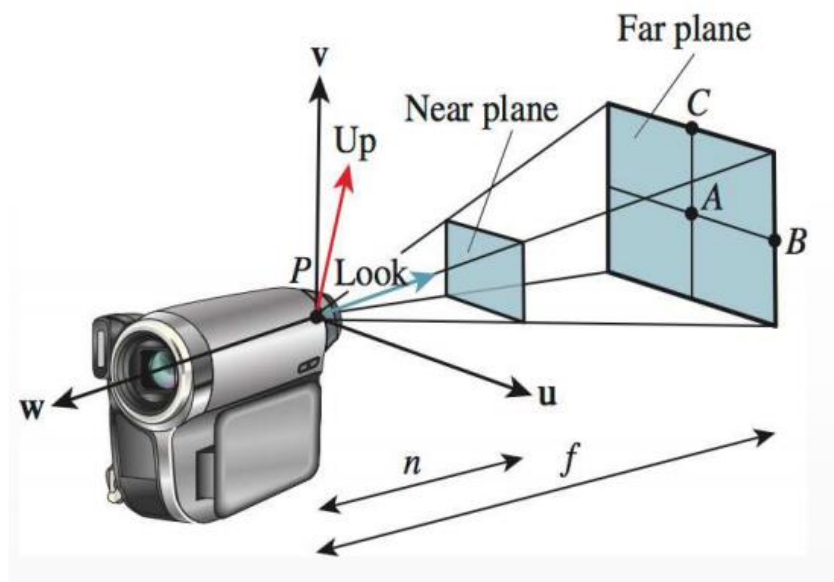
In green we have the coordinate system of the camera, where **E** is the position of the camera. By applying this transformation, we have in the end the reference system of the camera equal to the reference system of the world.

Look At

Often in Computer Graphics we use a convention to define the camera reference system, it is called the **Look At approach**, I need to define in my scene where the camera is placed and where is looking at. We can define the position of the camera in a quite simple way, the point is that could be tricky to define the orientation of the camera by using **Euler Angle**.

The camera frame is defined by an orthonormal basis (in the image $\boldsymbol{u}, \boldsymbol{v}$ and $\boldsymbol{w}$), the position of the camera is defined by a point (in image the point P), the direction of the camera is defined by a vector/versor (in the image the **Look vector**), and the global camera orientation is defined by another vector (in the image the **Up vector**).

The Up vector is mainly used for creating the remaining basis vectors by using the cross product (and sometimes vector sum/sub), the Up vector sometimes is the y-axis, but this isn't always the case especially if you are pitching.



In the end for creating the camera orthonormal basis you just need the Look vector, Up Vector and the point P (this is helpful for determine the vectors). You then use the coordinates of the basis as a rotation matrix.

$$\boldsymbol{w} = -\frac{\boldsymbol{Look}}{||\boldsymbol{Look}||}$$

$$\boldsymbol{u} = \frac{\boldsymbol{Up} \times \boldsymbol{w}}{||\boldsymbol{Up} \times \boldsymbol{w}||}$$

$$\boldsymbol{v} = \boldsymbol{w} \times \boldsymbol{u}$$

Rotation matrix for the View transformation

$$\begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

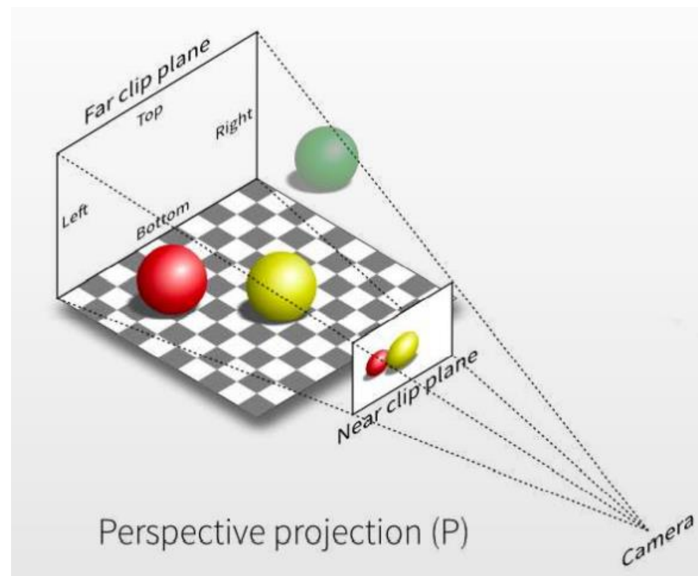
This matrix will rotate the origin to the axis of the camera, but it still needs to be translated to the origin, so we apply a second matrix. It is like saying, move all the world as it (with camera orientation) is to the origin, and then rotate it as it matches the word-origin system.

$$M_{view} = \underbrace{\begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{rotation}} \underbrace{\begin{bmatrix} 1 & 0 & 0 & -E_x \\ 0 & 1 & 0 & -E_y \\ 0 & 0 & 1 & -E_z \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{translation to origin}}$$

In the end it is the composition of the translation to the origin multiplied by the rotation we have just build ,we apply this to all the object in the scenes. This will make all the world being converted in a reference system which is expressed by the camera since now it is place at the origin of the world.

View-Frustum

In Computer Graphics we express not only the center of projection the camera, not only the focal length we even express a so-called **view-frustum**. This is the volume of visible objects placed in the scene. We express this volume as truncated parameters centered in the center of projection and limited by two planes the **near-plane** and the **far-plane**.

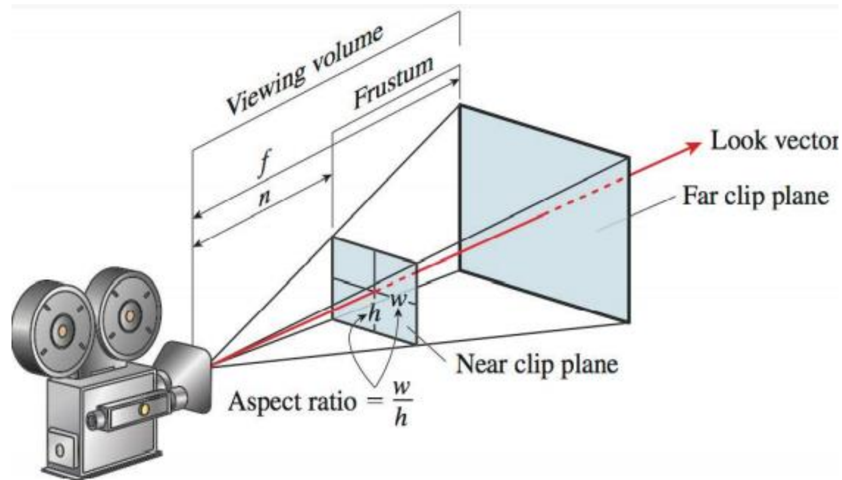


The usually the near-plane is the projection plane, everything which is between the near and the far is visible when an object is more distant then the far-plane then the object is not more visible (going outside the far-plane could end in a "popping" effect).

The **view-frustum** is composed by :

- 4-Sides
- Near-plane
- Far-Plane

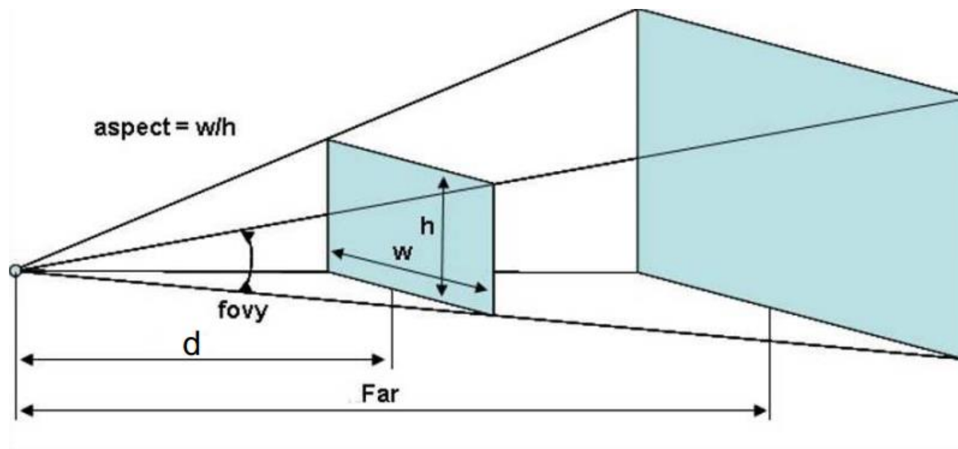
Sometimes the view-frustum is defined by using the aspect-ratio, other times by directly using the position of each side planes composing this truncated pyramid.



The near-plane, aka the projection-plane can be defined by :

- Focal length ***d***, width ***w*** and height ***h***.
- Using **FOV**, which is an angle that express the view volume of the aperture, this is more intuitive, and this approach is similar to the zoom **in** or **out** of the camera.

You can put in relation the focal length ***d*** and the ***w*** and ***h*** of the image plane and the **FOV**:



$$fov_x = 2 \cdot arctan\left(\frac{w}{2d}\right)$$

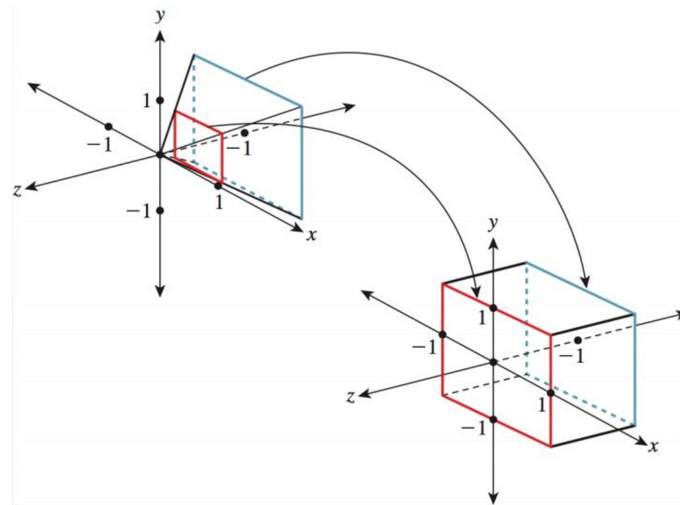
$$fov_y = 2 \cdot arctan\left(\frac{h}{2d}\right)$$

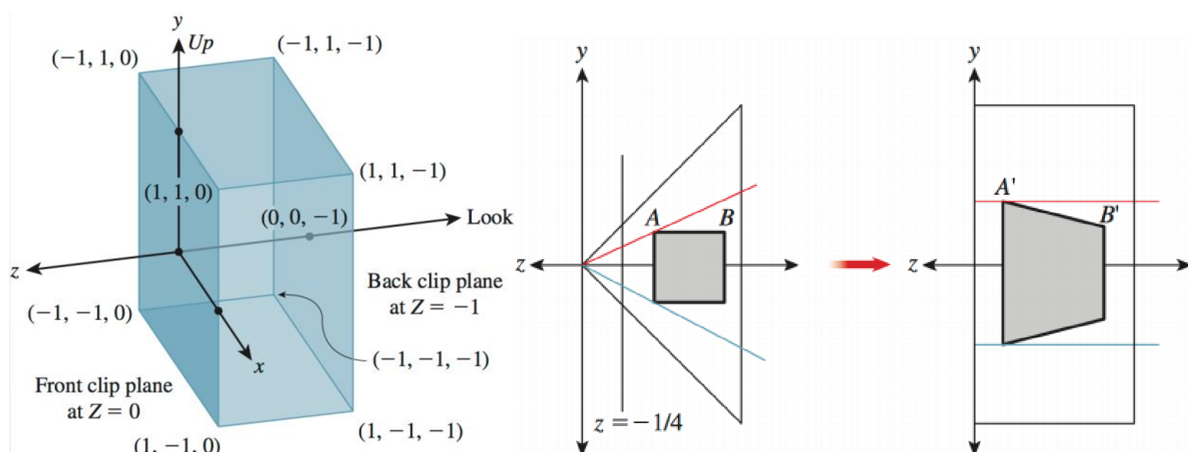
The canonical view volume

Just because we are in a virtual world, we do something which may appear quite strange, but we do it because it really helps in speeding up computations and to help one operation that we will do after **projection** (the **clipping**). What really happens inside the rendering of the GPU pipeline, after the **view transformation** we actually don't apply that matrix once we are in the canonical configuration. What really happens is that all the world will be deformed in such a way that the

view-frustum from a truncated pyramid will become a cube, this **cube** is called the **canonical view volume**.

In this way the **near plane** will be parallel to the xy plane (OpenGL use unit cube, DirectX use the half cube).





When you apply this deformation of the **view-frustum** the coordinates changes and they are called **Normalized Device Coordinates** (NDC).

We do this operation for two reasons :

- The projection does not need anymore the metrics with the *d* on the main diagonal, we don't have to apply the **perspective matrix**, we will apply the matrix for an orthographic projection.
  - Keep remember that the objects inside the **view-volume** will be distorted.
- This change of the perspective projection matrix over the canonical view volume matrix will speed up the computation for the projection and the clipping stage.

As you have noticed in the case of **projection** we have some **not uniform notation** in the various resources, because it depends by the coordinate system adopted.

You may find differences in the matrix used for the conversion to the **canonical view volume**, there are different ways to define this kind of matrix. (In the book there are three or four different version of the same matrix).

- *f*, is the far plane.
- *n,* is the near plane.
- *fovy* is the vertical fov.
- *c,* which is the reciprocal of the tangent function that has fovy/2 as parameter.
- *A* is the aspect ratio.

This is the most common matrix :

$$\begin{bmatrix} c/a & 0 & 0 & 0 \\ 0 & c & 0 & 0 \\ 0 & 0 & -\dfrac{f+n}{f-n} & -\dfrac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Putting everything together we got the canonical-view-volume matrix combined with the transformations needed for the view-transform.

$$M_{persp} = \begin{bmatrix} c/a & 0 & 0 & 0 \\ 0 & c & 0 & 0 \\ 0 & 0 & -\dfrac{f+n}{f-n} & -\dfrac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -E_x \\ 0 & 1 & 0 & -E_y \\ 0 & 0 & 1 & -E_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

With this we have finished the transformation applied in the Vertex Shader.

Window to Viewport transformation

After the clipping, which is done in the **canonical view volume** (we will see it later), there is another transformation done on primitives to **screen coordinates**.

When we convert to the canonical view volume, then we do the clipping which will be easier (and faster) on a cube.

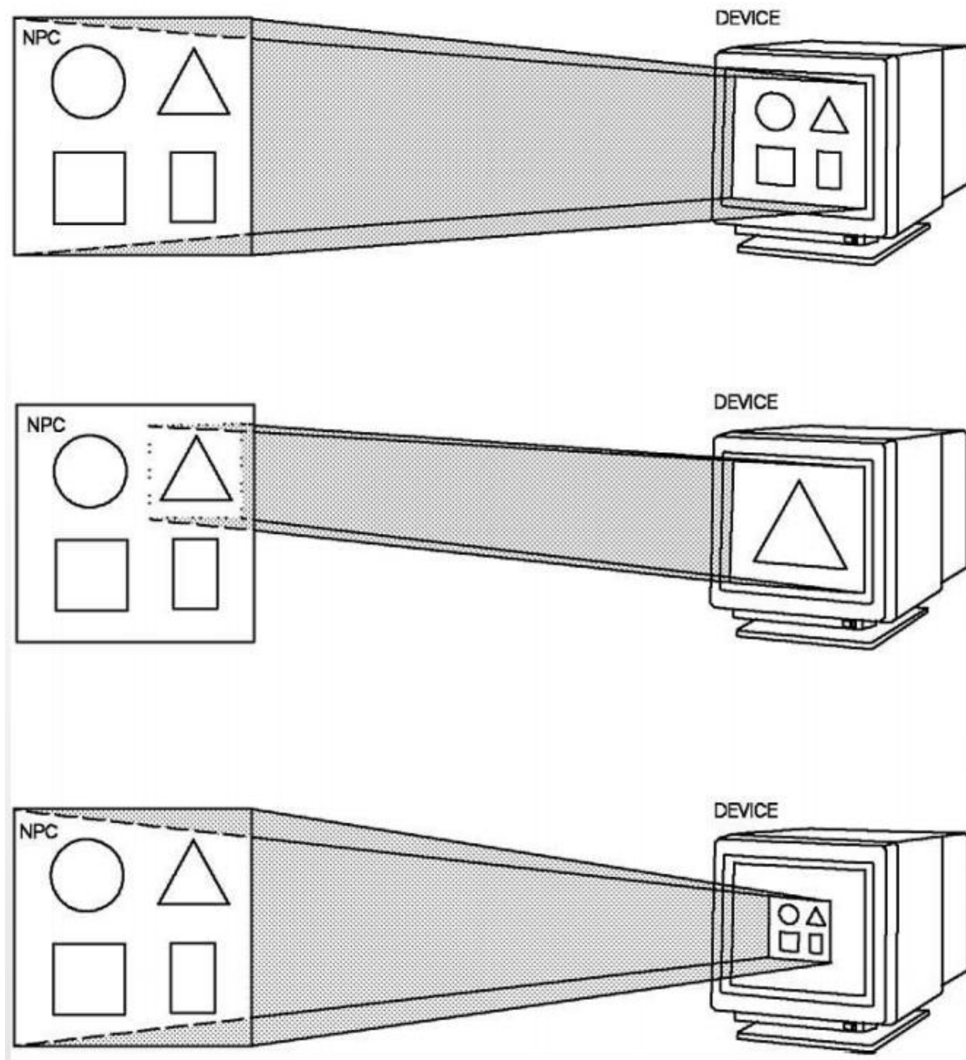
Then we rasterize it, where we elaborate the set of potential pixels called fragments, which if they passed the ***z-test*** they will become the **final pixel** in the scene.

These fragments are created as they would be pixels on the image.

We need to convert the **NDC** to the width and height of the final image on that will be saved on the frame buffer and then showed to the screen, this transformation is called **Window to Viewport**.

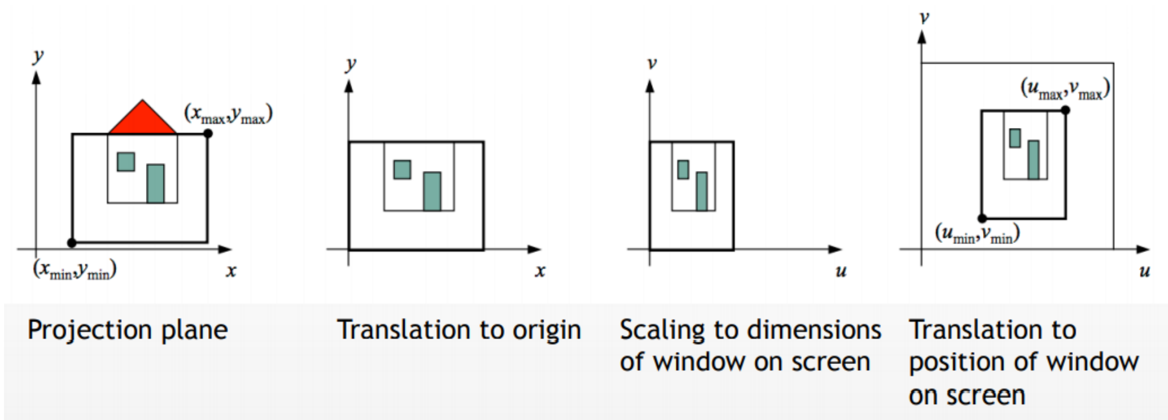
This happens mainly in 2D, since the projected coordinate are **NDC**, and they are expressed inside a range of [-1 to +1] to screen-space coordinates which are expressed in a range from [0 - *number of columns*] and [0 to *number of rows*].

This information is known since we know the dimension of the screen or window where we want to render.



On the left we have the **NDC** image (output from the previous stage) and they are mapped on screen on the right.

What really happens is a composition of **translation** and **scaling** in 2D.



1. The roof of the house (the red) is **clipped**.
2. The projection plane is translated to the origin.
3. The projection plane is scaled to the dimension of the window of the screen.

4. The position of the "window" (the projection plane) is translated (mapped) to the position of the window on the screen.

In the end we have the coordinates of the vertices expressed in **screen coordinates**, when these coordinates will pass to the **rasterizer** all the fragments will be expressed in such a way that if they become "**final pixels**" they will directly go on the screen with no more transformation.

$$M_{viewport} = \begin{bmatrix} 1 & 0 & u_{min} \\ 0 & 1 & v_{min} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{u_{max}-u_{min}}{x_{max}-x_{min}} & 0 & 0 \\ 0 & \frac{v_{max}-v_{min}}{y_{max}-y_{min}} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_{min} \\ 0 & 1 & -y_{min} \\ 0 & 0 & 1 \end{bmatrix}$$

Remember, the order of the projection transformation is :

1. View Tranform
2. Perspective transform. (Canonical View Transform)
3. Clipping
4. Window to Viewport transform.

Spatial Data Structures

There are other techniques that are introduced in the pipeline in several parts, but in particular in the **Application Stage** (before transferring the vertices to GPU).

This family of techniques are made for accelerating rendering, the point is that day by day we are keep increasing the number of polygons to process and we still need to maintain a fast performance of framerate constant.

Spatial data structures are data structures which are used to organize the data of the scene and objects inside my scene in order to **speed up** different computation which will make faster all rendering process.

In the Application Stage we have lot of situations where we have to queuing the actual position of the object, for example the **collision detection** because we want to know if the objects are colliding or not. The spatial data structure will help to organize the objects in a specific area of my scene.

Usually, the data structure are organized in a **hierarchical** way, and their creation is a computationally expansive generation this is why is done in pre-processing. This anyway will comport a performance improvement from **O(n)** to **O(log n).**

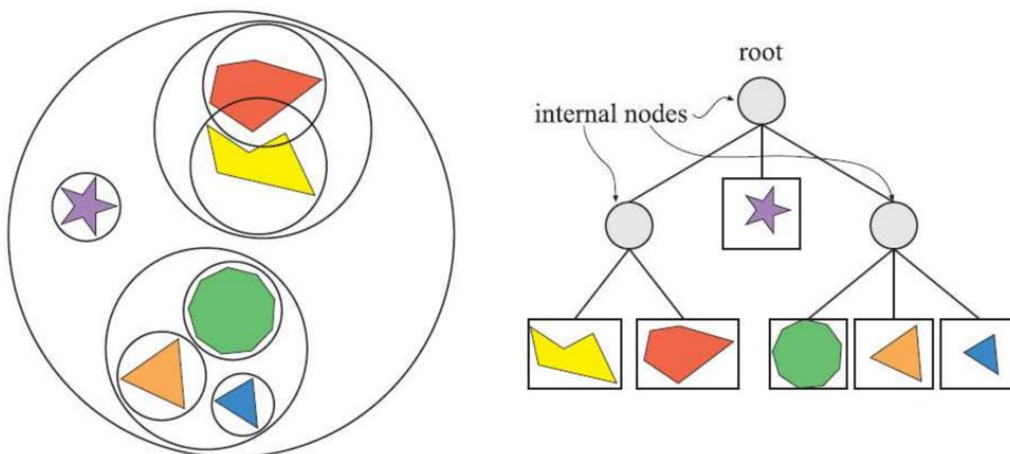
There are different data structure used and some of them can be combined, the most common are:

- **Bounding Volume Hierarchies (BVHs)**
- **Uniform Grid**
- **Octrees**
- **Binary Space Partitioning (BSP) Trees**

Bounding Volume Hierarchies

A Bounding Volume is one of the most used approach, it is a simple shape that represents a volume which **encloses** the original model as tightly as possible, usually we use **sphere** or **boxes** and I take the object and I putted in a sphere or in a box as tight as possible.

They are mainly used for preliminary test, for example to know if they are colliding, the idea is that by using this simple shape for do simple test this test will return some information about the Bounding Volumes that will avoid the program to do more complicated test on the mesh level (test collision with bounding boxes is easier than doing it on).

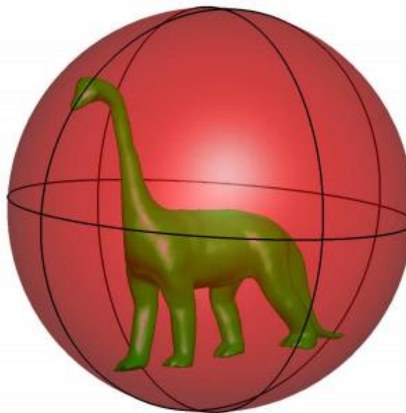


These volumes can be put in hierarchy inside a tree, where the single objects are the leaves. and then I can put in hierarchy different sets of the objects. The three objects on the bottom in the same group. Each leaf got a bounding sphere, and each node of objects will be a Bounding Box.

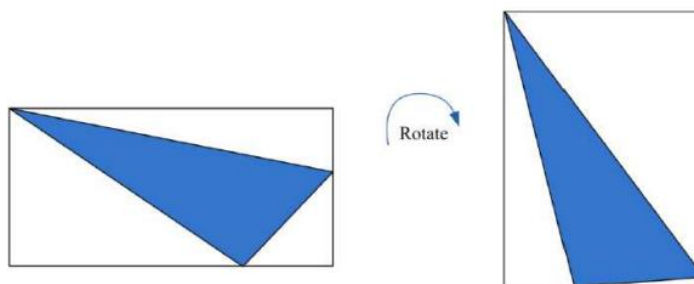
In this way i can check the largest object in the room to see if i need to do some computation in that zone, without having to do it on each child of the node these boxes are affected by some computation.

The Bounding Volumes shapes can be :

- **Bounding sphere**, consider that the sphere is not so easy to build, is not too easy to find the minimum radius of the sphere in order that contains all the object. <u>The sphere is not the tightest of the object, this will result in lot of false positives.</u>



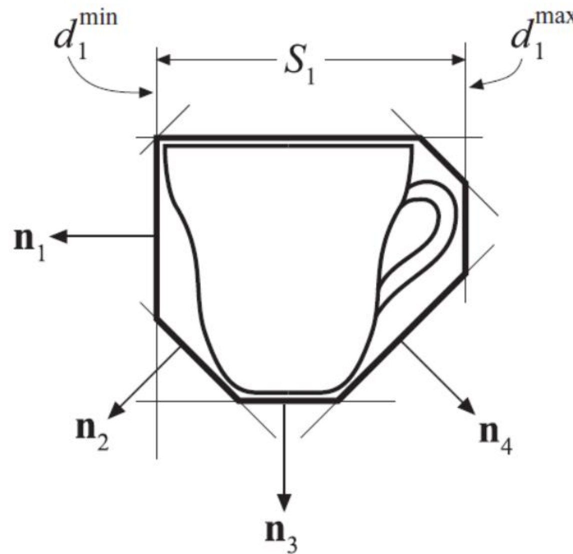
- **Bounding Box**, more efficient from sphere, we have 2 types :
  - **Axis Aligned Bounding Box** is a box which is aligned with the axis of the world, it is **easy to build** so I take the object and I simply process the vertices and I search for the minimum and maximum of the x, y, z axes. It is easy to build and **easy to use it for test**, but the size of the boxes is fixed parallel to the position of the world, means that if the object rotates the box must be rebuilt. **<u>Still not very tight.</u>**



  - **Oriented Bounding Box**, this is a more efficient version of the Bonding Boxes, in this case the box follows the model orientation, it is really tight around the geometry, and we don't need to recompute it when it rotates, because it will rotate with the object.

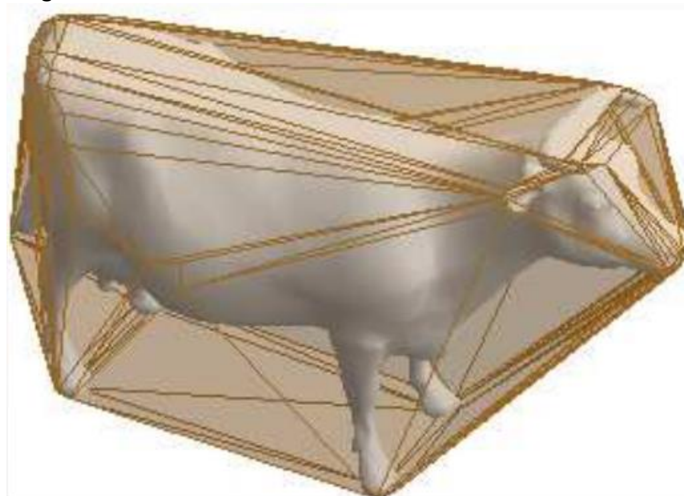
    It is usually built-in local coordinates, when the object is loaded before applying the transformation, is a more bit complicated to build, once the object is created the same transformation applied on the mesh are applied on the box (like a second mesh attached to the object mesh).

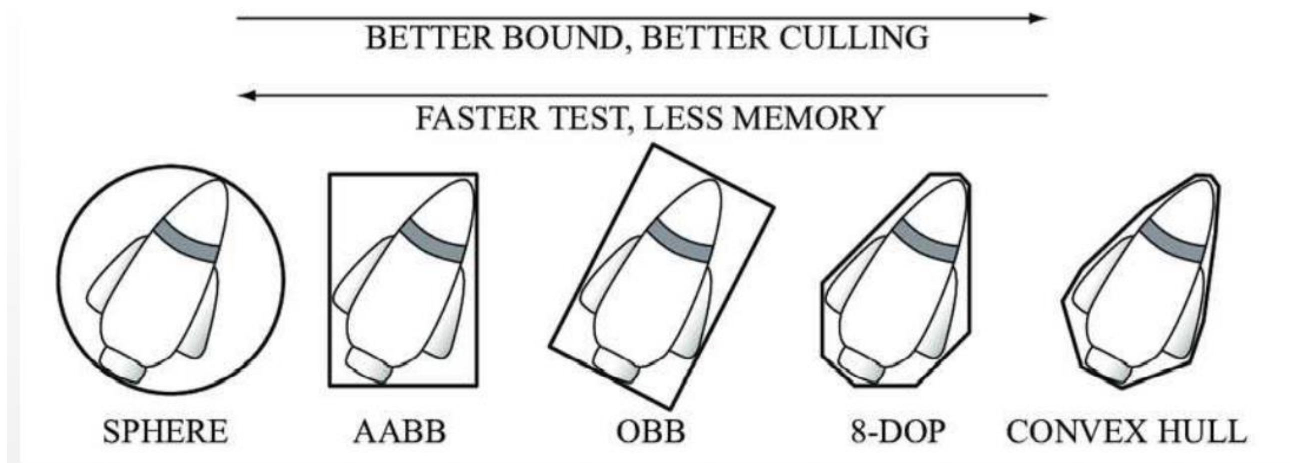
- **K-DOP** (**Discrete Oriented Polytope**) is a generalization of Axis Aligned Bounding Box, we introduce the concept of SLAB.
  The SLAB is a composition of three arguments, a **normal vector** and two values along the vector normal which express two planes perpendicular to the normal vector, the two values will define an interval.



  In the image if we take **n1**, it is a horizontal normal vector with the two values of the slab *d1min* and *d1max* which are giving the minimum and maximum interval parallel to the direction of the normal. So, the slab is given by this three, and it defines a certain area in the space, the idea is that you can define a set of normals and intervals in order to cut a polygon that will be closer to the mesh object.

- <u>**Convex hull**</u>, this is the smallest volume containing the original vertices. The more external vertices of the convex hull are vertices of the original mesh. You build a convex volume which contains the original mesh.





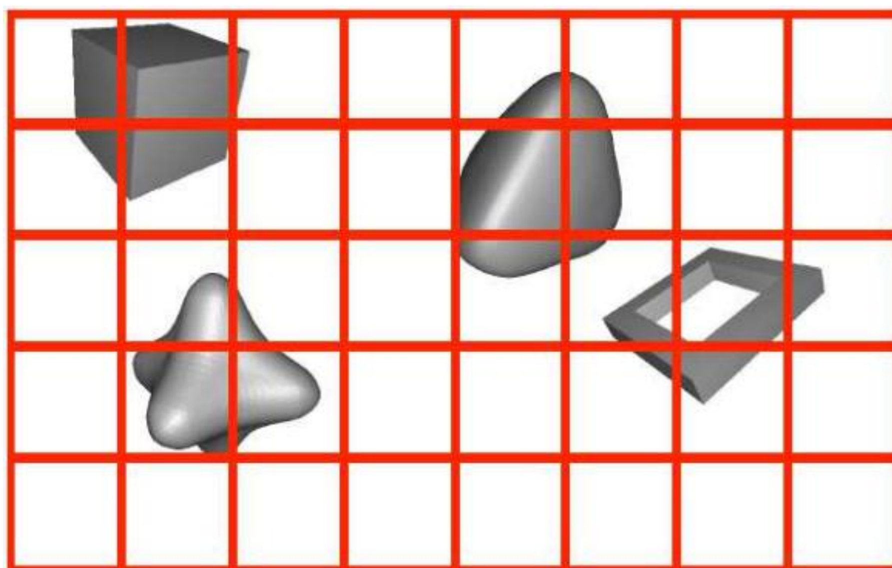
The better it fits the original object the better is the result of the test, more accurate. An intersection test giver you, the simpler is the bounding volume the simpler is the construction and simpler will be the test but obviously less accurate.

Uniform grid

The are other approaches more scene based, we may have many objects similar (like different instances of similar objects) which are placed in the scene.

We may need to do the test about the interaction between these objects with another closer to it. In this case having a Bounding Volume may not be the best choice.

We can divide the scene in **uniform cells,** and we can keep a **table** or **hash table** regarding in which cell the objects are placed. I simply take one cell at the time only among the objects inside one cell since they are close enough to collide.

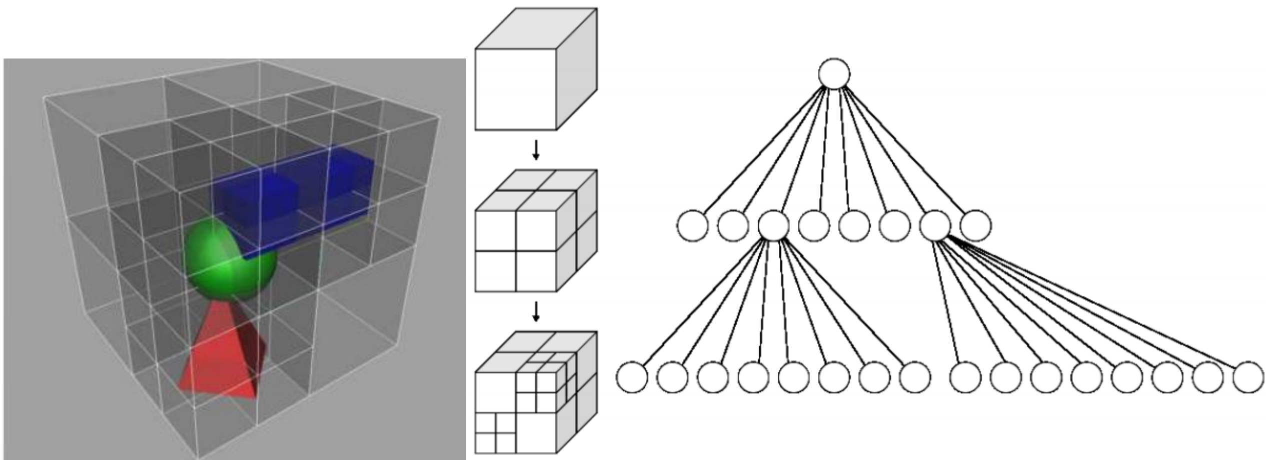


The same is for launching an object and wanting to know which object I’m hitting; I will check one cell to see if there is an object if that is the case I will move on or a more specific test. Even if **uniform grids** have some issues, is possible to use that on some occasion or by applying together with different data structures.

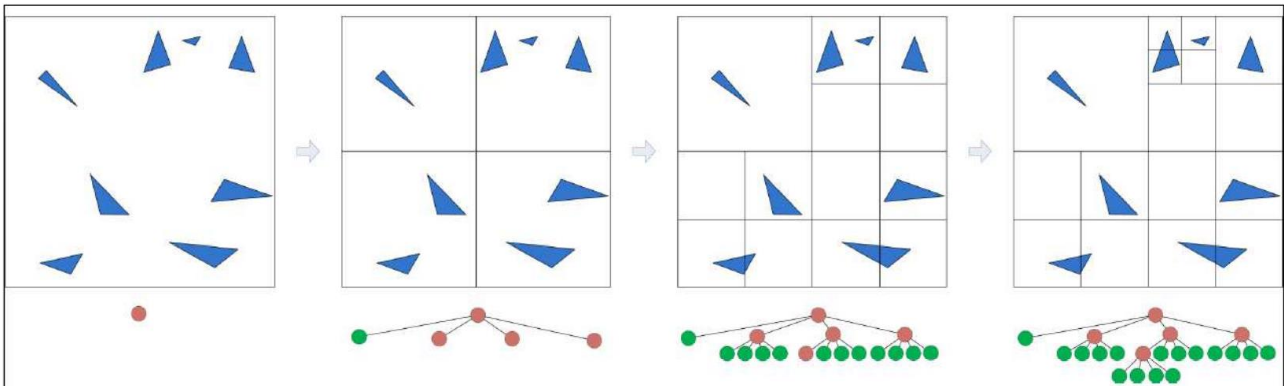
Octrees

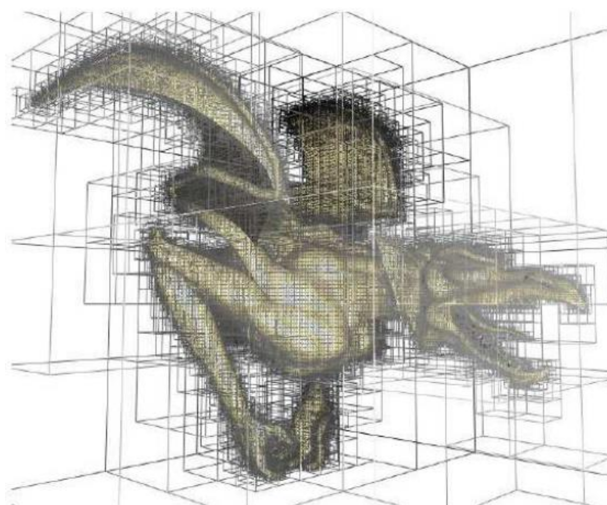
It is a tree structure, and it is based on the recursive partition of the 3D world; it is made by 8 trees.

We start from a Bounding Box of the whole scene, then it divides the world in 8 cubes (or **octants**), then I'll dived again some of these octants in other octants but not all of these "big chunk" (or starting octants) only someone who hit a certain number of objects.



Then I iteratively keep subdividing in this way, the problem is when one object got subdivided by an octree, an option is that so you can assign that object to two nodes.





Binary Space Partitioning Trees

The point is that the dimension of the cells which is uniform in the octree, because I divide exactly in the half of the octant, and obviously this solution can have issue if we have objects non uniformly placed in the scene.

The **BSP Trees** is a data structure which is more adaptive, the idea is that we divide the space in two not in eight, we decide an axis and we divide in two parts respect the axis, even the position of the plane is not perfectly in half it is based on the content of the scene.

This is surely more complex, because you have to analyze the scene to understand where you have to cut but is more effective, we will talk about the **Axis-Aligned BSP**.

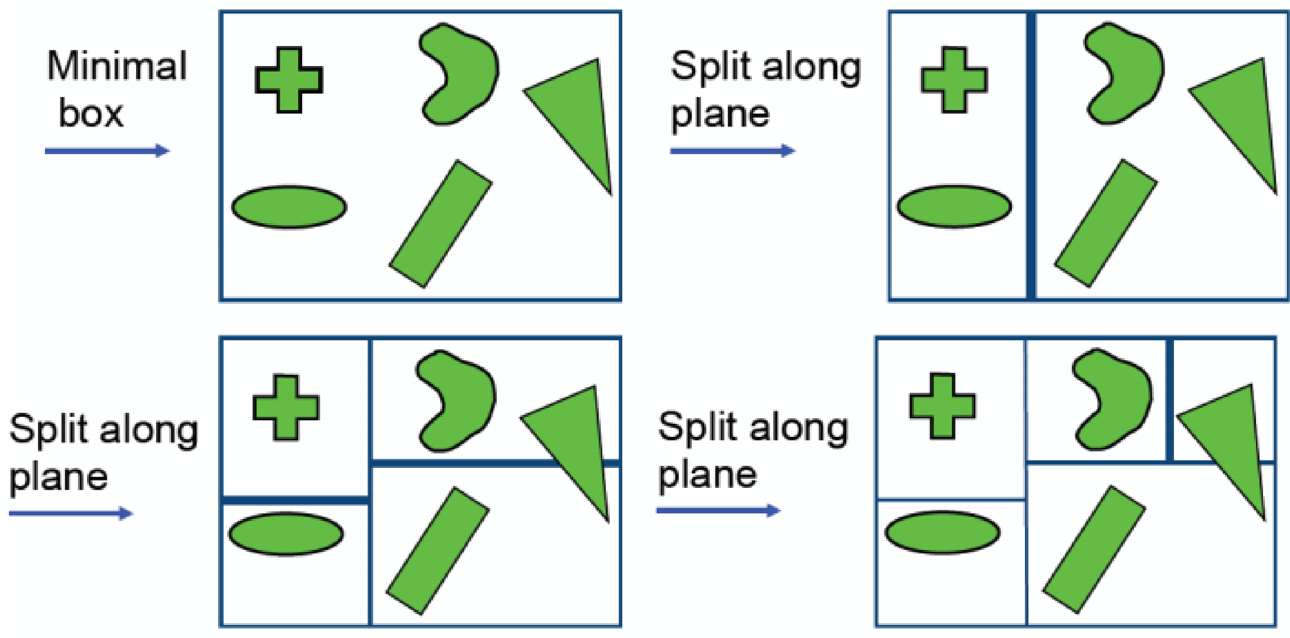
Axis-Aligned BSP

Mainly it is a **kd-tree** (where k=2, so a two-dimensional space)

I have to determine :

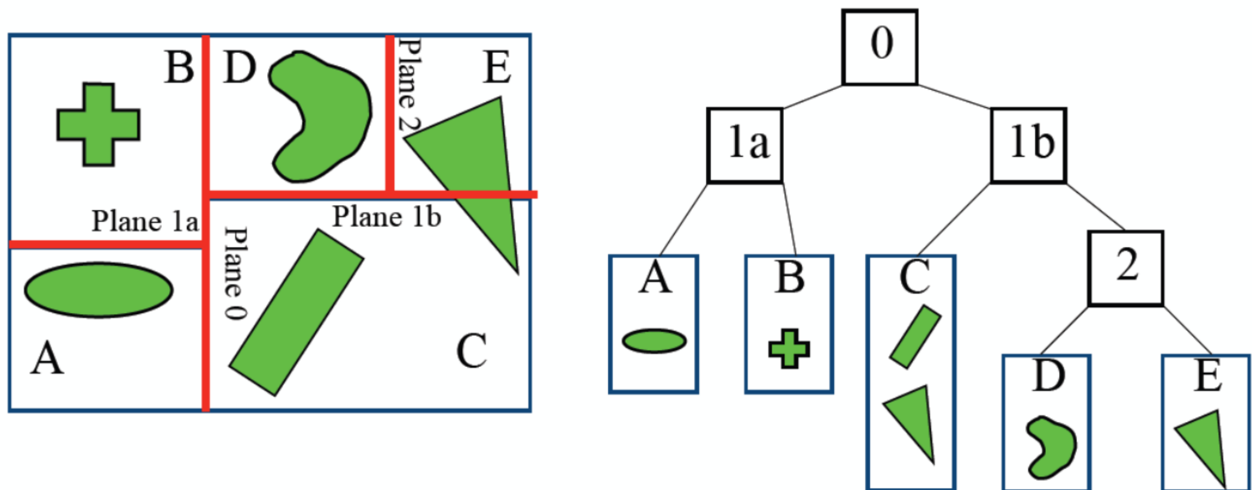
- The axis to subdivide
- The subdivision planes

After this I evaluate the composition of the two subspaces, this will be repeated recursively until termination criteria are satisfied. The main idea is to separate less dense area, which I can avoid subdividing.



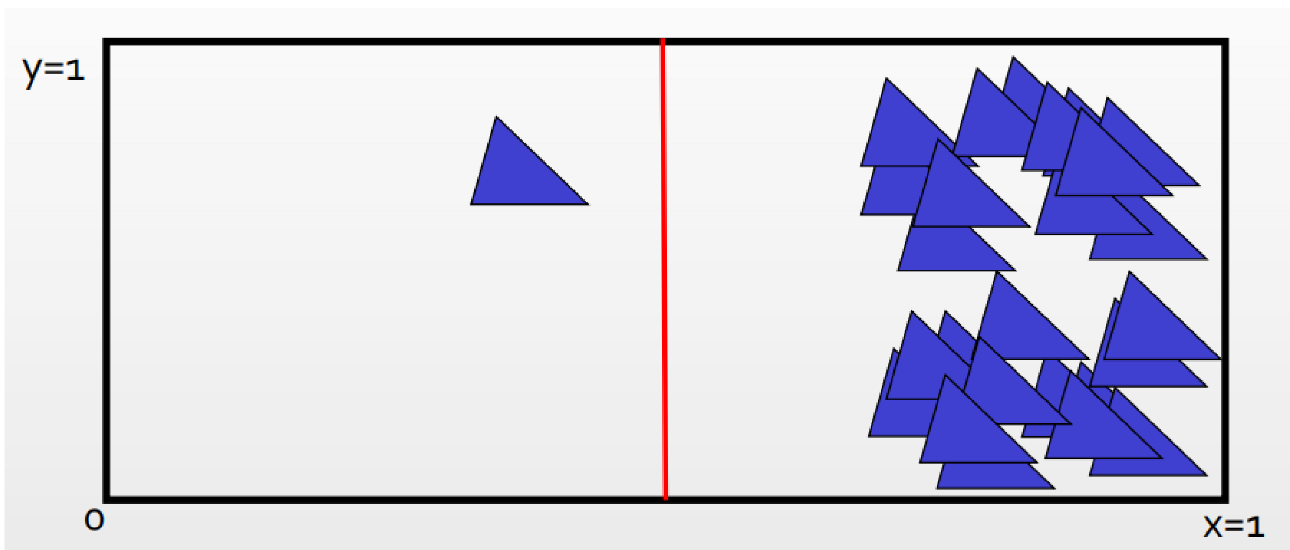
There are issues when I have polygons crossed by the subdivision plane (like octrees), it is not a simple decision what to do, it depends on a determinate situation :

- I can assign one node only, but how which ?
- I can assign that to both nodes, but this bring a repetition in the tree.

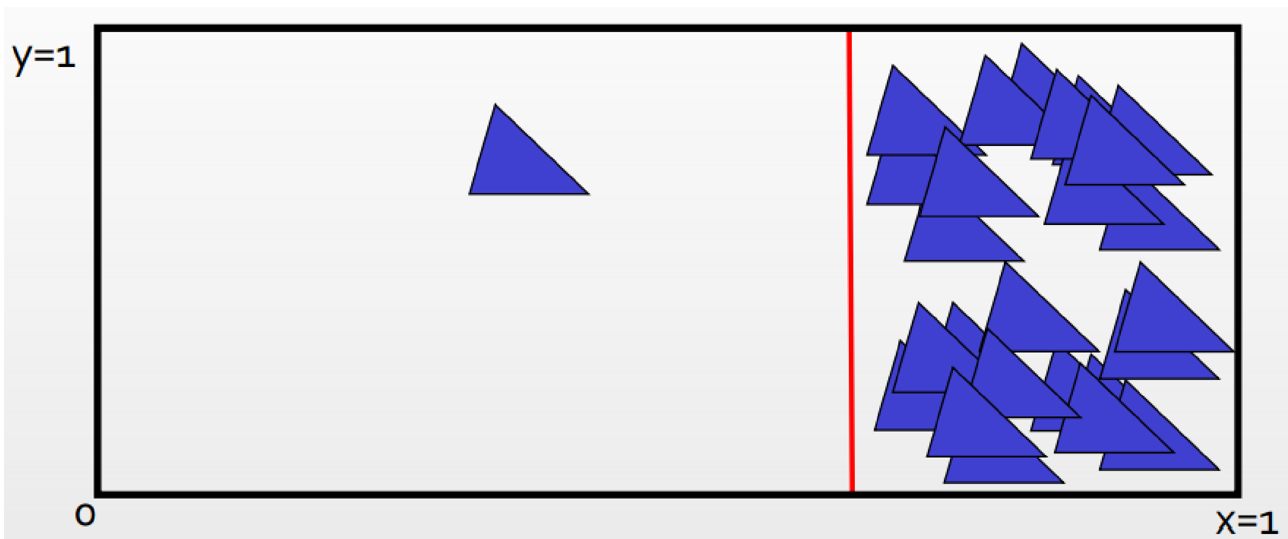


The main characteristic of the BSP-Trees is that I decide where to cut, because we have not a fixed subdivision (like octrees).

In this extreme example, if I divide in the middle I'm in an octree situation



The correct approach is to have an optimized split, just at the beginning of the group of triangles on the right, in order to have the right area very tight without not too many empty areas in the volume.



How it is possible to decide this ?

I have to apply some heuristic algorithms which analyze the size of the scene in order to optimize the subdivision.

Scene Graph

Spatial Data Structure mainly are made of trees or graph giving a spatial organization of the model. There is also another graph data structure which is the **Scene Graph** (very common) it is more **logical organization** of the scene than spatial; it considers all the features involved in rendering.

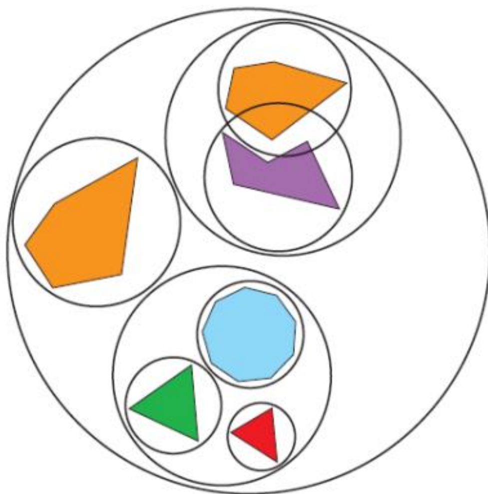
- Transformations
- Materials
- Textures
- LoDs

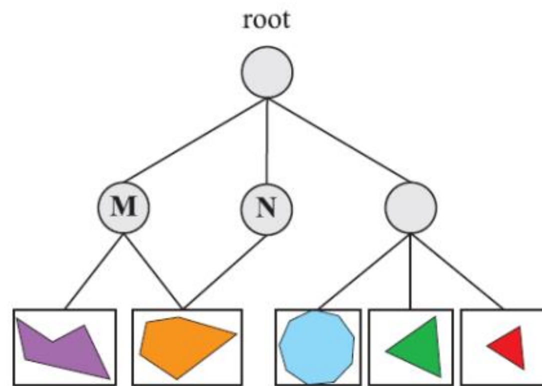
It is not creating for optimization or acceleration, this is a **logical** description of the logical relationships between objects in the complex scene, for providing information to the rendering process of the game engine. Each node of the graph could be a light, model, camera, … the connection between nodes results in a **relationship** between these elements.

The use of the graph for the spatial data structure is totally different from the use of the Scene Graph (it won't speed up anything).

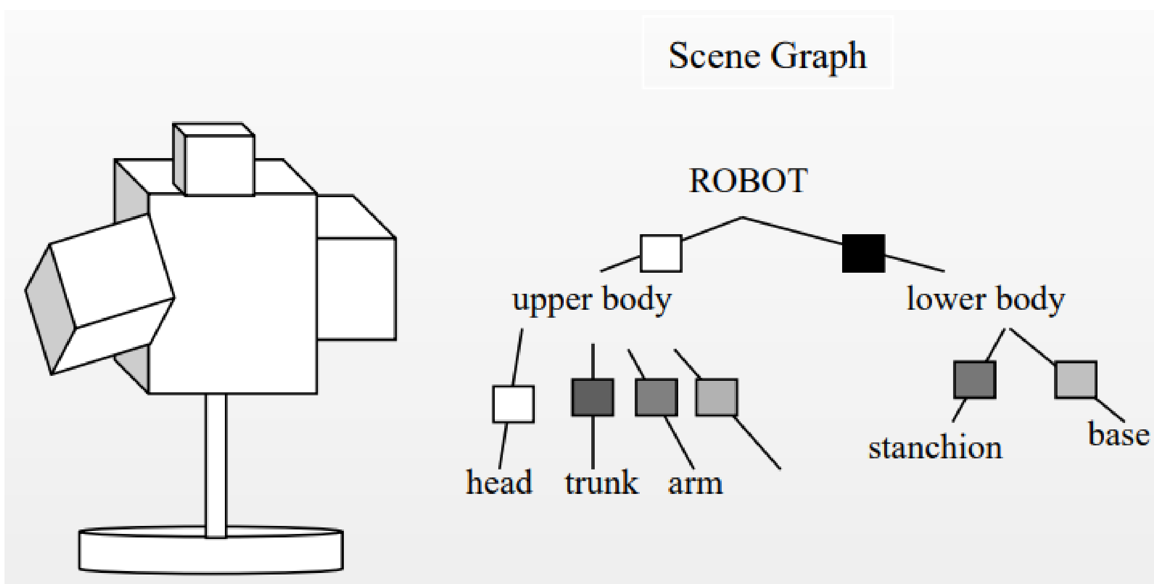
The edges (so the connection as I said before) represents a relationship between the elements (nodes), this will create a hierarchy.

Several nodes can be connected to the same **child node**, this will result in a **DAG.**

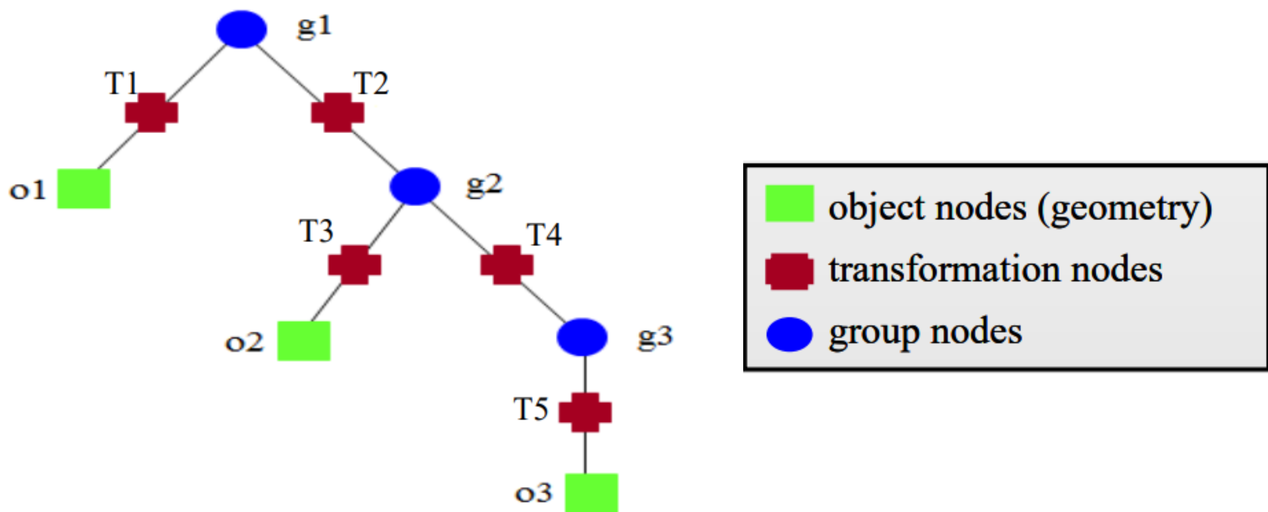




We have a hierarchy of the bounding sphere but in that case it is a tree, **M** and **N** are two transformations where **M** is applied to both purple and orange object where **N** only on orange, the leaves are representing the **main model** of the object that will be transformed.



There is the scene graph representing a hierarchy for an articulated model of a robot, where you have two sub-trees for the upper body, lower body and subdivision for both the part.



Often you don't see a tree-structure but consider that **transformations** in the **Scene Graph** sometimes are seen as **nodes** instead **of edges**, the connection between this nodes and **model nodes** means that the transformation is applied to the object (or more the one).

In the case that the model has child nodes, this means that the transformation is applied to the subtrees, in this way i can represent a **CTM**, **Cumulative Transformation Matrix**

Object o1 -> CTM = T1

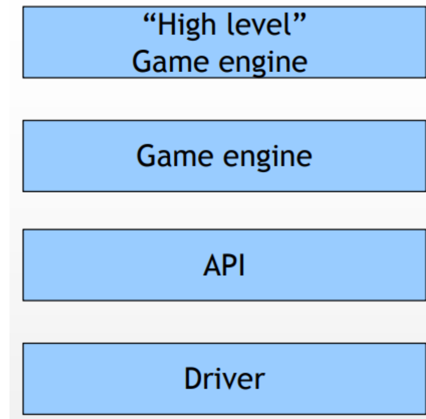
Object o2 -> CTM = T2T3

Object o3 -> CTM = T2T4T5

The Scene Graph helps understanding the order of transformations of all the different parts of the subtree.

API for Graphics Programming

Mainly when we talk about **Graphics Programming** we have different level of abstraction we can use, from low to high.



- The lower level is hold from **drivers**, where a set of **routines** are mapped to the **GPU**, usually these drivers are provided by the hardware **manufacturers,** and they are hardware dependent and from the computer it is using.

- The **API** for **Graphics Programming** are living on top of the drivers, so they are at a higher level of Drivers and they provides a set of procedures used only for **graphics**, so this type of procedures will get translated in driver instructions. We usually rely on some API for communicating to the **GPU** through the **drivers**. API are at a certain low-level of the possible tools and framework for graphics programming.

- **Game engines** are something that are at more **higher-level** and easier to use for graphics programming. They are a tool for the development of programs, so it consists of more tools inside a one. Usually, the abstraction of a Game Engine is higher of an API, because the aim of this one is to give reusable tools to the developer for creating the game.

  There is two kind of game engine :

  - "**Higher level**" **Game Engine**, this is the situation right now, the main choice is Unreal and Unity. This has an impact in particular in the past years, relating to the professional figures in graphics programming. In the sense that actually we have really a separation between the developers using the game engines to develop the final games and developing the game engine by using other tools.

  - **Game Engine** (**low level**), which doesn't have any kind of **GUI** (for procedural games). There is not lot of kind of this game engines like Ogre3D, Panda3D, Love2D or some proprietary engines developed by studios for internal use only.

There are also other **visual tools** for **multimedia** production used, but not for videogames but for interactive application like art, museums and so on… usually they are node-based application with a limited amount of assets.

Direct3D

Is the **API** proposed by **Microsoft** it is part of the **DirectX** which is actually a **set** of **API** for different multimedia app. Obviously you can develop application **only for Windows** using this one, it was created explicitly for the development of videogames on **PC** and **XBOX**.

Historically the are lot of discussions, in particular the first version but from the **version 9** the discussions received a **better feedback**.

- The current version is **Direct3D 12**, at the moment this is more similar to **Vulkan** that then to OpenGL, before was more similar to **OpenGL**.

Direct3D 12 is closer to the hardware, lower **level** of **abstraction** (less easy to use), this for have a better control of the multithreading and for decrease the CPU utilization during games.

Direct3D has its own shading language which is **HLSL**.

OpenGL

The main difference is that **OpenGL** is not an **API**, it is a **specification**. The **Khronos Group**, which is a **consortium** of different entities which discuss and produce rules, once the rules are approved there will be **released** a **different specification** of the **API** (*so on Linux the OpenGL is a different specification from the one on Windows*). This is why it stands "specification", because we are using a specific version of the library.

OpenGL was born in **C**, but then in the years we have binding with several languages. OpenGL was not born for videogames but for **industrial applications** (graphic workstations), as an evolution called **IrisGL** (an API developed by **Silicon Graphics**, **SGI**).

The **shading language** for OpenGL is **GLSL**.

Direct3D vs OpenGL

Until **Direct3D 11** we could say that this sentence/comparison is completely true, but right now with **Direct3D 12** the comparison moved to **Vulkan**.

The history of the two API was completely different, which the first is owned from **Microsoft**, from this point of view it had a more **regular evolution**.

OpenGL Evolution

The **OpenGL** evolution of the specification was more difficult, because was given from the discussion from different partners. The idea is that we had the specification and then every manufacturer developed their own **extension** (the extensions are an actual hardware piece!) to **OpenGL** on hardware, and there was some mechanism to access this extension, the **issue** is that we have some **code** which looks **cross-platform** but executed on another architecture will give an **error** because the GPU doesn't **support** that extension.

- The first version, **OpenGL 1.0** was produced in **1994**, and until **2004** we had a **fixed pipeline** (I couldn't develop anything, no shader support just a **black box**).

- **OpenGL 2.0** introduced **programmable shaders**, so we could decide how to process the vertices and how to create and compute the final pixels on screen. However, that wasn't the only way to approach to the **GPU** and graphics programming :

    - **Fixed pipeline**
    - **Programmable shader pipeline**

This because a straight switch to the programmable shaders pipeline would have made unusable all the older code of the fixed pipeline, so the **fixed pipeline** was still present.

- **OpenGL 3.0** introduced some specific mechanism in order to **remove** the functionalities. This because the **previous situation** was a **mess**, too many things to maintain, so they decided to simplify the OpenGL pipeline and still trying to maintain the compatibility with the previous release version (but not completely compatibility with first versions).

  They introduced data structures called ***context***, this context will describe the overall approach to the use of OpenGL 3.0 (contains the shaders and other data).
    - **Full Context**, which means that includes everything. A code which is compiled, and it express the uses of the full context compiles because all the functionalities of all the current specification and also the one deprecated from the previous version.
    - **Forward Compatible Context**, in this case the code will compile only if calls for the current specification are done.

  The idea is to distinguish between the **modern** code and old code from previous version (partially adapted, but still rely on deprecated calls).

- **OpenGL 3.1**, 24 March 2009, they decided to **remove all** the **deprecated functionalities**, they analyzed the most part of the code and it was already updated enough to **truncate** the **backward compatibility**.

  From this version the **fixed pipeline** died, and the **programmable pipeline** with **GLSL** was the new standard. This means that the applications must use shaders, and that the data of the mesh must be read from the **GPU memory**, by using the *buffer objects* (not more instant methods to read the mesh from the main memory).

  They introduced a specific method to access the removed functions, this was the OpenGL ARB.

- **OpenGL 3.2**, 03 August 2009, introduced the **Geometry Shader** and introduced **Profiles**, it is another level of sophistication in the definition of the state.

  They have **spitted** each ***context*** in **two profiles**.

  Full Context :

    - **Core profile**, it includes all functionalities of **current specification** deprecated and not deprecated.
    - **Compatible profile**, it includes all the functionalities of all the **previous** and **current** specification.

  Forward Compatible Context
    - **Core Profile**, it includes all functionalities of current specification
    - **Compatible Profile**, not supported (has no sense).

- OpenGL 4.1 introduced two new stages of Tessellation Shader

The current specification is the **OpenGL 4.6**.

In the last years we had a fork of OpenGL

- **OpenGL ES**, which it is used for **embedded system** (used for mobiles). Current specification 3.2, August 2015, the pipeline of OpenGL is mainly based on OpenGL 3.3, so it is a programmable pipeline which uses **GLSL ES** (is a subset of GLSL language).

- **webGL**, it is the OpenGL ES for the web, for years there was different solution that simulates the 3D rendering in real time for browser, but the main issues in the past was given by the dependency of plug-ins to render the graphics.

  Mainly it is a **Javascript API**, and it uses element from **HTML5** like the **Canvas** and **DOM** (*Document Object Model*), this means that it is natively supported by modern browsers. It is really more recent project, the first specification was proposed in March 2011 which was using the **OpenGL ES 2.0**.

  The current specification is the **WebGL 2.0** released on 2017 January, which is based on the **OpenGL ES 3.0** .

Vulkan

Has been considered the successor of **OpenGL**, released from **Khronos Group** at **GDC 2015**. It is based on **Mantle API** proposed from **AMD**, which is a new generation of API, then they decided to open the specification and the Khronos Group decided to use it as basis for Vulkan.

The **NVDIA** published the first driver with Vulkan **support** on 7 March 2016, the current specification is **1.2.172** released on 8 March 2021.

Main features :

- **Lower-level** API, much more then OpenGL, this gives **more control** on different aspects.
- **Optimize** the **balance** between **GPU** and **CPU** usage.
- More **control** on **multithreading**
- **Better** scaling on multi-core **CPUs**
- **Native multi-GPU supports**, like **SLI** or **Crossfire**, the idea is to move something that run on more heterogeneous system so with different GPU models and still having the same power of computation.
- **Intermediate binary format** for **shaders**, the **SPIR-V**, the shaders in OpenGL are compiled at run-time by the application. With Vulkan we moved to a different approach, which will precompile the shader in **SPIR-V** and then we load this intermediate at run-time without having to compile (much faster, this will give initialization speed-up, and a large number of shaders can be applied on the SPIR-V file).

Vulkan vs OpenGL

Vulkan is probably the future, it is more suited to the right now hardware. OpenGL started in a difficult way and can't reach what Vulkan is going to reach. The problem with Vulkan is given by the fact it is **really hard** and at a **lower level** then OpenGL, much more lines of code just for setting up the application. It can go to a really sophisticated level for managing the multithreading of the application.

Metal and fragmentation issue

Metal is the API proposed from **Apple** for **iOS** and **macOS**, it is mainly an API that has features:

- **Low CPU** overhead.
- **GPU** performance optimization.
- Shared **CPU** and **GPU** memory.
- **Pre-compiled shaders** (like Vulkan).

Now the choice of Metal by Apple as lead to a **fragmented situation**, in the sense that Apple as decided to not adopt **Vulkan natively** and so even if now it has been granted by the **moltenVK library** (an additional library between the macOS and Vulkan, released on February 26th 2018) the point is that it has created a **singularity** in the environment.

Even because the **OpenGL** support has **not been updated**, so it is still at the version 4.1, this creates a difficulty in the development of different game engines.

The fragmented situation

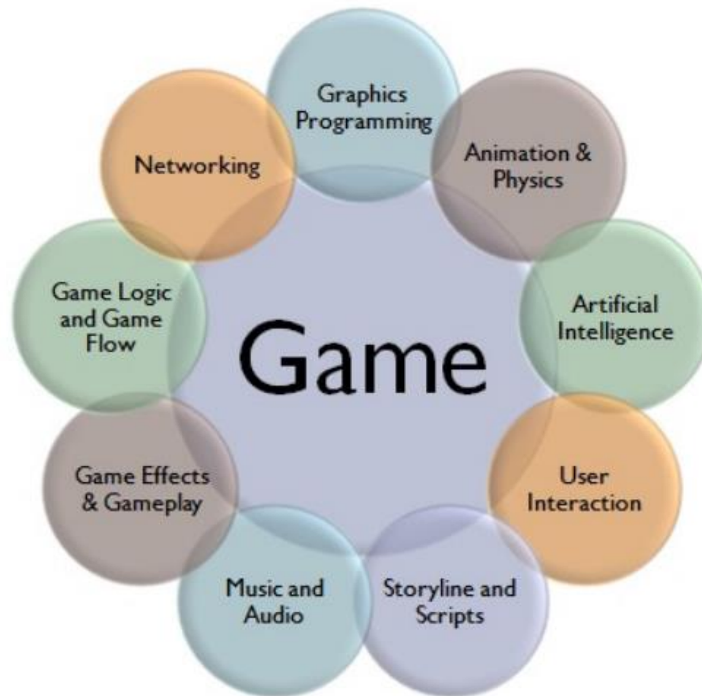
	Windows	Linux	macOS	iOS	Android
Direct3D	x				
OpenGL	x	x	x*	x^	x°
Vulkan	x	x	x#	x#	x
Metal			x	x	

* Full 4.1 specification with "something" from 4.2
^ through OpenGL ES (3.0 specification)
° through OpenGL ES
\# through MoltenVK

Limitations of using APIs for Graphics Programming

The Graphics API provide only the graphics rendering, this means that every other functionality must be implemented separately (external libraries for I/O, Window Management, Audio, …).

In the developing of video games\interactive applications, several other disciplines must be considered.

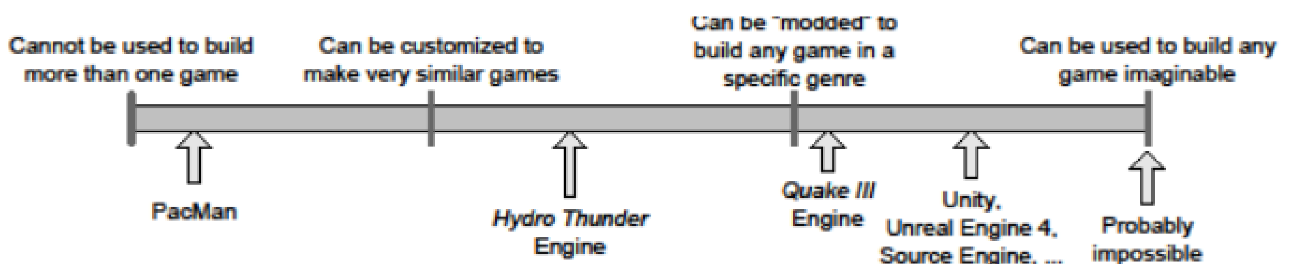


A **Game Engine** is a **high-level** support for development, which is independent from a particular **API**. Composed by a set of sub-engines for the development and management of resources. A Game Engine is identified by three main keywords : *Modularity, Flexibility, Reusability*.

Game Engine – Reusability

The Game Engine architecture is a Data-driven architecture, means there is present a big split between assets and software components. There is a possibility to reuse the same modules for different applications with different contents.

This is not always achievable, not all game engine has been designed with a high level of separation. Each game genre has different peculiarities, some genre has a higher level of reusability for a specific genre than another.



Game developer vs Engine developer

In recent years there is a split between developers, the one who develops the Engine (**Engine Developer**, develops all the wrappers for all APIs, for every target and development platforms) and the one who develops the Game (**Game Developer**, uses the game engine and its tools to develop a game).

Culling and Clipping

Cull is the act to remove elements from the rendering pipeline elements which are not useful for the creation of the **frame** (this will speed up the creation of the frames).

- **Visibility Culling** is a set of technique that prevent **meshes** to be **sent** to **GPU** (the fastest element sent to the GPU is the one that I don't send), so happens on CPU side.
- The **Clipping** it is done on the **GPU**, it is the process of discarding the primitives which goes out of the defined window region.

During **Physics Simulation**, **AI** and **Collision Detection** they must consider elements even if they **are not visible**, so we have to distinguish among not sending an object to the **GPU** because is not visible and **eliminating totally** the object from the state of the application because is not visible.

<u>This are techniques that avoid the GPU to do a partial computation or complete.</u>

The techniques aren't performed in the same point, they differ the location between **CPU** and **GPU**.

- The **View Frustum Culling**, **Portal Culling** and **Occlusion Culling** is done on the **<u>CPU</u>**. The **Clipping** is done on the **<u>GPU</u>** , on a specific stage **after** the **projection**. There is a culling operation that it is available to use on the GPU it is the **back face culling** it is applied in the Vertex Processing stages but it isn't the first to be performed. The GPU culling techniques doesn't allow the full control (the CPU yes!), they are **enable/disable** only.

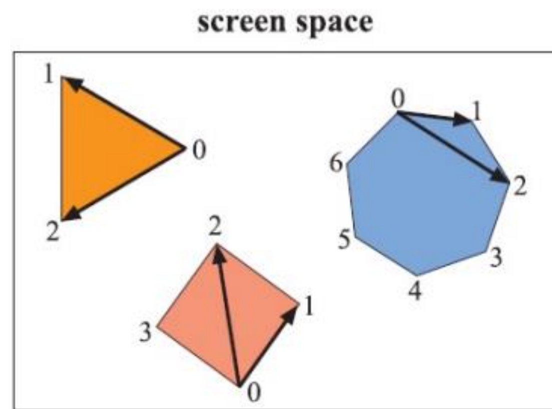
<u>VISIBILITY CULLING SET</u>

Proceeding in order of complexity (they are applied in different order inside the pipeline).

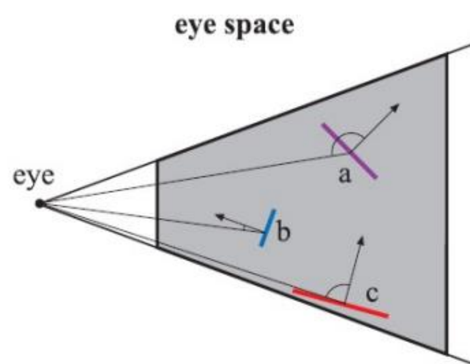
Back face Culling

This is implemented on the **GPU** sides and through the **Graphics API** I can only **enable** or **disable** it. If we have an object, composed by **faces**, each face is composed by a **front face** and **back face**, the one has a **positive direction** of the **normal** (**vice versa** for the **back face**). If an object is **closed** I **don't see** the **back face**, but actually the pipeline is processing both the faces of my mesh, so if I activate the back face culling I can tell the **<u>GPU</u>** to discard the computation of these faces and saving the resources.

There are two kinds of tests, the first one is done in **screen space**, after I converted the view-frustum in the unit-cube, I got my projected vertices, then we compute the **normal** of the **projected polygon** and then comparing that with the positive z-axis (by using the dot product, it will give me a 3D direction) of the world (this will change with the implementation of the world coordinate system).



The other test is done in **eye space**, this means that all my vertices are expressed in **view coordinates** (3D vertices), so we have done the view but still not projection. In the **view volume** will happen the computation of the vector from a polygon point to viewer. After that there will be the computation of the dot product with the normal previously calculated, in this way we can check if the angle is greater than 90° (back face).



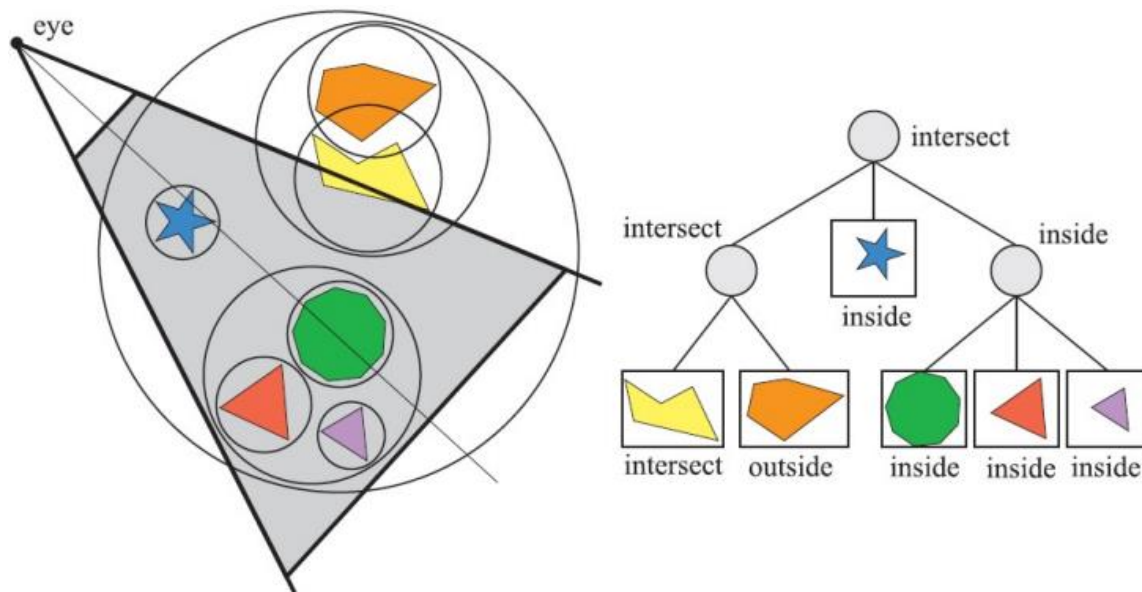
Hierarchical View frustum Culling

This operation is done on the CPU side, so on the Application Stage of the graphics pipeline. The main thing that happens is that if an object is outside the **view-volume** of this **frustum** then it is cut away.

Anyway, this comparison is done between the Bounding Volume of each object and the frustum.

- If the **Bounding Volume** is **totally outside**, this means that the object represented by the Bunding Volume is surely outside. Then it won't be sent anything to the Geometry Stage.
- It the **Bounding Volume** is **inside** or **partially inside**, then the object represented by the Bounding Volume will be sent to Geometry Stage to be rendered.

By exploiting the **Bounding Volume Hierarchy** is possible to enhance an optimization of this operation (like maybe avoiding sending multiple objects to the GPU), by comparing each Bounding Volume with the **frustum**.



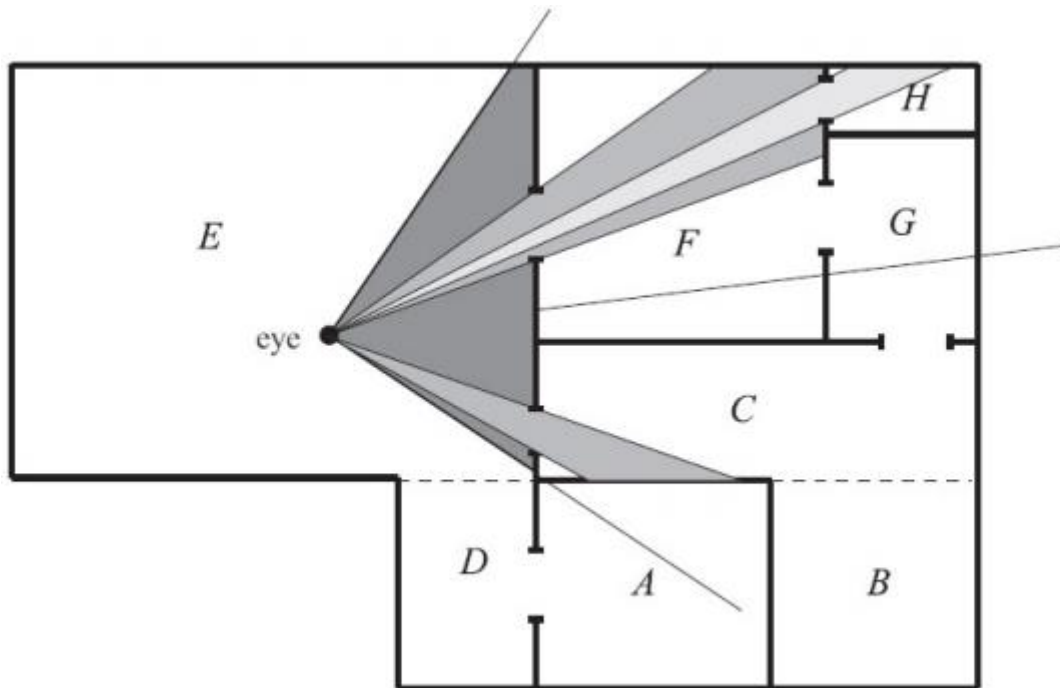
There is a particular intersection test done between **Bounding Volume** and **frustum planes**, will be seen in the future.

Portal Culling

This technique is applied when we have complex **indoor scenes** (inside a building with lot of rooms connected by doors), because in this the **view-frustum** culling is spanning completely the room, it is a situation similar to the Occlusion Culling, the idea is that usually I have the wall and I have some door or windows that allows me to see in the next or closer environment but not all the environments.

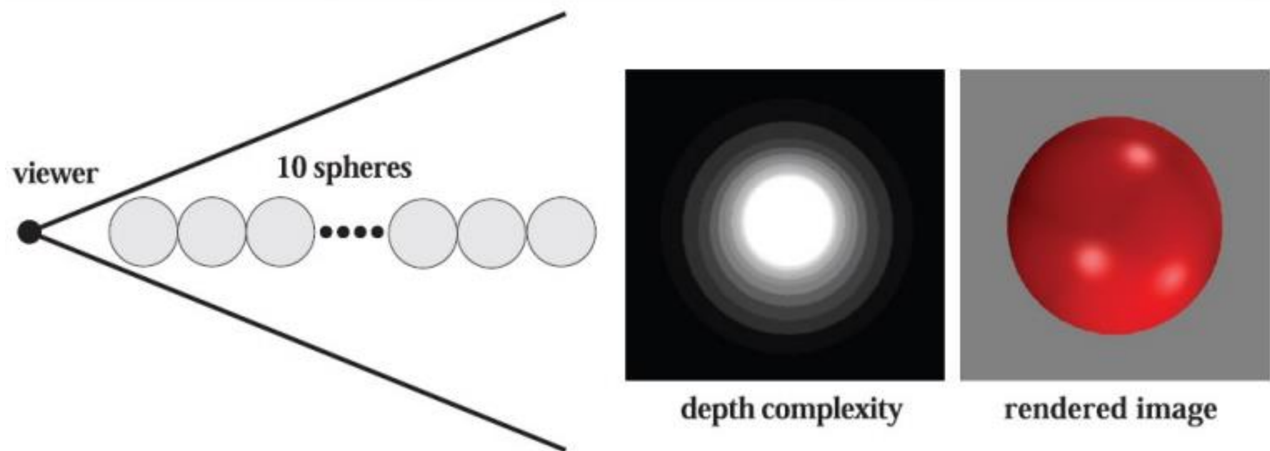
We consider the doors or windows as **portals**, and rooms are subdivided in **cells**, this is a complex data structure often build ad-hoc for each scene. Using the **position of the viewer**, in the direction where him is looking we can cull away a lot of geometry.

In the schema below we saw the larger **FOV** (which is the view-frustum) and we can **cull away** lot of geometry from the rooms, we can do several calls to the View Frustum Culling Techniques by changing the view frustum FOV to **making fit** the **angle** of different portals of the door. The lighter greys which pass to the door, doing that we can send to the **GPU** the **part of the room** that we can **show** from the door and discard the rest.



Occlusion Culling

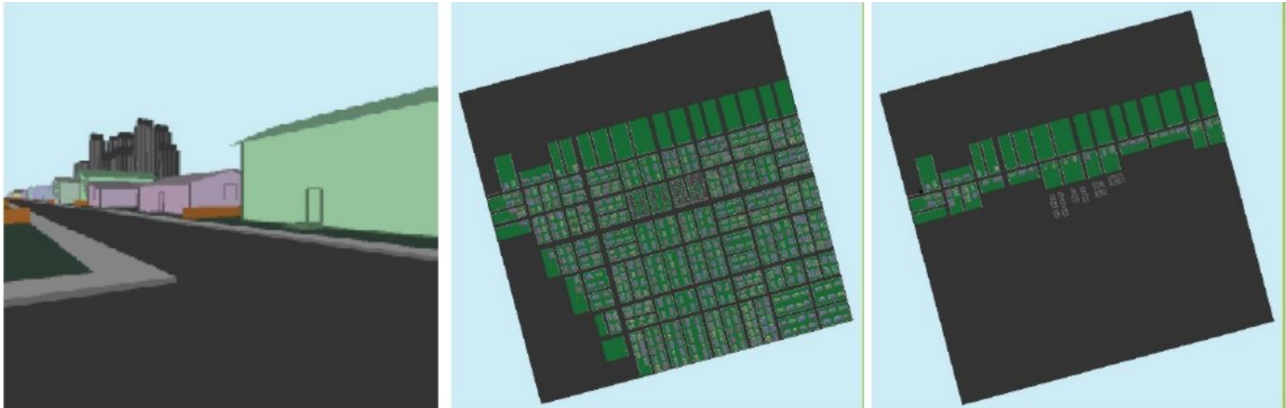
This is the most complicated technique; I may have an object which is inside the **view-frustum,** but it is **occluded** by another object in front of it. This is a technique that tries to understand if an object is covering the view of another because is not visible in the ending frame.



The is that if we just rely on **View Frustum Culling**,(which is easy to perform) we for sure gain performance but after that we don't have any other control until the **Z-buffer test**, which will solve the **visibility problem**. We may apply another technique to avoid this useless processing of geometry earlier (in the CPU side), this is the **Occlusion Culling.** This maybe an option for large cities or indoor scenes, with lot of buildings that occludes, if you look at near you only see the nearest buildings but not the others which are loaded anyway in the GPU (with Occlusion Culling disabled of course).

The advantage of having another **Z-buffer** algorithm earlier on the CPU side is that I can avoid sending to the GPU the geometry that would be processed by the **Z-buffer test** anyway, and as we know the fastest thing that the GPU can compute is the one which isn't sent to it.

The **View Frustum Culling** has **no idea** about **walls** that are occluding the objects, so the **Occlusion Culling** will discard the occluded objects inside the View Frustum.



The second image show what the **View Frustum** loads, the third show what is getting computed on the **GPU** with the occlusion culling enabled.

These are really effective techniques, but not really easy to apply it and not really easy to understand it.

We can do the test in the **image space** or in **object space** :

- **Image space**, I do some projection in order to have the vertices projected on 2D plane and I do a visibility testing in 2D.
- **Object space**, this means that I test visibility in 3D in world coordinate.

These are techniques which benefits from some sort of front to back sorting like using **BST** or **Spatial Data Structures.** Because even a small object closer to the camera can occlude lot of objects, so with front to back order we can check first the objects to the camera, because that object is the one which will occlude more inside the **view-volume.**

Occlusion Culling is mainly done on the **CPU**, but more recently there are examples of Occlusion Culling implementation on the **GPU**, the idea is to do :

1. **First render** of the **GPU** simplified , with just the bounding boxes (query on visibility set).
2. If they are in the View Frustum then they are **visible**, but **potentially occluded** by **others**, so we sent it to the GPU.
   a. We process the one sent to the GPU on the Vertex Processor by applying the transformation and projections on the Bounding Box (very limited number of vertices).
   b. We rasterize them (no fancy lighting)
   c. Depth test
   d. We count the number of pixels which passes the Z-buffer test, if this number is equal to 0 this means that the bounding volume is occluded, in the real render pass we won't send them else if the **number of pixels > 0** we can consider a threshold of fragments visible.
3. If **outside** we discard them.

On pipeline cycle for render the bounding volumes of the objects for checking if the bounding volumes are visible or not, then to the second I will send only the meshes of the visible bounding volumes (which have passed the depth test on the first pass).

It is a knife with two sides, these techniques use **two GPU passes** for one **frame rendering**, if everything is setup well and **the test succeeded then we gain performance**.

If the **test fails**, **we lose performance**, because if we apply the Occlusion Culling in a situation where aren't present lot of occludes, the **first pass** will be **useless**, because we will pass the mesh to the GPU in the second pass anyway.

It is not a technique which can be used in any situation, is something that must be considered and evaluated considering the scene we have to render (so when we have lot of occludes, large cities, indoor scenes, …).

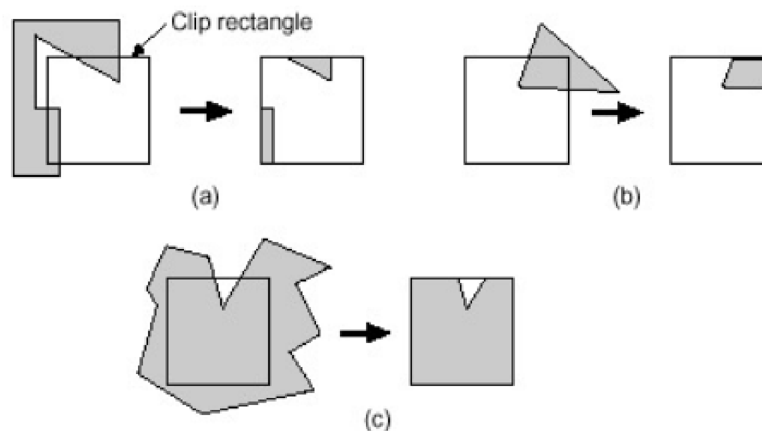
There are different proposals to optimize this process :

- The use of hierarchical structures like the BHV or octree..
- Optimizations to avoid CPU stalls while waiting the query response. Because the problem is that the second cycle of the pipeline stage, in the Application Stage based on the result of the first cycle must pass the visible meshes to the GPU.

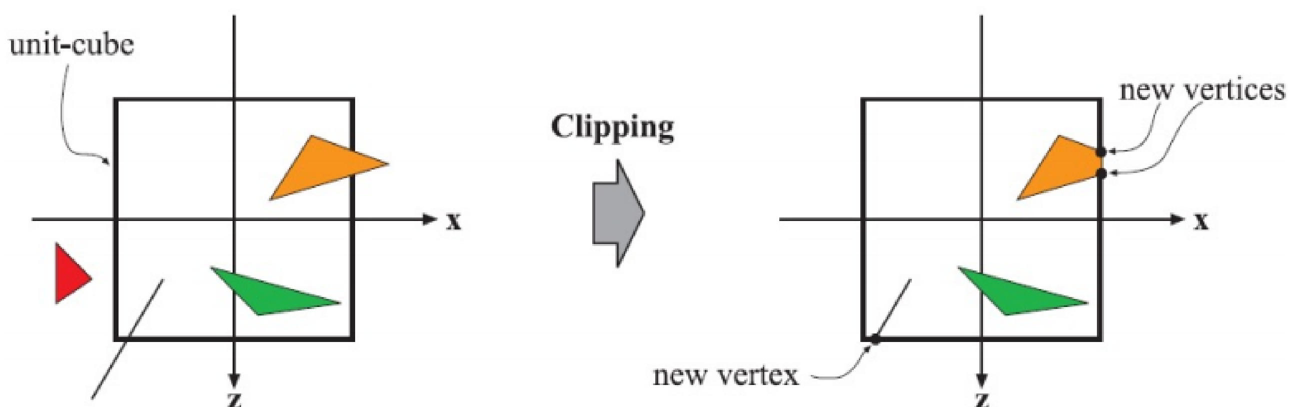
  In this way the CPU must wait until the first cycle (which will render the Bounding Volume or the Convex Hull!) is performed. To avoid the CPU stalling we can use additional techniques, like the exploiting of **temporal coherence** or using the **query queue**.

Clipping

Clipping it isn't actually a **Culling Technique**, it is applied after the projection, and it discards the **primitives** which are **not visible** inside the **projection volume** (Canonical View Volume).



This means to actually to **create new vertices** and **edges** in order to keep only the necessary information.



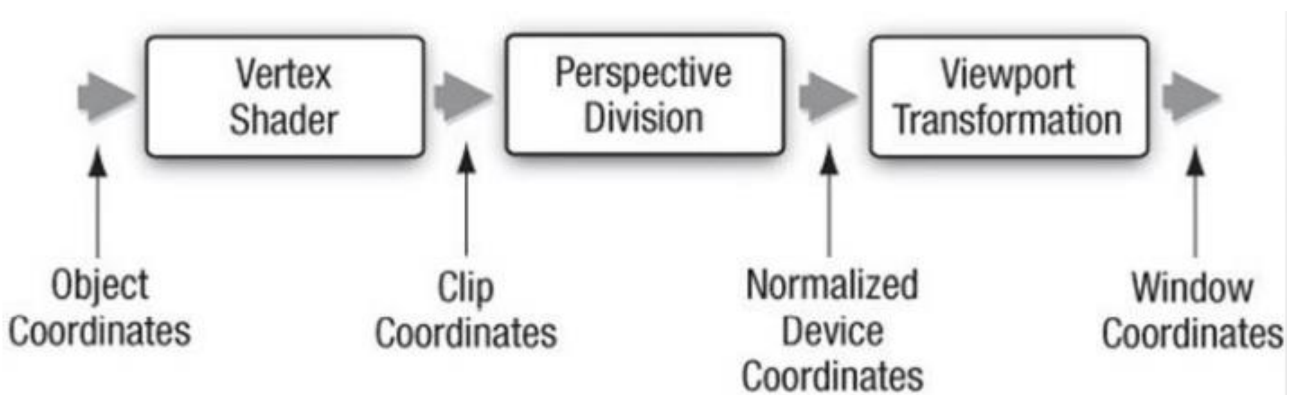
Going back to the **Projection Matrix**, then one which transforms the **View Volume** of the **frustum** in the **Canonical View Volume**.

If we apply the projection matrix to a point of the coordinate system we should get a perfect transformation, but this isn't actually true because we get the $w = -z$ instead of $w = 1$.

$$\begin{bmatrix} \frac{c}{a} & 0 & 0 & 0 \\ 0 & c & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{c}{a}x \\ cy \\ -\frac{f+n}{f-n}z - \frac{2fn}{f-n} \\ -z \end{bmatrix}$$

There is another fundamental passage called **Perspective Divide**, so we divide all the vector by the content of w ($-z$ or z depends on the coordinates), this will give us the projected vertices we want to keep.

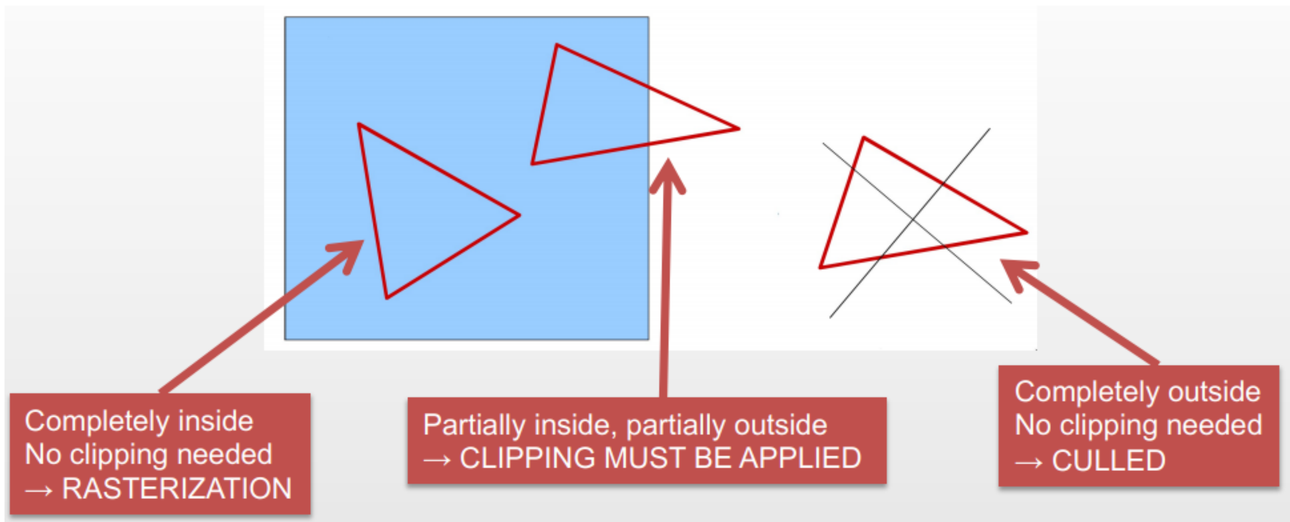
At this point, the things are a bit more complicated for **clipping**, because actually the clipping will occur earlier then **perspective division** which will give the **clip coordinates**, after that we apply the **perspective division,** and we will have the **NDC.**



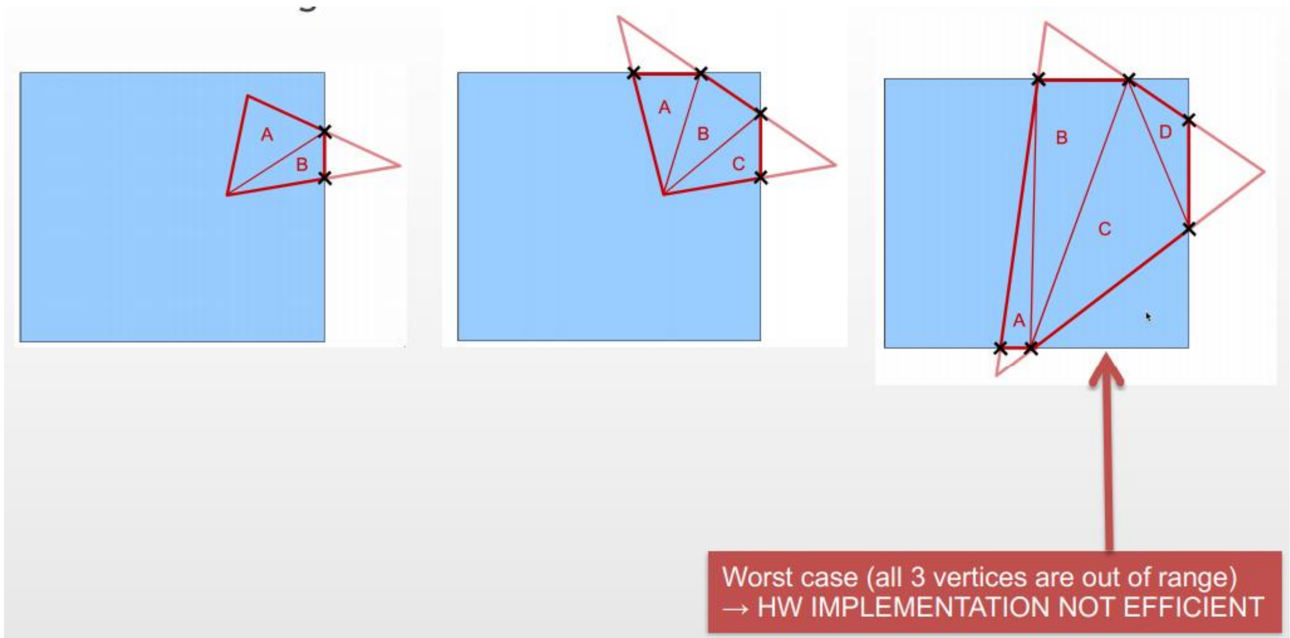
1. The **projection matrix** will express to us what we want to shear by using the transformation, this will give us the **Clip Coordinates** expressed in a range $[-w', w']$, with the $w' > 1$ (w of the vector that got multiplied is equal to 1).
2. **Clipping** of outside view volume primitives.
3. **Perspective Division** which will take the **Canonical View Volume** coordinates (**clipping coordinates**) and normalize them inside the range $[-1, 1]$, now they are the **Normalized Device Coordinates**.
4. **Rasterization**

Usually, these passages are simplified in books, because we don't have control, they just saying that the clipping is described as if applied to NDC, which is wrong!

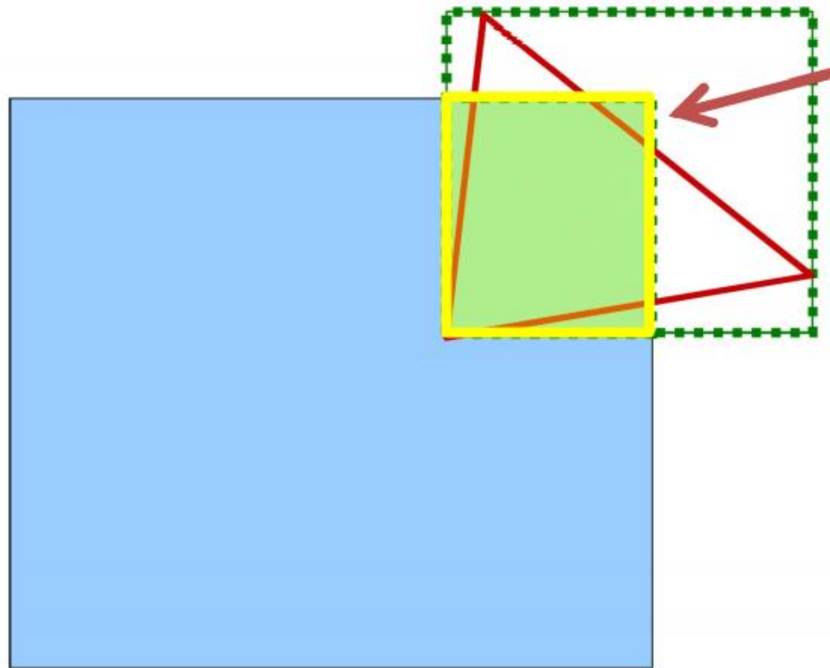
It is important to distinguish between **clipping coordinates** and **clipping**, the first one is the output from the Vertex Shader (usually after the projection matrix), the second one is the operation of cutting out the not visible primitives.



In the past the approach (obsolete method) the idea was to find the **intersection between** the **side** of the **cube** and the **polygon** and then to **create** new **vertices subdividing** new **triangles** and then **rasterize** the **triangles**.



In the current approach, we do a **Bounding Box** around the **primitive** and then we **intersect** between the **view volume** and the **Bounding Box**, we **<u>keep only the intersection</u>**.

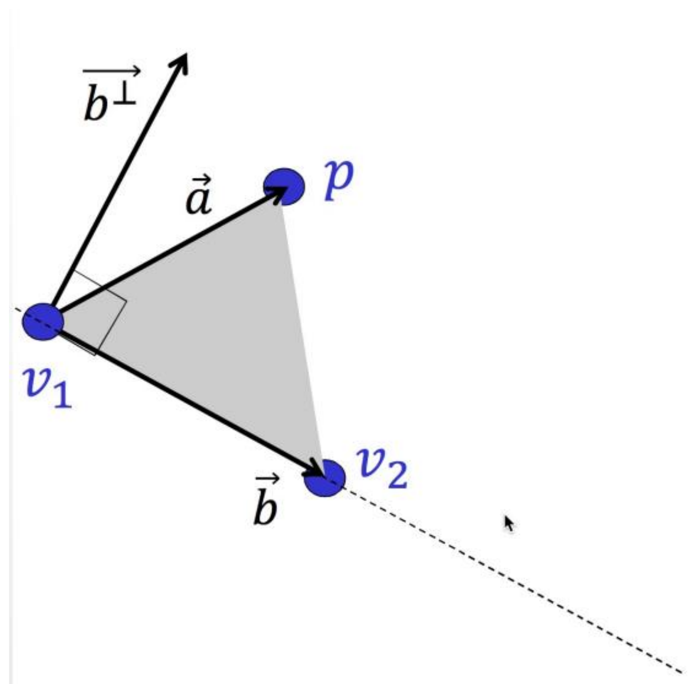


Rasterization

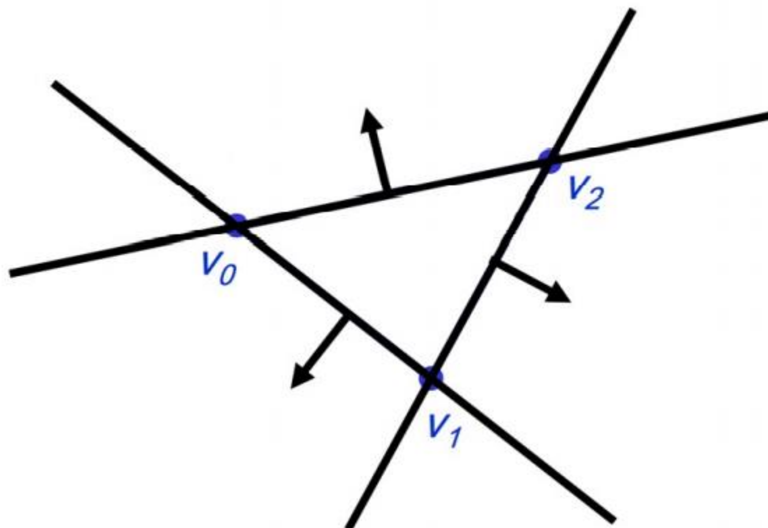
For **rasterizing that area,** we aren’t going to use the **scan conversion algorithm**, this is not doable because it is not really fast for a parallel computation.

We use a test called **half-plane test.**

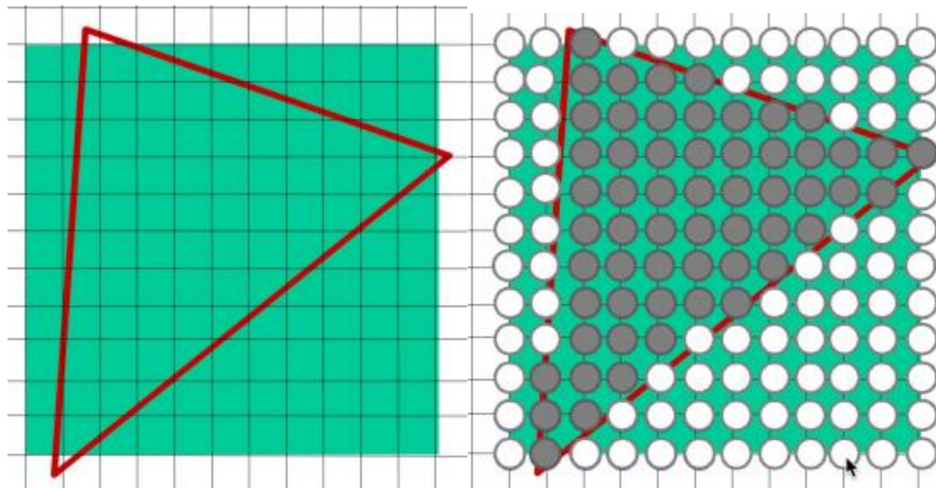
If I have a point **p** and i want to know if it lies on the **same side** of a **line** connect **v1** and **v2**, if I want to know if **p** is on positive side of the line or negative I do the **dot(a, b),** if the result is greater than zero it lies on the **same edge** of the **v2 otherwise not**.



I can do this test for determine if a point is inside a triangle, I just have to reply to this test for **three times one for each edge**, a point will be inside the triangle if and only if all **three** the **test fails** (dot product result < 0), as shown in figure means that the projection come from the inside (in the before figure means that the projection comes from behind).

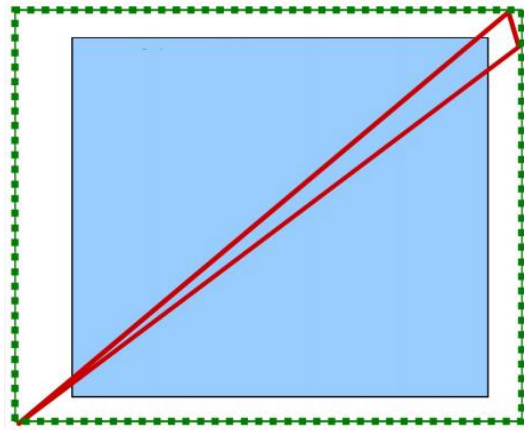


We can use this approach to determine if a point have to be rasterized, we take the **Bounding Box** of the triangle and we round it to **integer division** by **two** (it may be smaller than the original triangle).



Then for each vertex of the blocks we apply the **half-plane test** with the triangle edges.

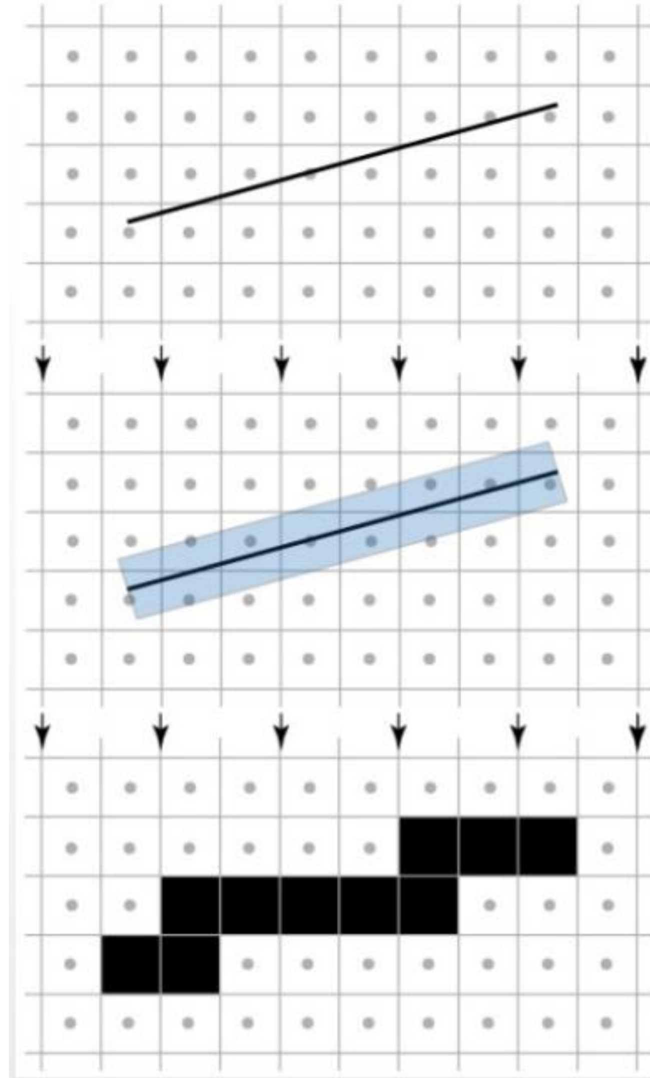
If all the four sides of the square are inside that one will be a fragment of the triangle, if not that one will not be a fragment for the triangle. This approach is **highly parallelizable** in **hardware,** and also this isn't a fixed rule this is **implementation dependent**. The issues are related with a potential overhead due to many unnecessary test on fragments. The worst case possible is a long and narrow triangle placed along the diagonal.



During the **rasterization** after the determination of the fragments composing the triangle, there will occure the **interpolatation** of texture coordinates, normals, colors of the original vertices of the triangle.

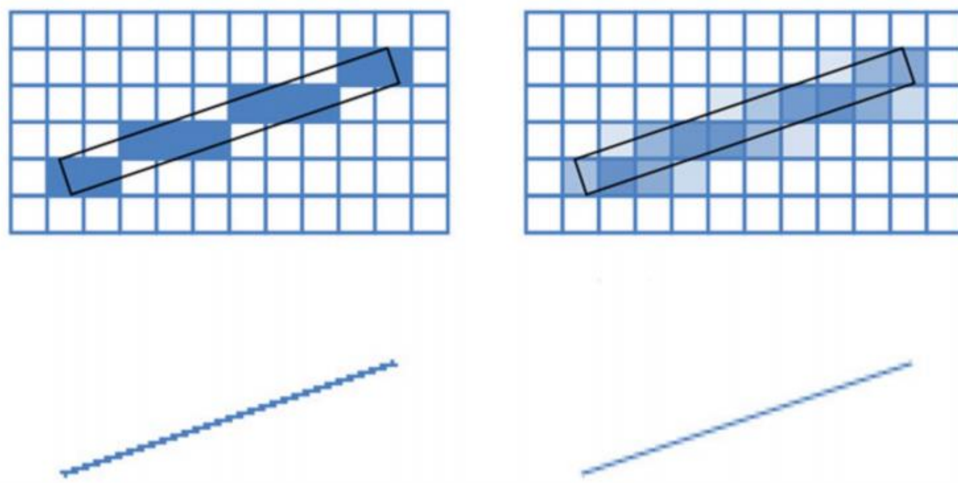
Antialiasing

The **rasterization** introduces a problem called ***aliasing***, when we sample a continuous line to convert it into a set of fragments by approximating the original signal we have an issue called **aliasing**. In our case the **aliasing** is given from an inclined line represented from *jaggies (pixels placed as a stair)* during the transformation from continuous to discrete.



If you do some **pixel painting** on some imaging software and you draw a line and you see the ***jagged*** representation this is a raster image. This is why we use **vectorial graphics** which uses a **mathematical representation** of **figures**, so you can sample that mathematical representation without having the *jaggies*, this because all the information is contained inside the representation.

The solution for this is **antialiasing,** this is another topic which is heavily investigated in the recent years since consumers obviously prefer a smoother image. Higher resolution we use the harder the *jaggies* are to see, but there is no way to remove completely them.



On the right there is the antialiasing applied version of the left image, we can see that **antialiasing** pick some less saturated pixel of the line placed in such a way (by sampling technique) that will give us the perception that there is no drastically change of colors/frequency.

All these techniques are done on the GPU without our intervention, is a technique that card manufacturers decided to use inside the GPU.

FSAA

Fullscreen Antialiasing, it is based on **Super Sampling** (or *oversampling*) methods and It is conceptually the simplest.

It renders the scene at higher resolution and then filters neighboring samples to create an image.

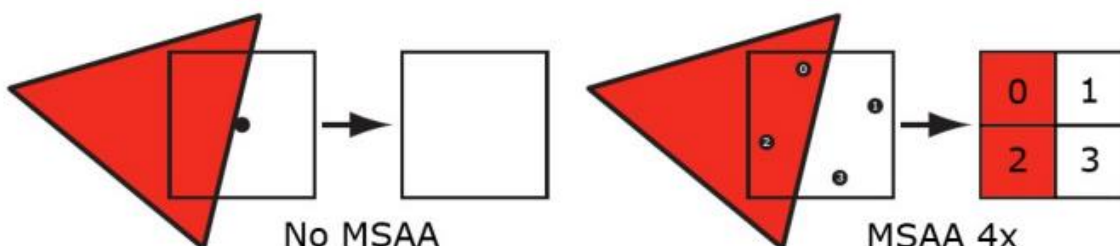
if I have to render a FullHD frame we sample an image Nx times bigger and then we down sample, where N is the sampling factor. Means that we will have to sample a **7680x4320** image for a **FSAA 4x**.

The algorithm is really simple, but the computational cost is extremely high.

MSAA

Multisampling Antialiasing is a more sophisticated technique, the idea was to sample multiple times the fragments on the edge and then do an average of these sub-pixel's samples.

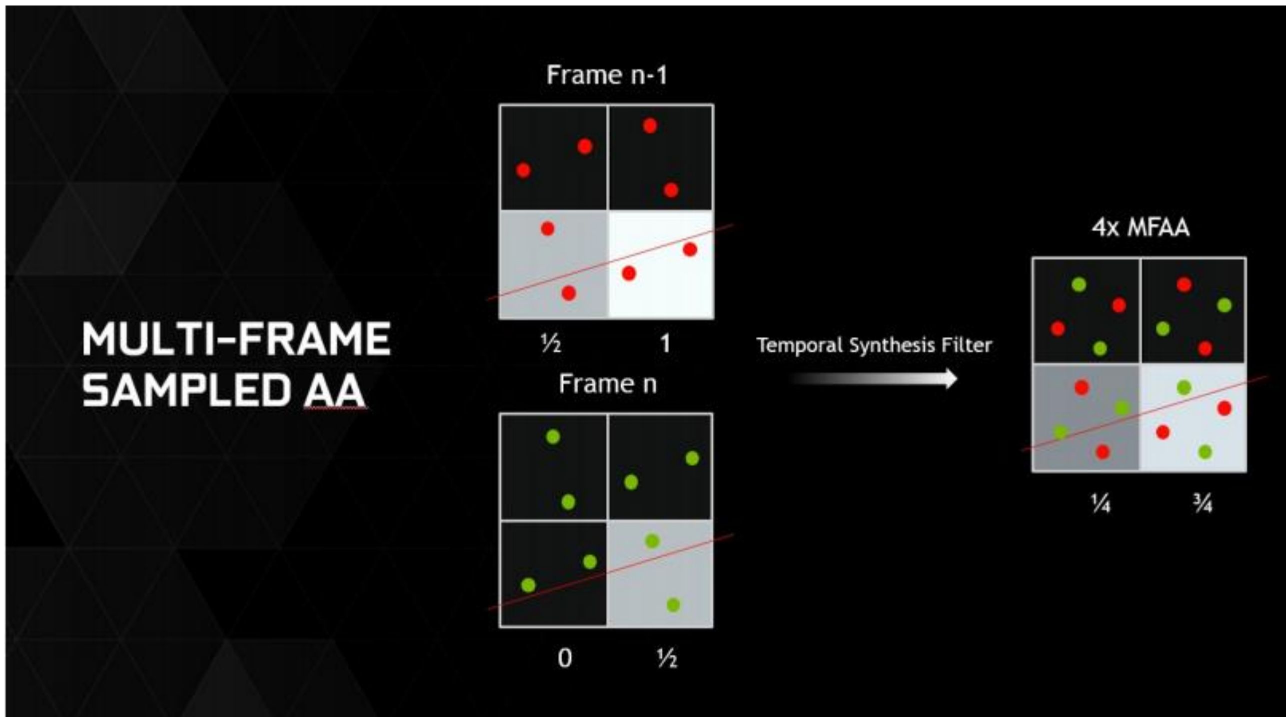
This approach is faster the **FSAA**/**SSAA**. This approach is less easy to use with **deferred rendering** or **multiple render targets** technique. Because, when we have to use the first pass of the **GPU Pipeline**, in this case we have issues, this kind of techniques often use a random or a selection of patterns to do the multisampling, I don't always use the same positions of samples.



MFAA

The idea behind the **Multi-Frame Sampled Anti-Aliasing** is to achieve the same quality of the **4xMSAA** with just **two sample** per fragment. This was proposed by NVIDIA with Maxwell GPU

architecture. It provides an alternation of **spatial position** of the sample and exploiting some temporal passage of the sampled value they have achieved the same quality of **4xMSAA** with less computation (at the cost of **2xMSAA**).



MLAA

The **Morphological Antialiasing,** born in the 2009. More than 10 years ago there is an introduction of antialiasing technique done in the post-processing.

It can reach the **MSAA** quality of **AA** with a **computational time** of *5msec*. It is based on image processing and pattern recognition techniques like :

- *edge detection*
- *analysis of patterns around edges*
- *edges "smoothing" with color blending.*

It is a **computational efficient** technique, it works only on information actually present in the frame, and there is no need to generate many samples to average a single value.

The idea behind the **MLAA** is that i won't apply **AA** as a **black-box technique** inside the GPU Pipeline, but I will apply the **AA** as **post process effect,** on the Pixel Shader to smooth the result of the previous Render.

SMAA

The **Enhanced Subpixel morphological AA** (SMAA) was released by Crytek, and it is a new improved technique based on the same approach of MLAA. It is an extension and optimization of **MLAA**, with more pattern recognition involved, and with the improved *edge detection* they also introduced **temporal** and **spatial multisampling**. It is **faster** and **more efficient** then **MLAA.**

FXAA

The **Fast Approximate Anti-Aliasing** was released in 2009 by NVDIA. The idea was the same of **image processing** and **pattern recognition** implemented with different level of complexity in order to produce in **post-processing** an **AA** image.

Intersection tests picking

They are very important techniques used in **Computer Graphics** and in lot of helpful situations like *ray-casting*, *culling*, collision detection, relative distances between objects…

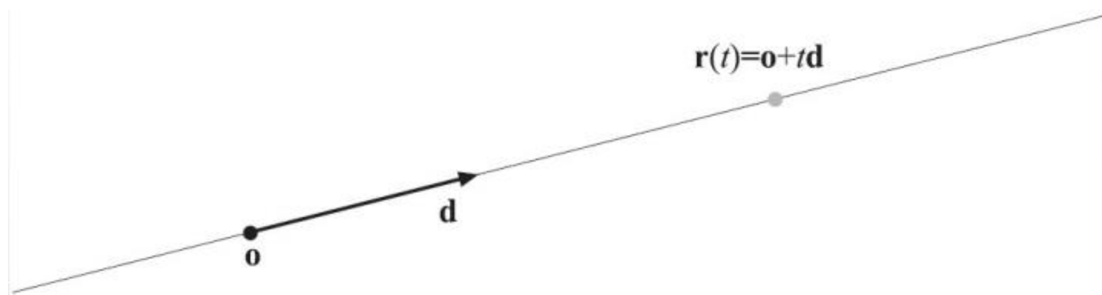
We need an intersection test when we want to know if two objects are **intersected** totally or partially. But also provides useful information like knowing **the intersection point** or the **distance to the point of intersection.**

Most of these techniques rely on some simple mathematical element, the **Ray**.

Ray

A ray is composed from :

- **Point**
- **Direction vector**
- **Scalar *t*,** for determining the points on the ray



Sphere

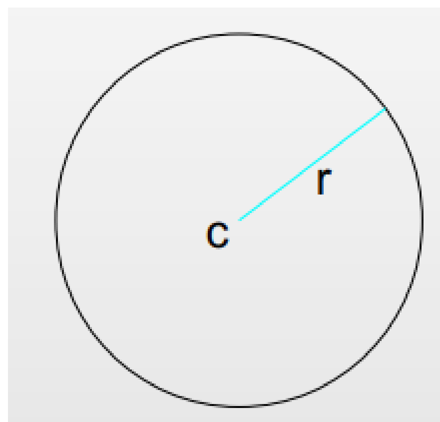
The sphere is element used a lot for doing the intersection, because it is a simple mathematical element, we have two notations for the sphere.

Implicit notation

$$f(p) = p_x^2 + p_y^2 + p_z^2 - r^2 = 0$$

Explicit notation

$$f(p) = \|p - c\| - r = 0$$



Intersection tests, rules of thumb

As goal we want to achieve the maximum efficiency, we do this test to avoid more complicated computation. This test is simple computation for avoid a harder one!

As usual to introduce some **preliminary test** we have to keep in mind that everything must be optimized and accurately designed.

In case of lot tests, we first do the easy and quick tests and then the complex one. In this way I can check the **BVH** to get a much more accurate sequence of operations.

To perform the intersection test in different orders i may decide to switch the order of the test I apply or in some cases I raise the number of the test, but I reduce the dimension.

For example, three tests in 1D cost much less then a single test in 3D.

Ray/Sphere intersection : mathematical solution

$$r(t) = o + td$$

$$f(p) = \|p - c\| - r = 0$$

The sphere the center which is *c*

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

$$\downarrow$$

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

$$\downarrow \quad (d \cdot d) = \|d\| = 1$$

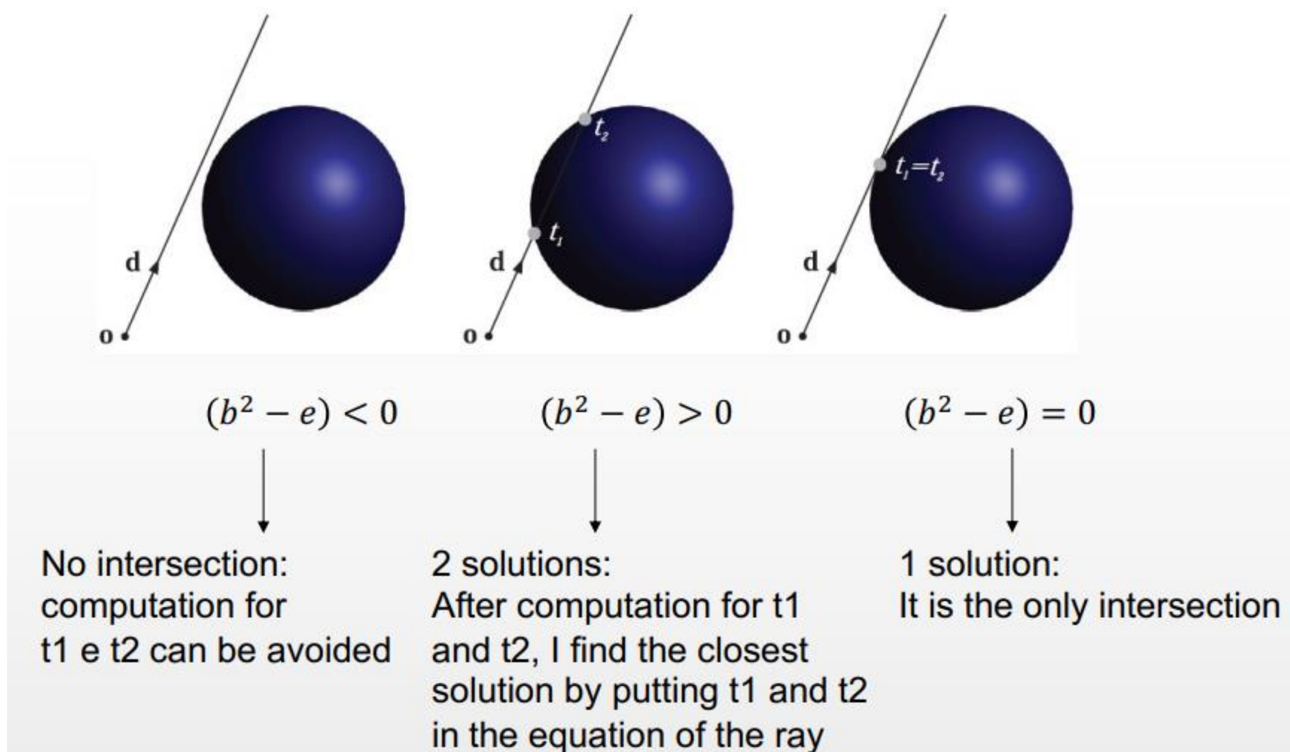
$$t^2 + 2t\big((o - c) \cdot d\big) + (o - c) \cdot (o - c) - r^2 = 0$$

$$\downarrow \quad b = (o - c) \cdot d, \quad e = (o - c) \cdot (o - c) - r^2$$

$$t^2 + 2tb + e = 0 \longrightarrow t = -b \pm \sqrt{b^2 - e}$$

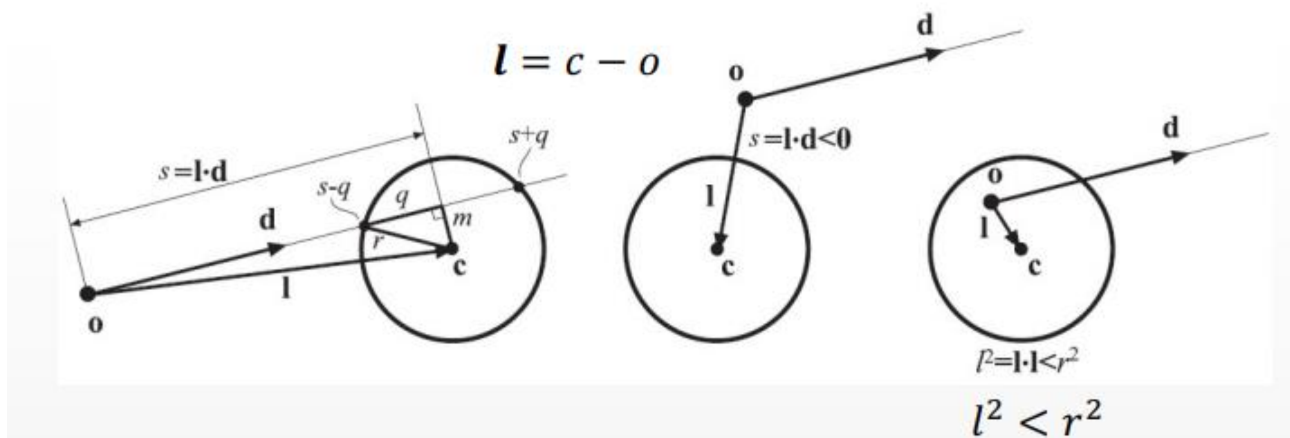
Solutions:

$$t = -b \pm \sqrt{b^2 - e}$$



In case of different delta's result I will find myself in three different situations.

Ray/Sphere intersection : geometrical solution



The vector ***l*** is a vector that represents the distance from the origin of the ray to the center of the sphere. The vector ***d*** is the direction of the ray.

If the squared vector ***l*** is less then squared radius ***r***, this means that the ray/vector ***l*** *has the origin* inside the sphere. This means that there will always be an intersection in this case, and if I need to know if they intersect, I can just stop at this computation.

In the case where the squared vector ***l*** is greater than the squared radius ***r*,** this means that the origin of the ray isn't inside the sphere, but we still don't know if there is an intersection. To know that we have to calculate the projection of the vector ***l*** on the direction vector of the ray ***d*,** we call that ***s***.

- $s < 0$, means that the sphere center is behind the ray origin, no intersection.

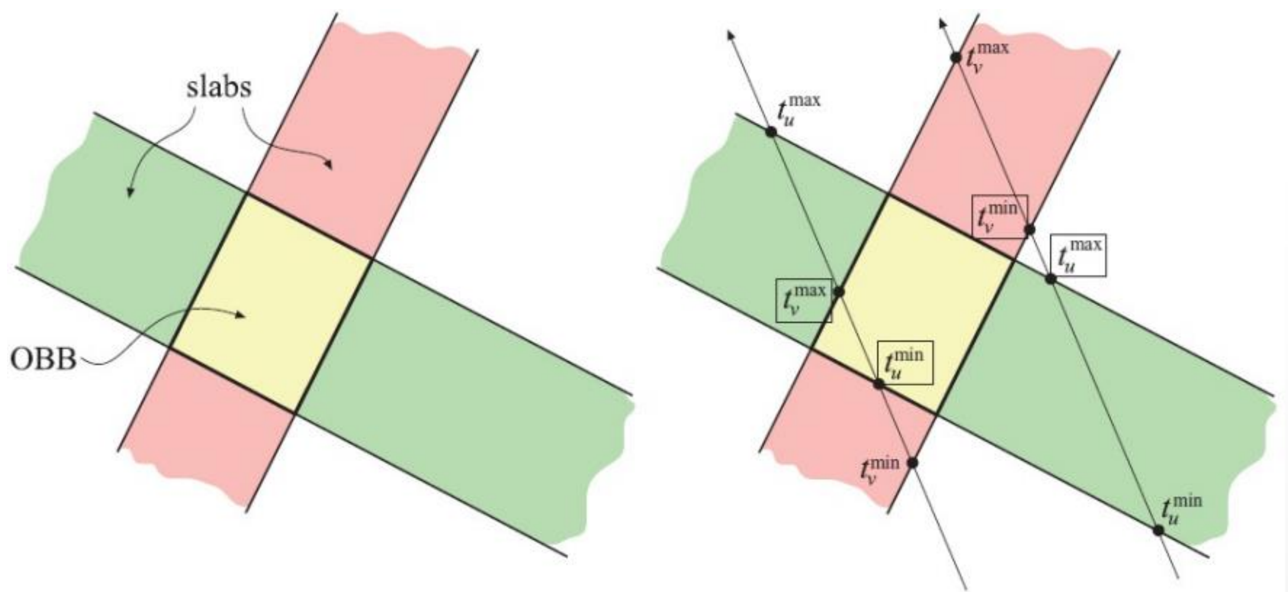
- $s > 0$, we can have two possibilities in base of the value of *m* which its length is obtained by the formula $m^2 = l^2 - s^2$
  - $m > r^2$, means that there is no intersection, the ray is out the sphere.
  - $m \leq r^2$, they intersect (and I can stop there if I just need to know this). Now we have to find *q,* to do that we solve the equation $q^2 = r^2 - m^2$**,** the solution of the ray intersecting the sphere are expressed from be $t = s \pm q$

Between the **mathematical** and **geometrical** solution, it's better to choose the geometrical, this because the test starts earlier. In the mathematical case I have to setup lot of operations before do the intersection test.

Ray-Box intersection (OBB and AABB)

I want to know if a **Bounding Box** of any object is intersected from a **Ray**, I can do this before the **Ray-Object intersection**, which is a more complex.

With very few computations I can determine if a box is crossing the BB. This approach is applicable to the slaps of the k-dop.



Imagine that we are looking to the box from the top, let's switch the test between **ray** and **planes** (which are the faces of the BB). Let's also consider the ***slabs*** of this planes.

Now let's consider two rays, the one on the right isn't intersecting the **BB** and the one on the left yes. Now let's consider the **Ray-Slabs Planes** intersection, this is really easy to do, I just have to put the equation of the ray in the equation of the plane (no solution if they are parallel).

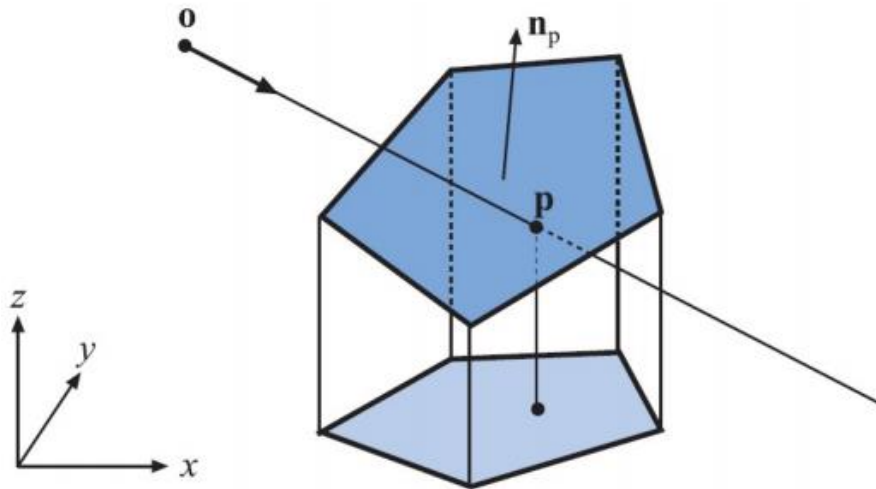
The intersection points for the two possible slabs are t_v for the red slab and t_u for the green slab, they both have a minimum and maximum value for the intersection of the slab.

Now we search for the ***maximum*** t_{min} and the ***minimum*** t_{max} , after we found these two values we check if $t_{min} \leq t_{max}$, in that case we have an intersection.

This intersection test is also appliable to the k-dop BB.

Ray-Polygon intersection

We don't have polygons with more than three-sides in **Real Time Graphics**, but for the sake of explaining you what it can be done we may have to consider an intersection between a ray and a generic polygon.



I'm showing you this case because is one of the cases where you can reduce the problem by reducing the dimensions.

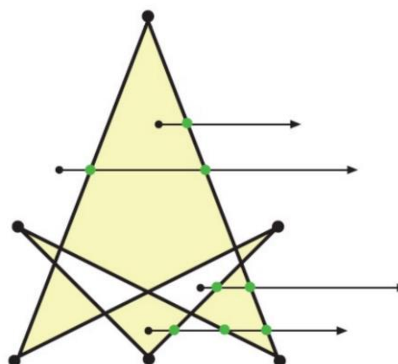
First of all, we consider the **infinite plane of the polygon**, if the plane exists we then check if the **ray** is **parallel** to the **plane**, this means that there is no **intersection**.

We can then reduce the problem from **3D** to **2D**, by **projecting** everything (polygon + ray) on one plane of the world (*XY, XZ or YZ*).

Now we want to know where the polygon area is greater. The polygon could be a face of an object, which could be oriented in a certain way, if we want to project this polygon-face on a plane and we want to check where the area is greater, we have then to discard the biggest normal component (which contains the orientation of the object).

After that I can check if a point is inside the polygon by using the **Crossing Test** (2D test).

Crossing test



A point is inside a polygon if a ray from the point in an arbitrary direction crosses an odd number of edges. This approach is good for generic polygons which are rarely used in **RTGP**.

In our case the most common case of Ray-Polygon intersection happens with triangles, and there are more efficient techniques proposed for this case (e.g., *half-plane test*).

Plane-Box intersection (OBB and AABB)

Plane equation:

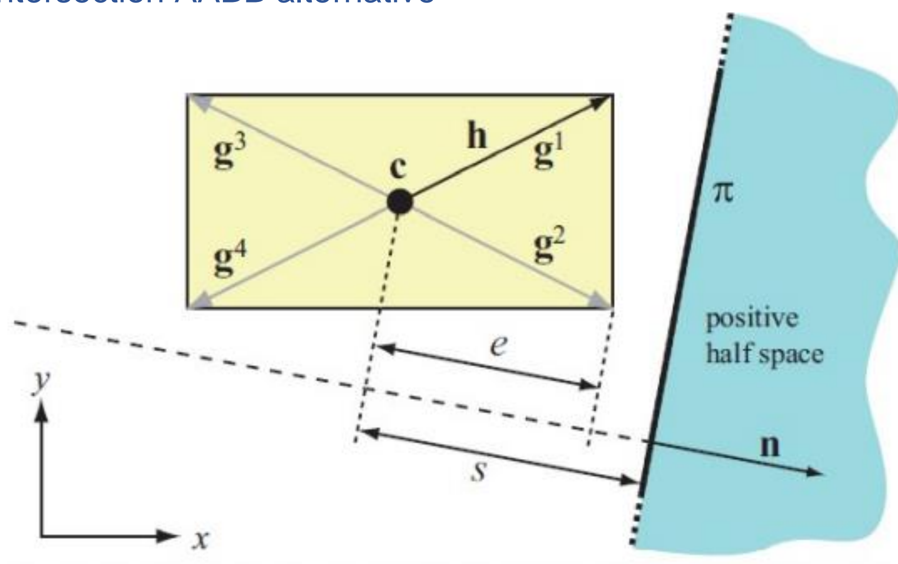
$$n \cdot x + d = 0$$

Each vertex $\boldsymbol{v_i}$ of the BB is inserted in plane equation :

$$f(v_i) = n \cdot v_i + d$$

If the vertices of an edge got all the same sign (sign of the half-plane), this means that there is no intersection between the plane and the box.

Plane-Box intersection AABB alternative



We compute the **e,** which is the *extent* of **AABB** projected on the plane normal.

$$e = |h_x|n_x + |h_y|h_y + |h_z|h_z$$

Then we consider the *s*, which is the signed distance from the center of the AABB to the plane.

$$s = c \cdot n + d$$

If $\boldsymbol{s - e > 0}$, the AABB totally in negative half-space.

If $\boldsymbol{s + e < 0}$ AA totally in positive half space.

If $\boldsymbol{-e \leq s \leq +e}$, the AABB intersects the plane.

Plane-sphere intersection

Plane equation:

$$n \cdot x + d = 0$$

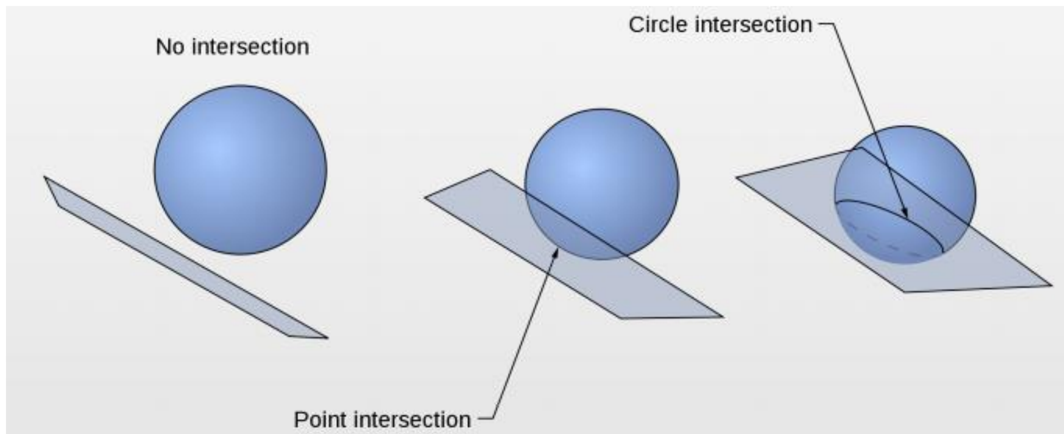
We got a sphere of center $\boldsymbol{c}$ and radius $\boldsymbol{r}$. You calculate the equation of the plane by using $\boldsymbol{c}$ as *x*.

$$f(c) = n \cdot c + d$$

$|\boldsymbol{f}(\boldsymbol{c})| > \boldsymbol{r}$, there is no intersection.

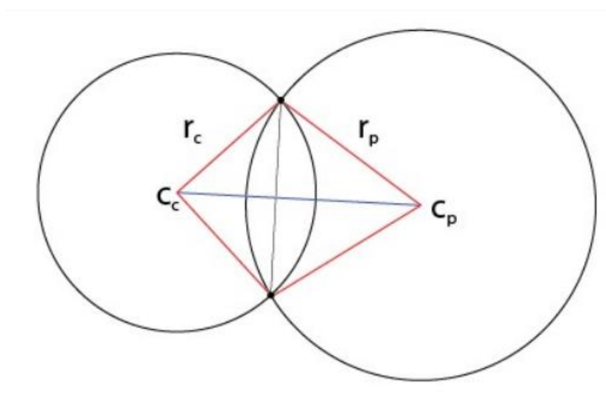
$|\boldsymbol{f}(\boldsymbol{c})| = \boldsymbol{r}$, the sphere *touches* the plane on a single point.

$|\boldsymbol{f}(\boldsymbol{c})| < \boldsymbol{r}$, the plane intersects the sphere.



Sphere-sphere intersection

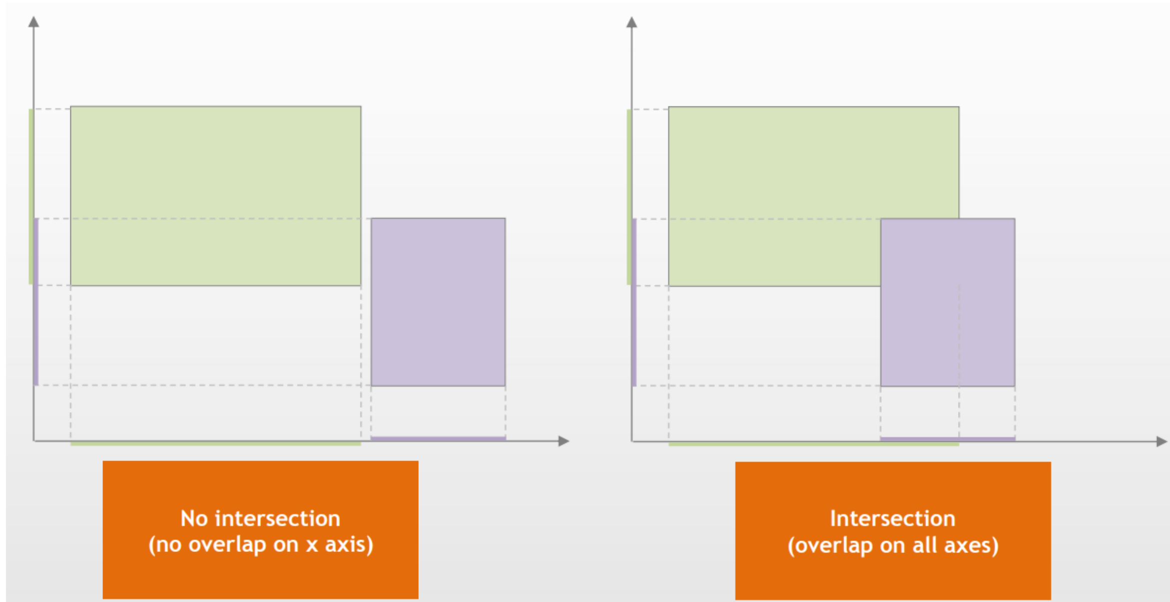
You simply take the center, and you compute the distance it is trivial.



AABB-AABB intersection

The **AABB** faces are aligned with the world axes. We may want to consider if two **AABB** are intersecting, in this is case having *three* **1D** test is better then have *one* **3D** test.

Because the 1D are really fast. I take the AABB and i consider the projection in all the directions. If there is an **overlapping** in **all** the **three projections**, there is an **intersection** between the **AABB**.



Sphere-AABB intersection

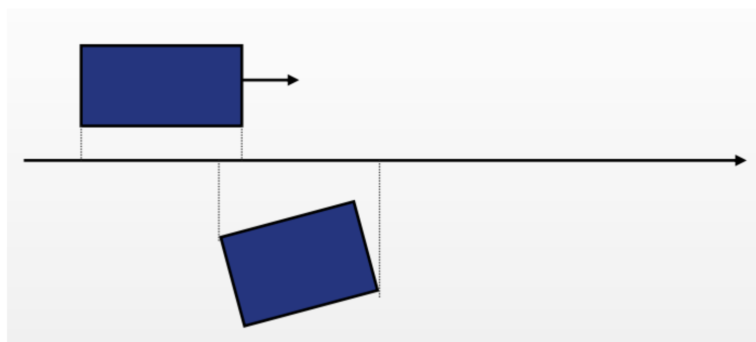
Even in this case we reduce to three **1D** tests, where we check for each axis if the center of the sphere is between the ***min*** and ***max*** **AABB** values. If the ***center*** is inside then there is an intersection, otherwise we **compute** the **distance between center** and **AABB**.

After these **three tests,** we compare the sum of squared distances compared to squared radius. If the sum is less than the squared radius, there is an intersection ***

Separating Axis Theorem (SAT)

The two convex polyhedral are disjoint if at least I can find one of these axes separating the object :

- an axis orthogonal to the face of the first polyhedral.
- an axis orthogonal to face of the second polyhedral.
- an axis generated by the cross product between an edge of the first and an edge of the second.



In this case i can find an axis orthogonal to the first box which separe the two objects.

SAT is used for some intersection tests :

- Triangle/Triangle
- Triangle/Box
- OBB/OBB

Triangle-Triangle intersection

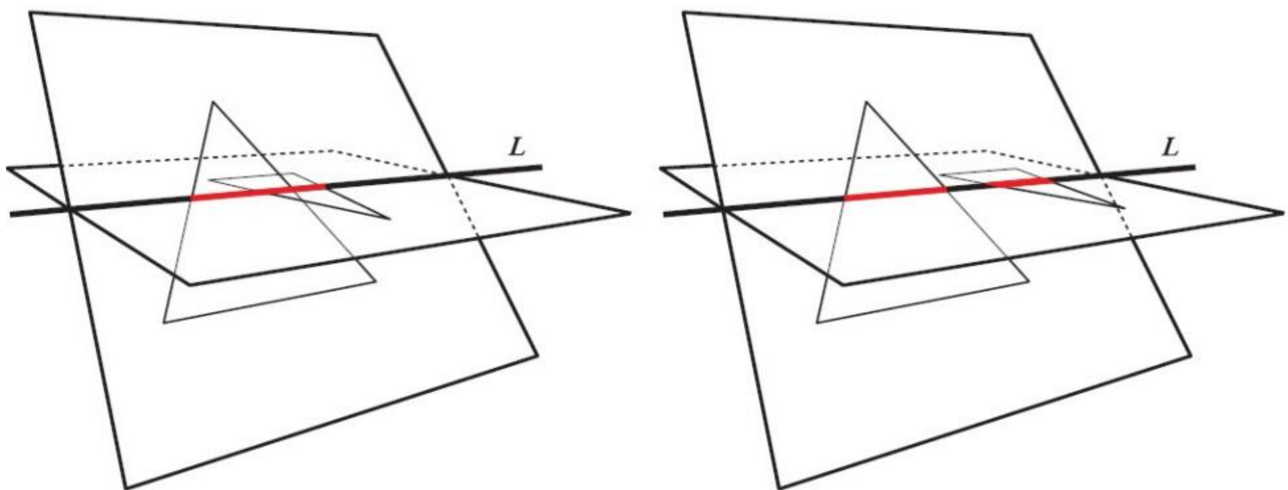
This is an alternative of the **SAT** specific for two triangles.

Given two triangles, we can test the intersection between the two infinites plain defined by them. So, we perform the first two tests :

- Test if the first triangle intersects the plane of the second triangle.
- Test if the second triangle intersects the plane of the first.

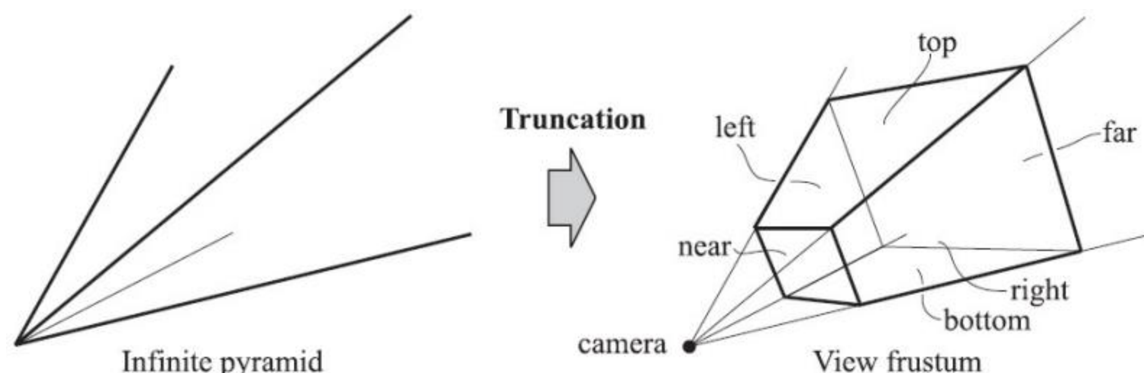
If both the tests fail, this means that there is no intersection between the planes (or that the triangles are on the same plane or on different but parallel planes).

In case of intersection there will be a resulting line **L**, now let's consider how this line crosses the two triangles. If these two segments are disjointed the two triangles are not intersecting, vice versa if they are jointed together.



View Frustum intersection

Frustum Culling is mainly based that I want to know if an object is **inside**, **outside** or **intersecting** the *View Frustum* (for the culling). We reduce this situation to a test between the **BV** and the six planes of the frustum.



View frustum testing

I can exploit the **BVH** and apply the tests on the **BV**.

- If the test says that a **BV** is completely **inside** or **outside** from the frustum, i can stop here.
- If the **BVH** intersects the frustum there could be applied other tests to lower level of BVH or it could get classified as "*probably inside*" (this for avoid).

In the case of **Bounding Sphere**, so **Sphere-Frustum intersection**.

We consider the positive **half-space** outside the frustum; I consider the signed distances from the sphere center to the six planes (six **Sphere-Plane** tests).

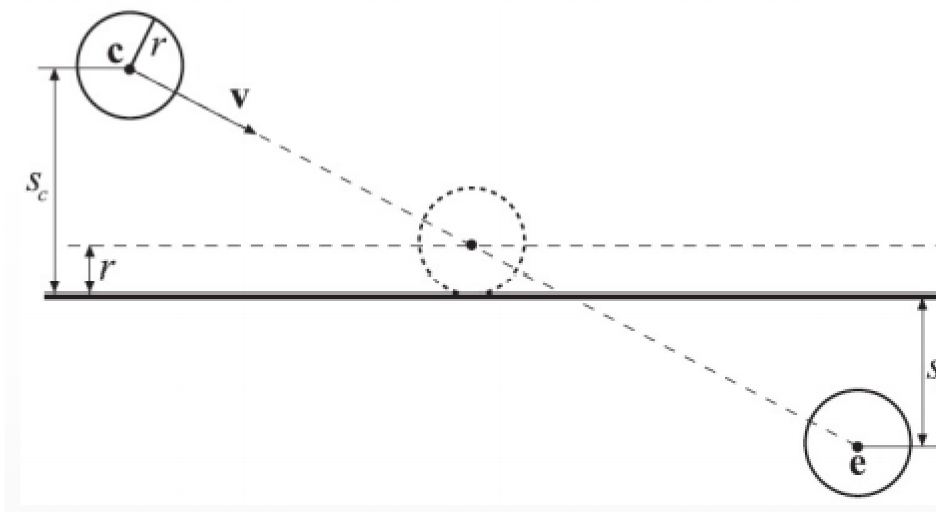
- If all the distances are greater than the sphere radius, this means that the sphere is totally outside the frustum (no intersection).
- If all the distances are smaller than the *opposite* of the sphere radius (-sphere radius), then the sphere will be all inside the *View Frustum* (no intersection).
- In all the other cases the sphere intersects the *View Frustum*.

Dynamic intersection tests

There are intersection tests which are dynamic, where the object is moving, and you want to consider different time of interval the situations. These tests are applied at discrete times, they are need for dynamic tests for moving objects.

For this kind of tests usually the spheres are considered as **BV**.

Dynamic intersection tests : Sphere-Plane



We have a **moving object**, and at certain frame the object is at certain position over the plane, at the following frame you again check the position of the sphere.

If the **center** of the sphere in the previous and current frame is on the same side of the plane, then there is no collision.

The same if the distance from the center of the sphere in the first frame is greater than the same distance but at the time of when the sphere touches the plane (vice versa for the distance in the second frame, it will be less than the distance when it touches the plane).

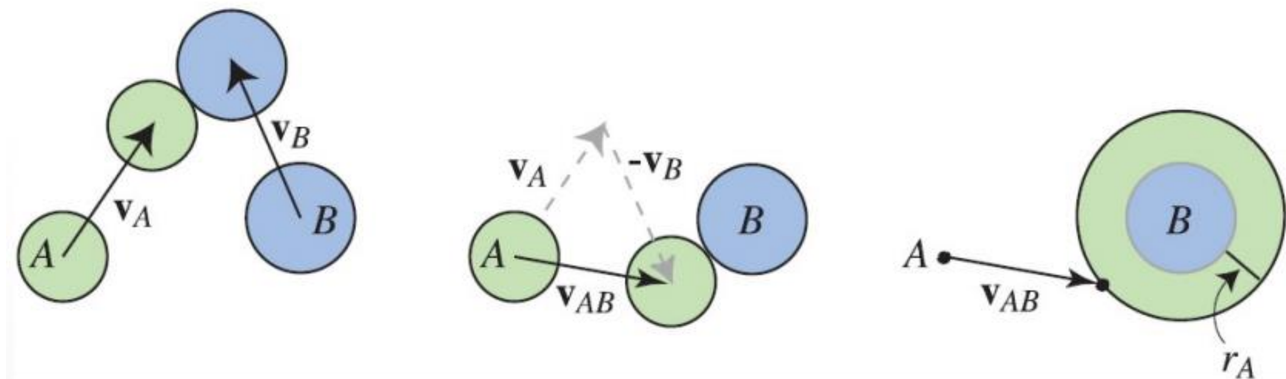
The **time of intersection** is computed by using the position of the center of the first and current frame, with this time i can decide what to do.

A simple response would be a reflection of $\boldsymbol{v}$ around the plane normal, or the sphere moved by $(1-t)r$.

Dynamic intersection tests : Sphere-Sphere

I can manage this situation by **reducing** the **complexity** of the problem, let's imagine I have **two spheres** the green (called **A**) and the blue one (which is **B**).

We have a starting situation which is dynamic, there is a directional movement, the aim is to know if they are touching. We can reduce the problem to make it equivalent to a **Ray-Sphere intersection**.



Starting from the original situation, and applying the **principle of relative motion**, I will get to an equivalent state if i compute a subtraction between the velocity vectors of the sphere $\boldsymbol{v_A} - \boldsymbol{v_B}$.

In this new situation (second image) the blue sphere (**B**) is now *static*, and the green sphere (**A**) is now moving using the new **velocity vector** $\boldsymbol{v_{AB}}$.

Now if I change the blue sphere (**B**) by adding to it's radius the radius of the green sphere (**A**), we get a new bigger sphere which is the composite of both **B** and **A.** In this way now we can consider **A** as a **single point**. We can now apply the **Ray-Sphere** intersection.

We will use the $\boldsymbol{v_{AB}}$, and the $\boldsymbol{l}$ vector which represents the distance between the two centers. Then I put everything in the equation of the sphere and reduce everything to a second order equation.

$$(v_{AB} \cdot v_{AB})t^2 + 2(l \cdot v_{AB})t + l \cdot l - (r_A + r_B)^2 = 0$$

$$a = (v_{AB} \cdot v_{AB})$$

$$b = 2(l \cdot v_{AB})$$

$$c = l \cdot l - (r_A + r_B)^2$$

$$at^2 + bt + c = 0$$

I compute the solutions and I take the smallest of the two solutions as the **time** of the **first intersection**.

$$q = -\frac{1}{2}\left(b + sign(b)\sqrt{b^2 - 4ac}\right)$$

$$t_0 = \frac{q}{a} \qquad t_1 = \frac{c}{q}$$

Then i use this time ***t*** inside the ray equation, of two rays that starts from the center of the spheres and passing the time of the first intersection as parameter inside it.

$$p_A(t) = c_A + tv_A$$

$$p_B(t) = c_B + tv_B$$

In this way i can statically know the point of intersection in a dynamic situation.

Picking

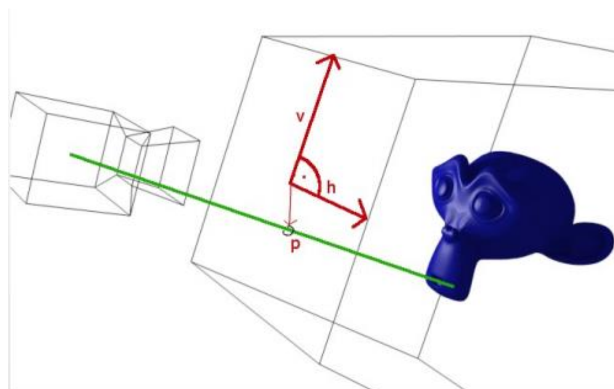
When in an application I move a selected object thought the mouse click and drag operation (or any other input device).

Actually, when we perform the click we are clicking on a window (final result of the Graphics Pipeline) not on the selected object itself. We are clicking just a colored pixel at the moment, so we got to get **backward** to the stage of before the color calculated.

The solution to this problem consists in shoot a ray in the direction of the pixel we have clicked with the mouse, and then we **convert** everything **back** (from 2D coordinates) to **3D mesh coordinates**.

This is an <u>**inverse projection**</u> from window coordinates to object coordinates.

BVH is exploited to determine the selected object, usually there is no need to perform Ray-Triangle triangle intersection.



Then I will invert the **Window Coordinate System** to the **View frustum Coordinates System**, to do that I compute the inverse matrix for the **projection**. By using that I will find a point on the near plane which I can use to determine a vector from the center of the projection to that point.

Other techniques 1

Picking is something that we may use in interaction application, not only one method to doing there is also another method "**color coding**" is an example, is less recent, you could think to assign some particular flat color not used in the scene. By using this auxiliary color for a single mesh (unique color for each mesh), when we click for picking you check on this buffer the level of the color under that position.

Very easy to use but you use a dedicated rendering buffer rendering cycle to map this color to pixels.

Other techniques 2

Another technique consists in rendering of the scene using a small window centered on the picking point. Then we got a list of objects intersecting the picking window with the corresponding depth information, anyway even this needs a **dedicated rendering cycle**.

It is possible the multiple selection of lines and vertices, performed in the Application Stage.

shader-based techniques

These are the most recent techniques; they are based on potentialities of geometry shaders. The identification of triangles is processed by shaders.

This will exploit the GPU speed for a more accurate picking and more complex selections, by the way this will involve all the Graphics Pipeline. It is used for performance and code optimization.

Collision detection (Collision Handling)

A fundamental step in most graphic applications **2D/3D** we got a lot of collisions, because we have objects we move around, and we don't want to see intersections going on. It's a very important part of the rendering Pipeline.

Actually, we use **Collision Detection** as term for identify this topic, but the correct term is **Collision Handling**, where the *collision detection* is just a part of the overall process.

The Collision Handling is composed from :

- **Collision Detection**, this will test if two objects are intersecting.
- **Collision Determination**, this will determine points of intersection.
- **Collision Response**, this will determine what actions should be taken in response to collision.

Collision Detection

Even in case of densely populated scene we need to make them work as fast as possible, by managing the **polygon soup** (mesh made from different meshes) , the **rigid body** motion (*roto-translations), the* **BVs** *(exploiting the BVH),* **Collision Map**, **Collision Solids**….

There is the "**Brute force"** approach, a formula that gives you the number of the performed tests at each frame by considering $\boldsymbol{n}$ **moving objects** and $\boldsymbol{m}$ **static objects**.

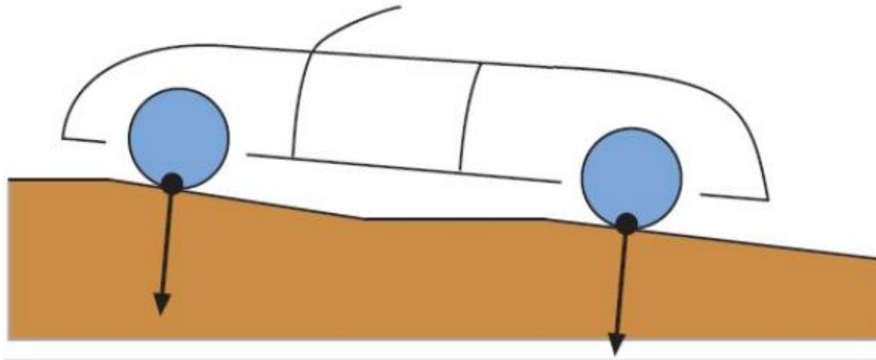
$$nm + \binom{n}{2} = nm + \frac{n(n-1)}{2}$$

This is expansive if *n* and *m* values grows, there is need for more efficient methods (could perform good if there aren't lot of objects). The **Collision Detection** is done in the **Application Stage** (mainly in the CPU), its results are managed in some way and maybe some of them are sent to the GPU (*e.g., roto translations after collision with an object*).

Simple cases

We have to consider the situation you are not forced to choose the more sophisticated but perfect technique if your scene has some simple features which allows you to choose for simple solutions, all the complexity regarding Collision Handling has a sense if the scene is complex (lot of objects and movements).

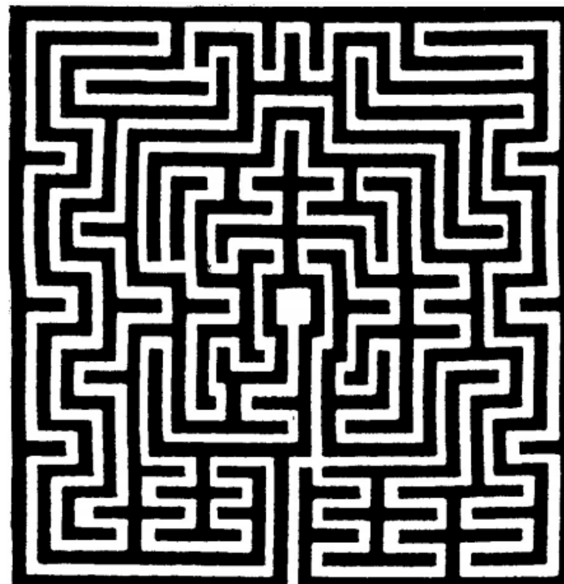
There are some cases when there aren't needed complex collision systems (e.g., Pong). In some games you just use a Ray-based intersection test.



For example, if you have a car which must always stay on a terrain, you simply take the wheels you check four rays going down and you check the intersection with the terrain. The distance between the wheel and the terrain, you will always try to keep the wheel tangent to the mesh in that point.

Another example, I have a **first-person camera** and I want to move inside a building with rooms, window and walls. **I want to avoid passing through walls**. I can shoot a ray in forward direction, and I find the intersection point then I check the distance with the position of the camera (a vector), if the distance enters a certain tolerance value I stop the camera movement (we can give a sense collision).

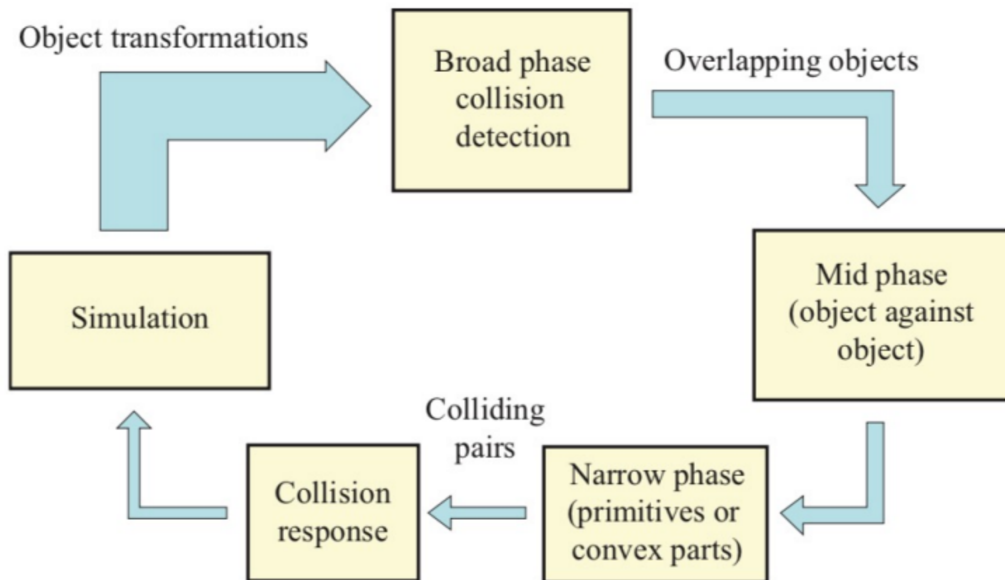
In the case of a **maze** with **flat terrains**, I can switch from **3D** to **2D**, I can limit the test on the XZ plane and discard the Y (considering that I don't have to jump, a regular maze).



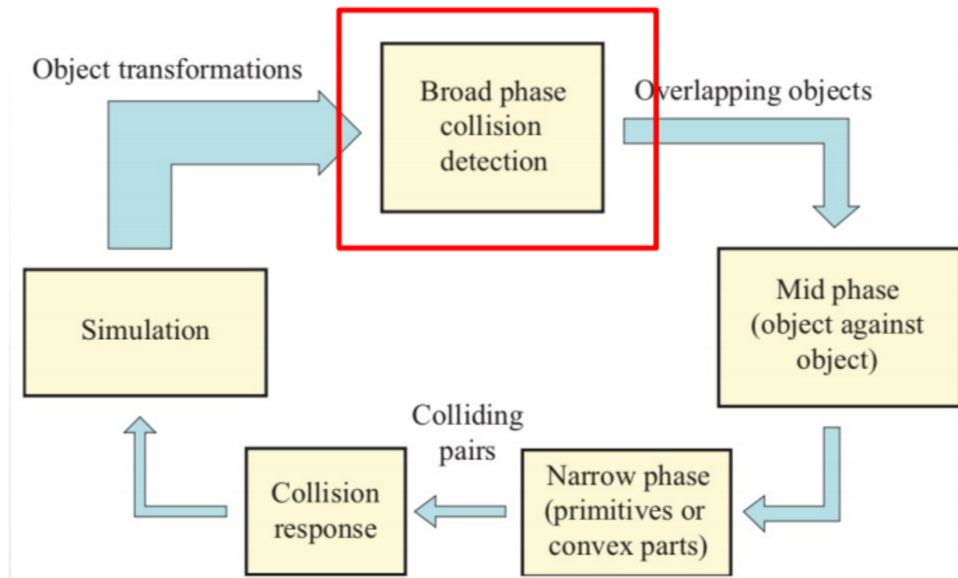
These techniques are easy and fast but not really accurate, especially for complex scenes.

Collision Detection for complex scenes

In this case I need these stages



Broad phase



I can identify pair of objects which are close enough in that frame, which may be better to check if they are colliding or not. But in a very large scene two really distant objects are probably not going to collide, so is better avoid this computation (the test of **BVHs** of each couple is not efficient $\boldsymbol{O}(\boldsymbol{n}^2)$), this is what the **broad phase** is made for.

This phase is based on **temporal** and **spatial coherence** of **objects** :

- Changes of position and orientation are small from frame to frame.
- I determine sub-set of objects close enough to potentially collide.
- Discard too distant objects.

It first identifies the **potential colliders**, and only for them I'm going to apply the intersection test (I send them to the *mid phase*). The others which are not **potential colliders** will not be checked one against each other (*e.g.,* two really distant object).

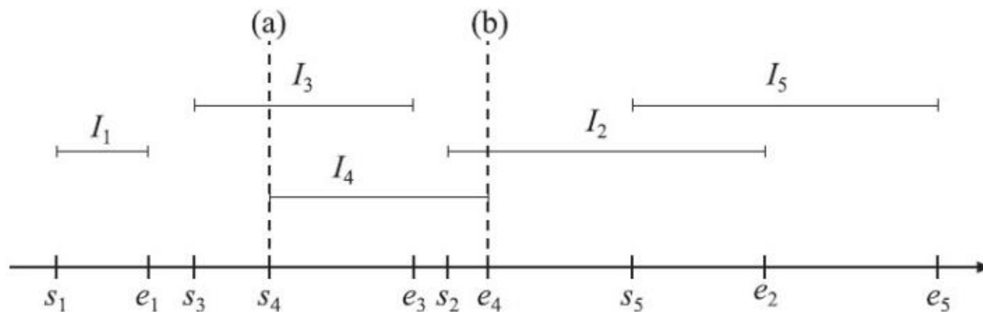
Broad phase : Sweep-and-prune

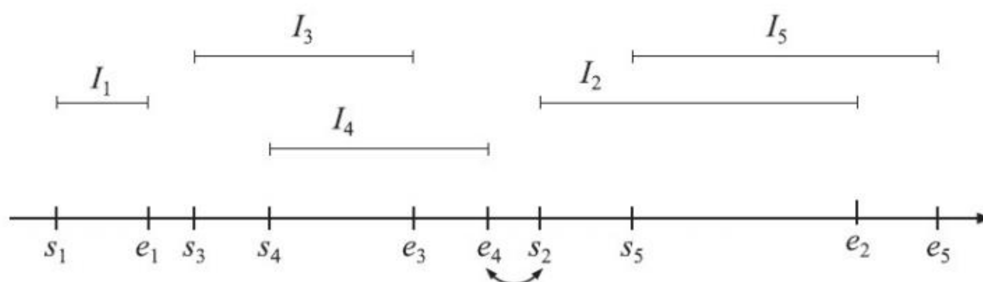
The sweep-and-prune algorithm is one of more efficient technique for the broad phase. You take an **AABB** which may contain the object in any possible orientation. I consider all the possible orientation of the object and I take the maximum extent that the object can have (very large BB).

I use it because if I take two of this very large **AABB**, if they intersect then their projection must intersect on all the three axes, so I reduce from one 3D test to three 1D tests.

- Ordered list of the intervals, where the intervals are the projection of the AABB on the axis (so i get three ordered lists).
- Search for interval overlapping.
- If intervals overlap on all three axes, then I have to send the couple to more complex tests in *mid phase.* If they are not overlapping i assume that in the next frame they are not colliding (*too distant*).
- Sorting of intervals list for the next frame.

This list has to be managed and ordered continually.





The $\boldsymbol{i_3}$, $\boldsymbol{i_4}$, $\boldsymbol{i_2}$ are **overlapping** on the same interval, in the following frame the object $\boldsymbol{i_2}$ has moved on the right and there is no more overlapping between the axes.

The complexity of this algorithm is $\boldsymbol{O(n\,logn\,+\,k)}$ the list sorting can be reduced, because for *temporal coherence* I can use ***bubble sort*** or ***insertion sort*** which are $\boldsymbol{O(n)}$**.** The $\boldsymbol{O(n)}$ is the list sweeping and $\boldsymbol{O(n)}$ is the report of *k* overlapping intervals**.**

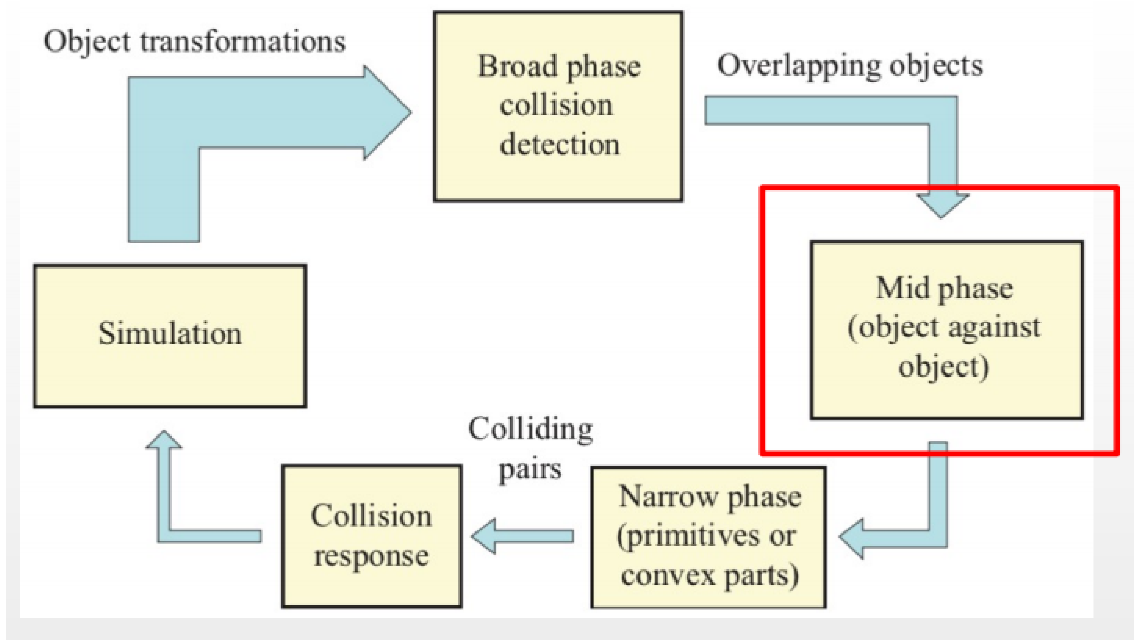
Broad phase : Grids

Division of the scene in cells, I keep the content of each cell and I send to the middle phase only the **BVs** of the objects inside the same cell.

This technique has some issues :

- **The choice of cell dimension**, I can use the greater AABB to determine the dimension of the grid.
- **Efficient memory management**, larger is the scene emptier the scene will be.

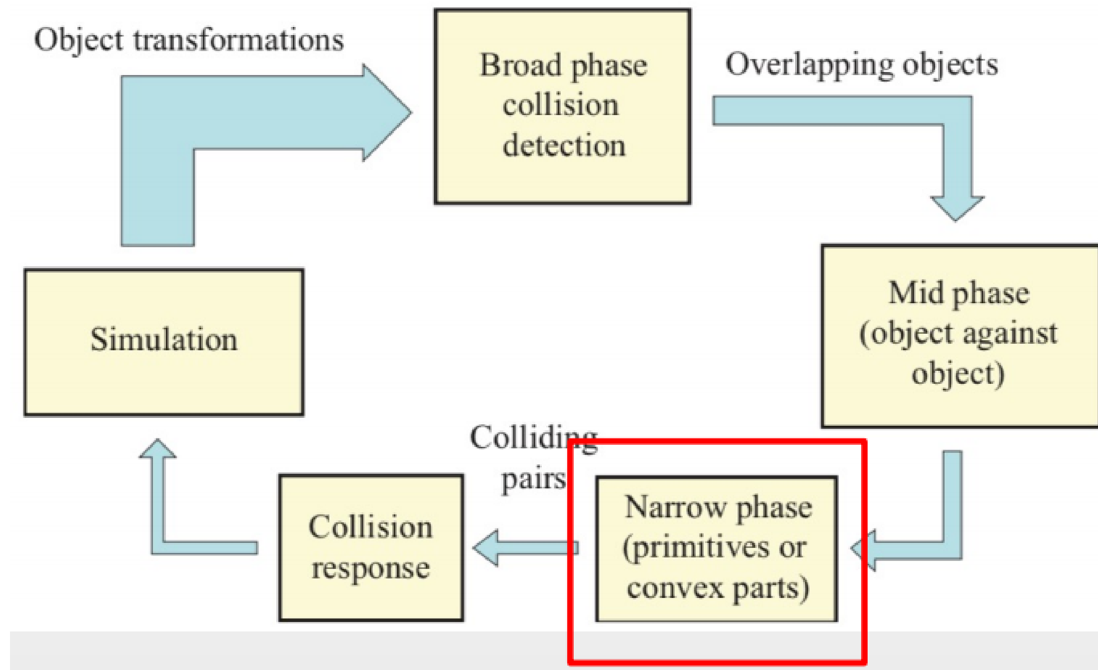
Mid-phase



It is mainly based on the intersection tests. In this phase I check **BVs** against other **BVs** (simplified test), if the two **BV** intersects there is a collision, i then have two options :

- I can directly apply the ***Collision Response***.
- In the case i need to find and accurate collision point at mesh level i will go to the ***Narrow-phase Collision Detection***.

Narrow-phase (optional)



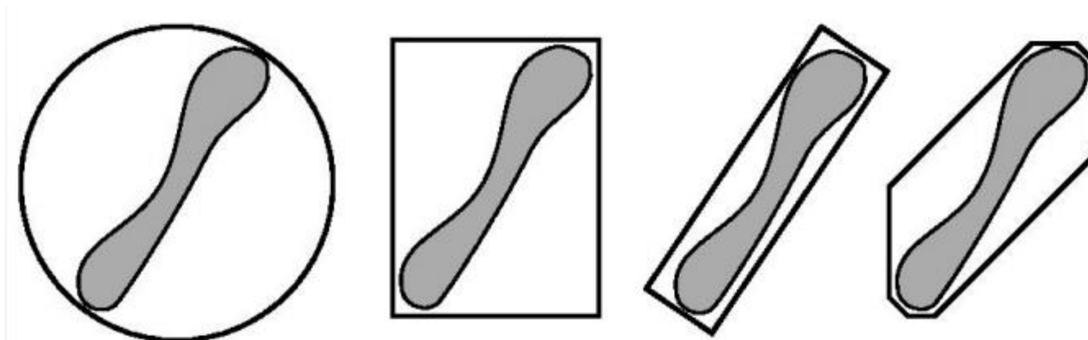
In this phase i go at primitive level to know the point of intersection on the actual mesh (for whatever reason is needed, maybe a *headshot*).

This also could be used for querying the distances between triangles in order to use that for maintaining a relative distance between two models.

This phase often exploits a BVH at primitive level, this in order to avoid as much as possible Triangle-Triangle intersection tests.

BVH at mesh level

It is a BVH for Collision Detection applied to the mesh of a single model, this is organized to cover subsets of primitives (triangles), and it is done mainly to avoid Triangle-Triangle tests as long as possible.



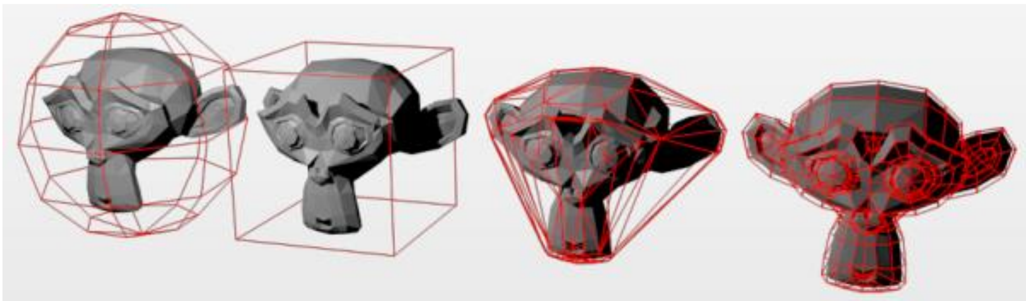
For the construction of this BVH I have two approaches :

- **Bottom-up** I start from the primitives and then I analyze the mesh and I create subsets of triangle and I create the BV on that, then I keep going up creating the tree from the leaves.
- **Top-down**, I create the whole BV of the model and then doing divide-et-imperia I split the mesh and I create the hierarchy from the root to the leaves.

BVH, Collision Solids, Collision Map, …

In some game engine could exist a distinction between **BVH** for culling and BVH for collision detection. We discussed as having the BVH of the scene, that may go at lower level that consider also the mesh, but in some engines there are two separated BVH.

- **BVH** for the **Visibility Culling**.
- **BVH** for the collision, this because a game engine is composed by different modules each one doing a specific part in sophisticated game engine you may have a specific module for collision handling based on some physics simulation often they rely on some other library or framework.



This due to the fact that a **Game Engine** is ***modularized***, and there could be a specific module for the handling of collision (maybe based on a Physic Engine) and for that you will use a different BVH.

For **Physic Engine/Collision Detection** system you have different name for BVs like *Object Proxies*, *Collision Solids/Shapes, Collision Map, …* In the end these names are defining some BV/Collision Volume associated with the mesh.

Not only the name changes in this BVH, but there are also new different shapes that are used like *capsule, cone, cylinder*. This BVs are not rendered (like the BVH for culling), they are used for Physics Simulation and Collision Handling.

Some of these Collision Volumes could not be related to objects of the scene, they are used for other purposes. *e.g.,* the sphere associate to an **FPS** camera, or some volumes are used to trigger events (*when you pass an invisible plane it will close the door*).

Collision Detection : Cost function

Number of the complexity of your scene.

$$t = n_v c_v + n_p c_p + n_u c_u$$

n_v, number of **BV-BV** tests.

c_v, cost of **BV-BV** test.

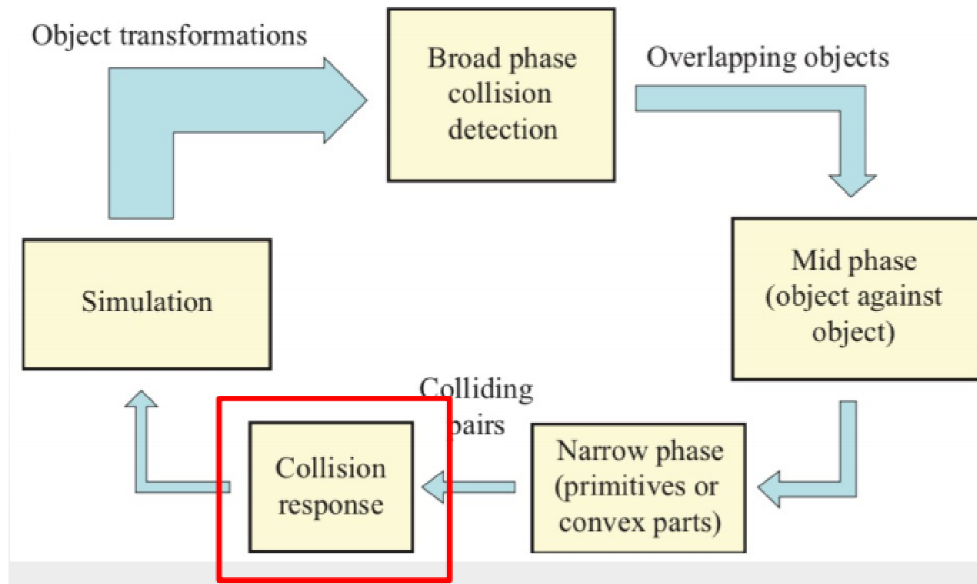
n_p, number of **Triangle-Triangle** tests.

c_p, cost of **Triangle-Triangle** test.

n_u, number of **BVs** updated due to model's motion.

c_u, cost for updating **BV**.

Collision Response



In these phases we choice the **action** in response to a *collision*, this is decided by the developer, could be a simple response like a displacement or calculated by a physics simulation model (this is used in the most realistic games, which uses a specific module *Physics Engine* usually defined in Application Stage).

Physics simulation

You want that **Collision Handling** is automatically settled up being based on already defined module (Physics Engine).

A good **Physics Engine** must be able to :

- Consider different physics phenomena and conditions.
- Must be efficient (is a process that runs before the rendering, it will be the transformation matrices that will be send to the *shaders*), this is another thread to execute. We are still aiming for real-time performance.
- Has to be accurate (***truncation error*** must be low), this means based on ***numerical method***.

Usually when physics engine can manage object with different mass :

- **Point mass**, where only **3DOF**, this means that the object will be able to have just the linear motion.
- **Rigid Body**, the object doesn't have a deformation after collision, we have **6DOF**, so the object will move and rotate.
- **Soft Body**, the objects is deformable (*cloth, liquids, etc.…*). Will be more computationally expansive.

In **Physics Simulation**, all the simulation will be applied only to the collision solids. When you see an object in the game reacting to a force, actually the physics engine doesn't care about that mesh.

It doesn't consider the mesh at all; the physics simulation is applied to the **BV** of the Physic Engine for representing the Mesh.

At each simulation step you have to perform these operations (the first two could be constant and set up one single time) :

- Setup parameter of the objects like *mass, …*
- Setting up the parameters of forces : *gravity, friction, …*
- Calculation of acceleration
- Calculation of velocity
- Calculation of position

Accuracy

The accuracy of the Physics simulation is affected by the adoption of different **ODE** (Ordinary Differential Equation) solving methods.

You have to compromise between the time step and ODE solving method complexity.

Some well-known methods :

- Euler-Cromer
- Euler-Richardson
- Range-Kutta
- Verlet

For example, let's stay on the **Newton Second Law** with **Euler method** for solving.

You have initial acceleration, which you can get to the velocity and the to the position. This method is correct on paper, but when it is being applied computationally you **lose accuracy** of the method, this because the formula is applied at each step without keep track of the previous step.

On Paper :

$$\vec{x}(10) = 0.5 \cdot 10 \cdot 100 = 500$$

Computationally :

t=1: position = 0, velocity = 10	t=6: position = 160, velocity = 60
t=2: position = 10, velocity = 20	t=7: position = 220, velocity = 70
t=3: position = 30, velocity = 30	t=8: position = 290, velocity = 80
t=4: position = 70, velocity = 40	t=9: position = 380, velocity = 90
t=5: position = 110, velocity = 50	**t=10: position = 470, velocity = 100**

A more accurate simulation is the **Verlet integration**, it is the most used in real-time graphics, this method keeps the position of the previous time step. And you do a Taylor expansion of the previous and current position (the *odd* derivatives disappear from the equation).

$$\vec{x}(t+\Delta t) = \vec{x}(t) + \dot{\vec{x}}(t)\Delta t + \frac{1}{2}\ddot{\vec{x}}(t)\Delta t^2 + \frac{1}{6}\dddot{\vec{x}}(t)\Delta t^3 + O(\Delta t^4)$$

$$+$$

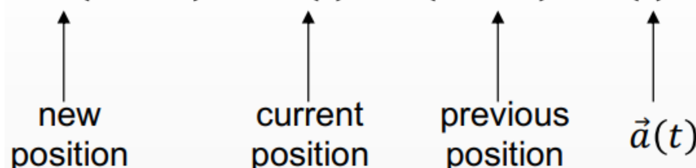
$$\vec{x}(t-\Delta t) = \vec{x}(t) - \dot{\vec{x}}(t)\Delta t + \frac{1}{2}\ddot{\vec{x}}(t)\Delta t^2 - \frac{1}{6}\dddot{\vec{x}}(t)\Delta t^3 + O(\Delta t^4)$$

In this case you don't need the velocity because it isn't explicitly used. You just need the position and acceleration (determined by the forces)

$$\vec{x}(t+\Delta t) + \vec{x}(t-\Delta t) = 2\vec{x}(t) + \ddot{\vec{x}}(t)\Delta t^2 + O(\Delta t^4)$$

$$\vec{x}(t+\Delta t) + \vec{x}(t-\Delta t) = 2\vec{x}(t) + \ddot{\vec{x}}(t)\Delta t^2 + O(\Delta t^4)$$

$$\vec{x}(t+\Delta t) = 2\vec{x}(t) - \vec{x}(t-\Delta t) + \ddot{\vec{x}}(t)\Delta t^2 + O(\Delta t^4)$$



This method could be reverse with a negative timestamp, and it is also good for the numerical stability which it gives.

Forces in Physic simulation

Gravity

The most common phenomena used is gravity, at least I have this force applied to the objects (I can use the same formula but different acceleration if I'm on a fantasy planet).

$$\overrightarrow{F_g} = m \cdot \vec{g} \qquad\qquad |g| \approx 9.8\, m/s^2$$

Friction

Another very common force is friction, the forces exported by each surface on the other, in this case is called ***dry friction***.

Usually, dry friction is simulated with coulomb friction, this friction is a force which is parallel to the surface in the opposite direction (if I push an object on the ground the force goes in the opposite direction).

$$\overrightarrow{F_k} = \overrightarrow{F_\perp} \cdot \mu_k$$

I can have also ***drag***, which is **simplified fluid resistance** (e.g., a car running on the street, I can simulate the resistance of air).

$$\overrightarrow{F_{vis}} = \vec{v} \cdot \mu_{vis}$$

As you can see they are based on a single parameter that express how strong the force is being applied and another which contains the direction (so must be a vector).

Linear momentum

More comprehensive simulation of the forces, in particular with roto translation we need also to introduce and simulate the different physics parameters like linear momentum. Multiplication between mass and the velocity

$$\overrightarrow{M_l} = m \cdot \vec{v}$$

Impulse

The impulse is a change in the linear momentum in the same direction of the movement, so I take the linear momentum and I change it. The impulse is the result of a strong force applied in a very short time, which usually change an object change very rapidly the speed and direction.

$$\overrightarrow{I_F} = \vec{F} \cdot dt$$

$$\overrightarrow{I_F} = d\overrightarrow{M_l}$$

(Change in linear momentum in the same direction)

Angular momentum

When we introduce the rotation, we have also to consider the angular momentum which is defined with the I, a parameter that describes the resistance that a body applies to change its angular speed (*Inertia*). The moment of inertia is *how strong the resistance of this object to this movement.*

The other parameter is *w*, which is the angular speed (during the rotation movement).

$$\overrightarrow{M_a} = I \cdot \vec{w}$$

Torque

The torque (*momento meccanico* or *moment of a force*), it is like a twist applied to the object to make it rotate it about an axis.

$$\vec{\tau} = \frac{d\overrightarrow{M_a}}{dt}$$

Constraints

The idea is that the computations performed to simulate a physically accurate reaction to forces is something which is not so easy from the computational point of view. In real-time application we need to really optimize the speed-up process by approximating it a bit.

The physics engine to run may take longer time expected in real-time application, this because having 6DOF in Rigid Body simulation is a complex situation to manage. But there are some situations where we still want rigid body physics simulation, in certain simulation we may identify some movements some degrees of freedom which are not required.

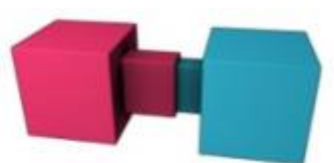
e.g., If I introduce a constraint in the physics simulation, ion way to limit the DOF from 6 to 3, then the simulation will be performed in a faster computation time.

Some constraints from robotics :

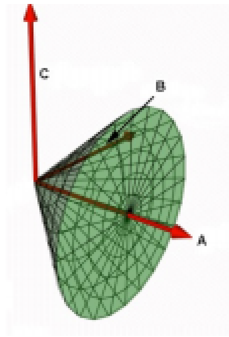
Point2point : There is free rotation on the three axes but is not possible to translate. It is used for anthropomorphic model where I want to simulate the falling of human and I want to simulate the rotation of the hands/joints during the movement, the arms must not translate (staying connected to the overall object) but free to rotate around.

Hinge : The rotation is allowed only on a single axis and there is no translation, for example a door.

Slider : The rotation is not allowed and the translation has a limited movement.



Cone : This is a special case of the point2point constraint, in this particular case the rotation is limited to a cone shape. This is usually used in the ragdoll animation.



With these constraints i can *speed-up* the **Physics simulation**, because it limits the possible transformations to a subset on some specific axis or direction.

When we have to solve the system in the **Physics Engine**, if we apply the constraints then the first things that system resolver does is to check the constraints and try to satisfy them.

If we have introduced more than one constraint then there is a policy we have to choose between two :

1. The satisfaction of one may not satisfy another.
2. The satisfaction of all constraints at the same time is not efficient.

Usually are applied relaxation methods, which will try to satisfy one at time in a sequential way, this in order to reach the final solution.

Example-1 Collision of 2 objects

I have two object colliding; we usually have two periods : one ***period of deformation*** and on ***period of restitution***.

During the collision each object apply a force on the other, we have a very short collision time and a relevant change in velocity (in a very short amount of time, this will also change the direction).

For the first object we can integrate the collision force on collision time :

$$m_1\overrightarrow{v_1}_{after} = m_1\overrightarrow{v_1}_{before} + \overrightarrow{I_F}$$

For the second object and for the **Newton's third law** we have :

$$m_2\overrightarrow{v_2}_{after} = m_2\overrightarrow{v_2}_{before} - \overrightarrow{I_F}$$

During collision each object applies a force on the other which is an impulse (the *c* scalar is a value for the impulse):

$$\overrightarrow{I_F} = c \cdot \vec{n}$$

Then we have also the so-called ***coefficient of restitution***, which is a measure of the collision elasticity. This parameter is between 0 and 1 which tells how elastic is the collision which I'm simulating, this is an approximation of real-world materials characteristics.

- ***Coefficient of restitution* = 1**, means that we have a perfectly elastic collision.
- ***Coefficient of restitution = 0,*** *means that we have a maximum loss of kinetic energy.*

We can arrange at this point everything in this equation, and we can define the coefficient of restitution has :

$$\left(\overrightarrow{v_1}_{after} - \overrightarrow{v_2}_{after}\right) \cdot \overrightarrow{n_c} = -C_r \cdot \left(\overrightarrow{v_1}_{before} - \overrightarrow{v_2}_{before}\right) \cdot \overrightarrow{n_c}$$

$$C_r = \frac{\overrightarrow{v_1}_{after} - \overrightarrow{v_2}_{after}}{\overrightarrow{v_1}_{before} - \overrightarrow{v_2}_{before}}$$

There is another possibility for getting the **coefficient of restitution**, but this applies in the specific case of an object bouncing on a fixed surface, like a ball falling on the ground. In that case the **coefficient of restitution** is given by the square root between ratio of **bounce height** and **drop height**.

$$C_r = \sqrt{\frac{h}{H}}$$

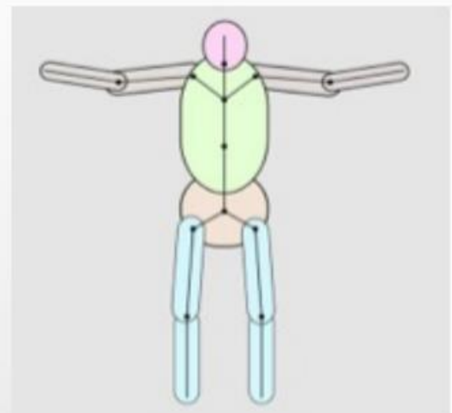
At this point I can define the **linear impulse** has :

$$\overrightarrow{I_F} = \frac{m_1 m_2 (1 + C_R)\left(\overrightarrow{v_1}_{before} - \overrightarrow{v_2}_{before}\right) \cdot \overrightarrow{n_c}}{m_1 + m_2} \cdot \overrightarrow{n_c}$$

I can use the same **initial equations** (*m1* and *m2*) to compute the velocity **after the collision.**

Example-2 Ragdoll Physics

The physics is applied to a hierarchy of collision shapes which are associated to model *bones*, joints, ,… for simulate a realistic falling (the accurate constraints used in this case are the **cone joints constraints**).



How much realism is needed?

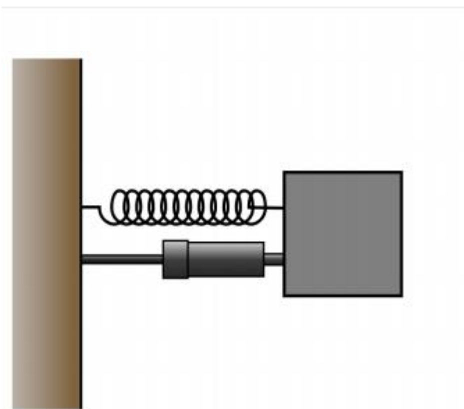
The developers decide the level of realism in base of the goal of the application. If I'm making a car simulating game i will really need an accurate Physics simulation. Since we are in real-time graphics we need to optimize everything, in case of an accurate simulation game we will need a sophisticated **ODE** (*Ordinary Differential Equation*) *solver*.

In case of more simple game, like an arcade, I don't really need an accurate Physics simulation, so i relax the overall complexity of the computation by using a greater approximation.

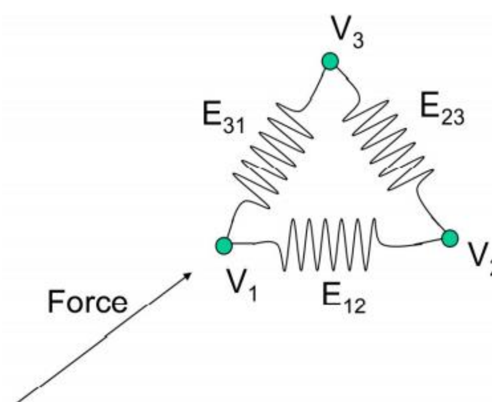
Some very complex phenomena can be approximated by using simple techniques

Example-3 Spring-mass-damper systems

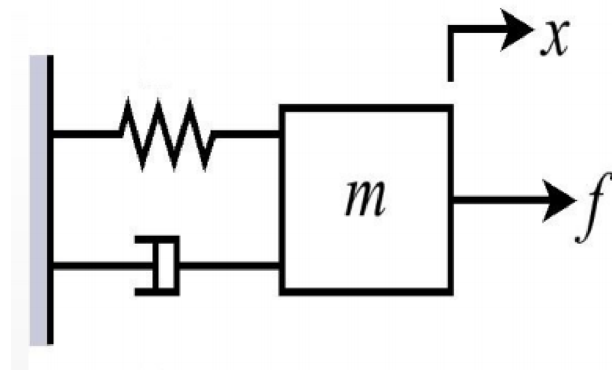
You have a mass, and you apply to them both a spring and a resistance, if you consider a mesh and every vertex is a mash and the edge is a spring, then we can simulate some phenomena on a mesh like cloth simulation or flexible body (resistance force + elastic force).



Like in the figures this is something which propagates along all the mesh, these two forces applies movement to the attached vertex and so on… This is obviously not realistic in real-world, however this is very easy to implement, and we got a way to express a complex phenomenon in the end (even if not realistic).



The problem is that is something not so easy to control, we have really to find tuning of the two forces and so we are not really apply something which has a link with real physics, this is obtained empirically. So, we tune parameters since we got an impression that this is similar to the real one phenomenon.

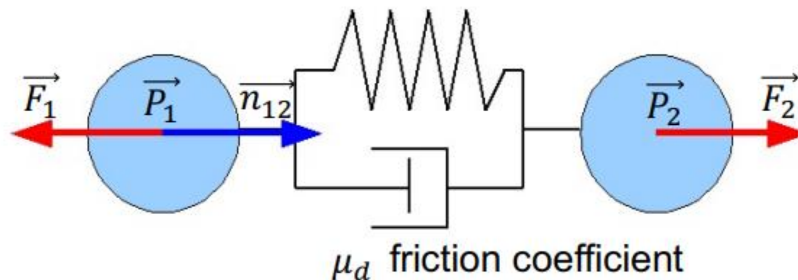


$$\vec{F_s} = -k_s \cdot \vec{x}$$

The two forces are defined in this way, we have the sum of the two where the spring force is a classical **Hooke's law**, so we have an ***elastic coefficient*** $-k_s$ which is multiplied for the displacement of the spring in *rest position* $\vec{x}$.

$$\vec{F_d} = -\mu_d \cdot \vec{v_d}$$

The resistance is the **Stoke's law** so a simplified friction force, obtained by the multiplication of the opposite **friction coefficient** μ_d with the **velocity of the spring movement** $\vec{v_d}$. By tuning the **elastic coefficient** and the **friction coefficient** i can have different effects.



$$\vec{F_1} = \left(k_s \cdot \frac{|\vec{P_2} - \vec{P_1}| - l}{l} + \mu_s \cdot (\vec{v_{P_2}} - \vec{v_{P_1}}) \cdot \vec{n_{12}} \right) \cdot \vec{n_{12}}$$

$$\vec{n_{12}} = \frac{\vec{P_2} - \vec{P_1}}{|\vec{P_2} - \vec{P_1}|}$$

If i have two balls attached with a **spring-mass-damper**, i apply one force to the one in the left and i make the spring from the rest position l to something larger. I can compute the force applied to the first ball which is the sum of **Hooke's law** and **Stokes's law**, and if the spheres have the same mass i already have the force which is applied to the other ball but in the opposite direction.

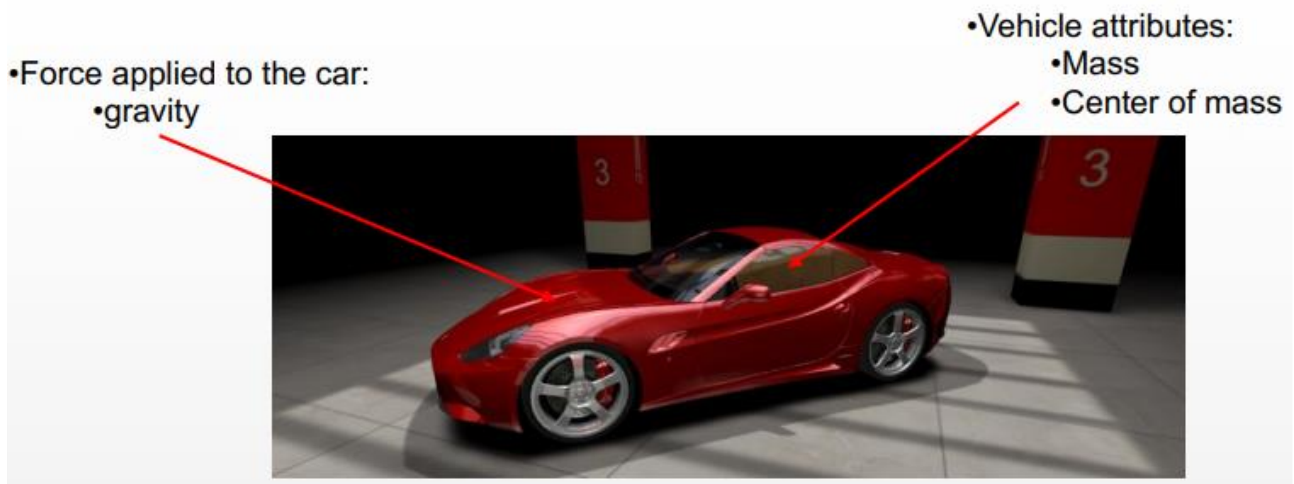
$$\vec{F_2} = -\vec{F_1}$$

This kind of simple phenomena is also used for “**control**” **forces** :

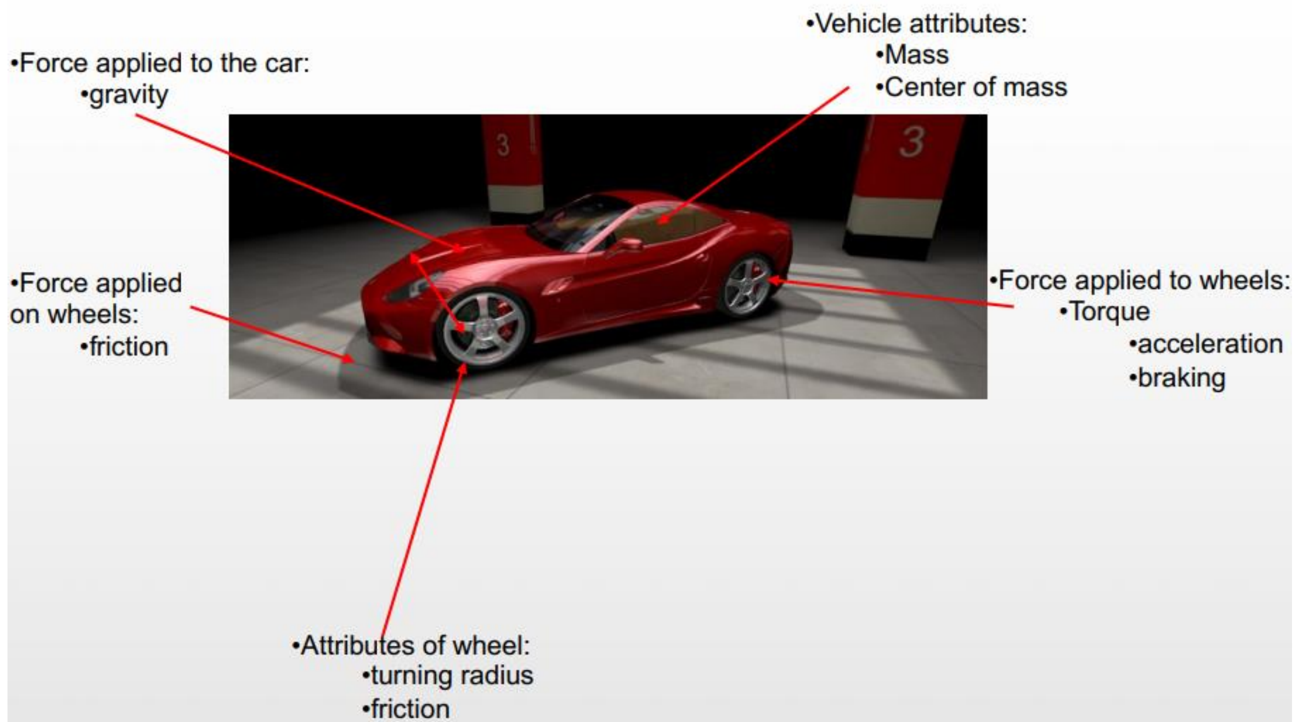
- I set a **rest position** of the spring to **0** and a very **high elastic coefficient** and **low friction**, then the this will keep 2 objects attached.
- If I set the **spring** with **rest position greater than 0** I main a minimum distance between two moving objects.

Example-4 Vehicle

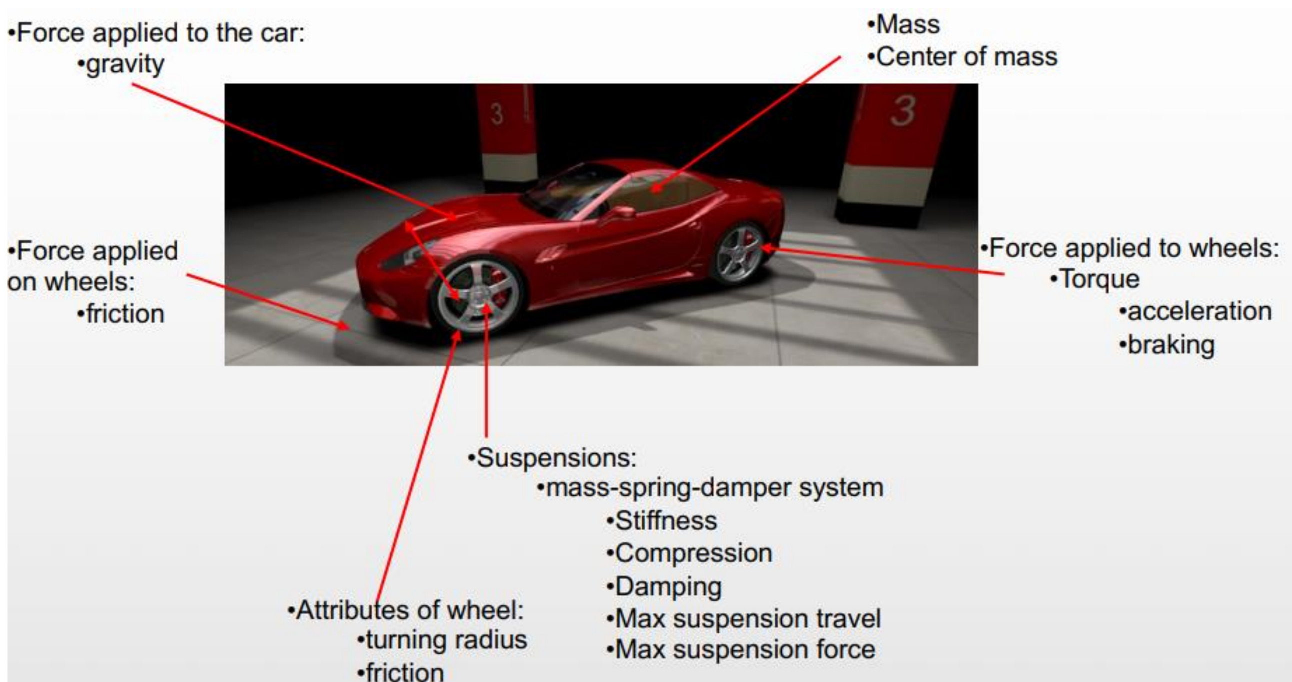
A realistic car simulator is extremely complex, a more approximated approach is obtained by applying and combining forces and methods described previously to achieve an effective but less computational expensive simulation.



Taking for example this car from **Bullet library**, I set up *gravity, mass, center of mass* of the vehicle.

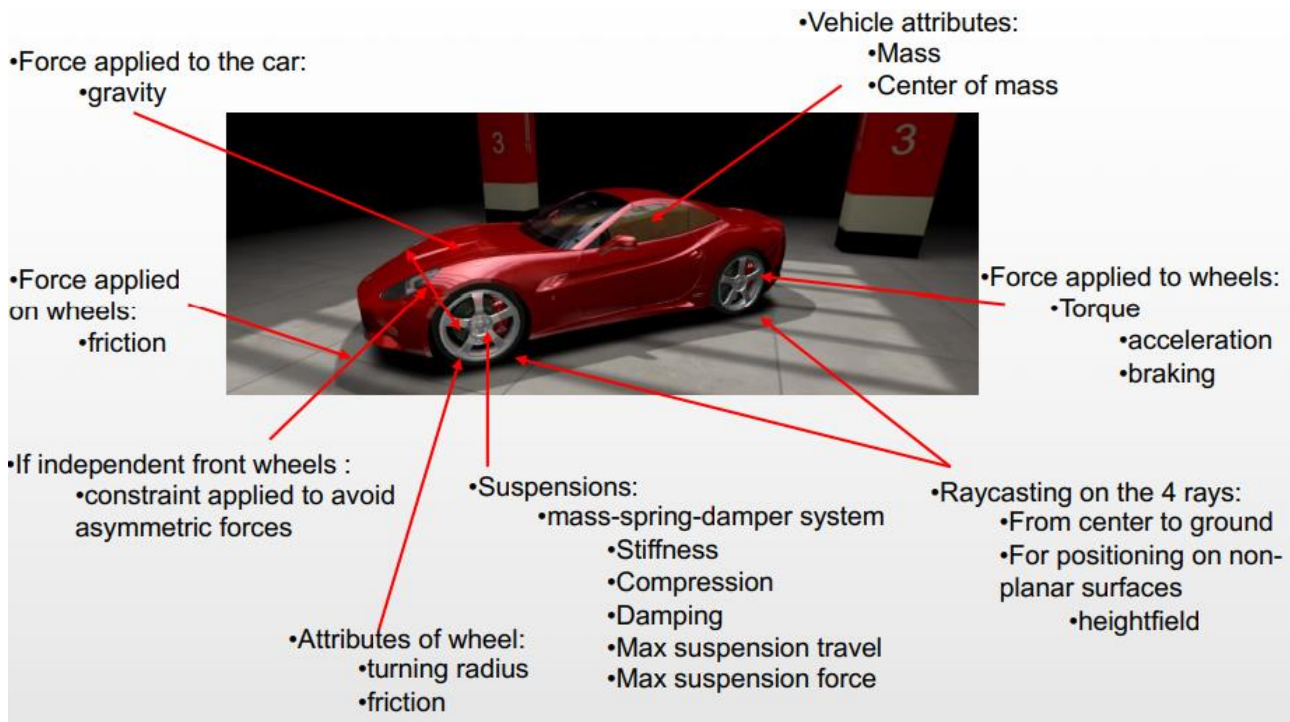


Then I setup a set of simple forces : ***friction*** on the wheels (from the ground), ***torque*** to the wheels for acceleration and braking. Then the front wheels got attributes : *turning radius* and *friction*.



Then I have the suspensions, I can apply a **spring-mass-damper system** and I can set up the following parameters :

- **Stiffness**
- **Compression**
- **Damping**
- **Max suspension travel**
- **Max suspension force**



If the **front-wheels** are independent I can apply constraints in a way to avoid asymmetric forces, and I can apply ray casting to keep the vehicle on the ground.

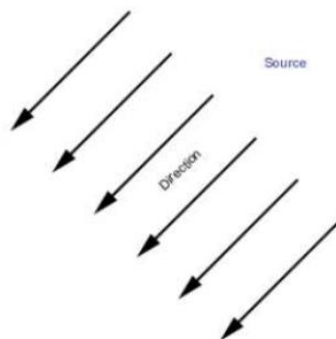
For an arcade vehicle all these parameters are considered in order to simulate physics of a car.

Light sources in Computer Graphics

A **light source** in the scene is not visible, the model of an object that contains the light is part of the scene, but the light itself no, the light is more "virtual" in respect of the object. Lights in games are made without shapes and there are different types of lights.

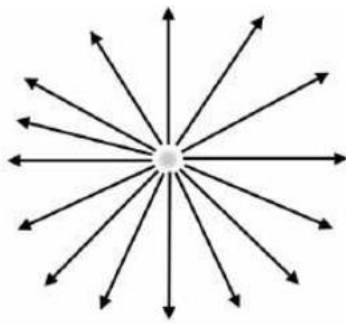
Directional

Kind of light you receive from the sun (*uniform illumination* same intensity of light in every point), this light is so distance from the scene that all the light rays hit parallel the scene. In this case you define light using a vector, it is particularly good for simulating the solar rays.



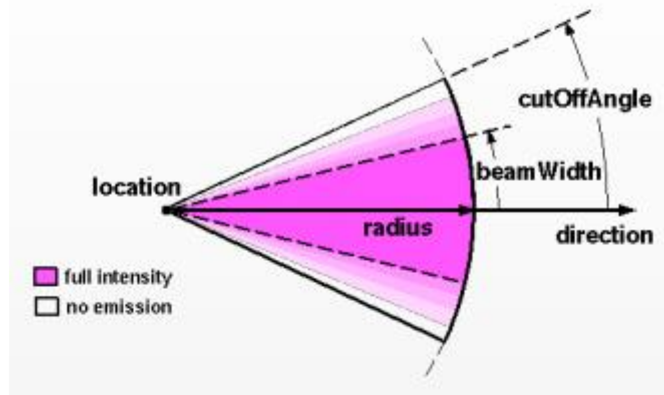
Point light

The one you use close your scene are the point lights, it is a point that emits light. There is no dimension, you define it by using a point. You can apply it a decay rate, means that the power of the light decays more as you move away from the light source.



Spotlight

It is a special case of the **point light**; it isn't a complete point but just a *cone-shaped portion* of it. Is possible to setup a **cut-off angle**, in this way it is possible to have two concentric cones in the view, where the more external is the less intensive.

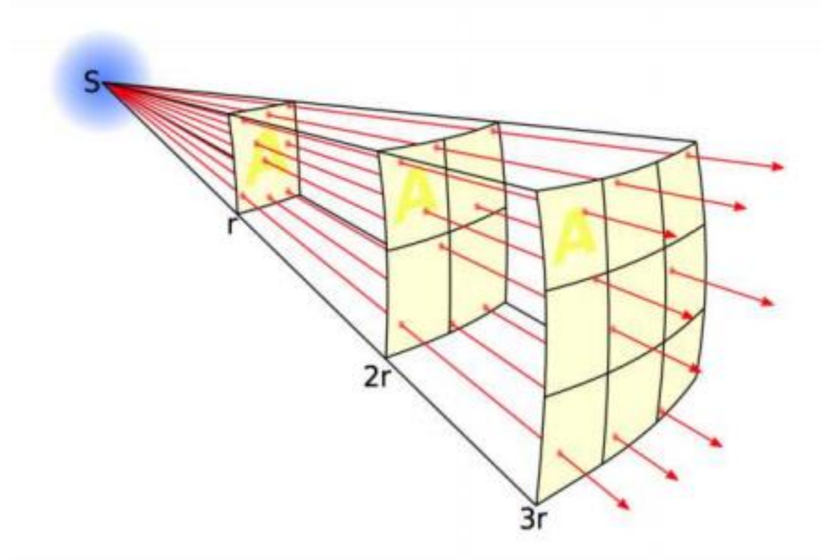


Area light

Is a light with a specific shape and size, it is usually approximated using a set of point lights distributed over the surface. The effect of all the point lights makes this final simulation. It is just a bunch of point lights.

Inverse-square law

In the real world the light decays as more I get away from the sources, so it is up to developer implement this. **The intensity of the light is inversely proportional to the square of the distance from the light source**.



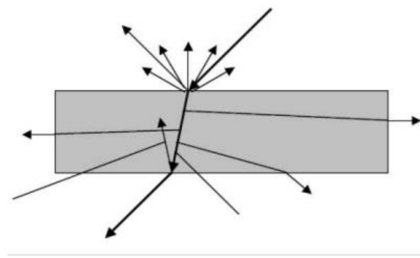
Materials

The light which has some physics properties, touches an object and the property of the light interacts with the properties of the material in such a way that our brain changes the perception of that light.

There are three possible interactions between light-material :

- **Reflection**
- **Absorption**
- **Transmission**

For the law of conservation of energy, the incident light is equal to the sum of the **reflected light**, **absorbed light** and **transmitted light**.



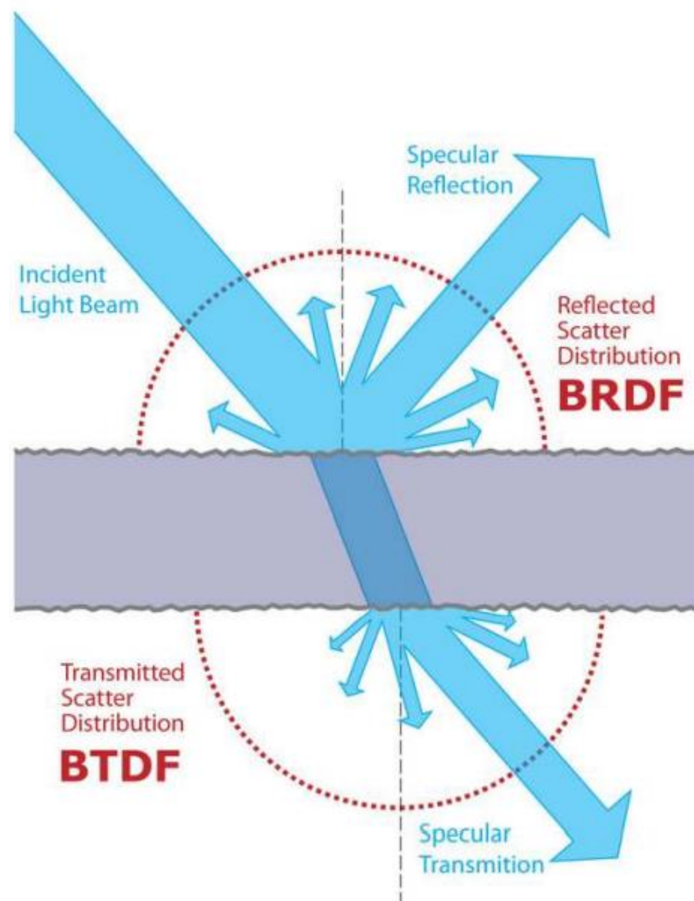
In more complex approximation I can forget about the law of conservation of energy, anyway this is a good law to follow for respect Physics Simulation.

BRDF, BTDF, BSDF

Bidirectional Reflectance Distribution Function is a function which describes how light is is reflected from the material. For example, a material got a BRDF parameter that describes how the material itself interacts with the light, then the function tells that the light coming from a certain angle (*incident light beam*) of incidence to the material will be reflected (*reflected scatter distribution*) in a certain way.

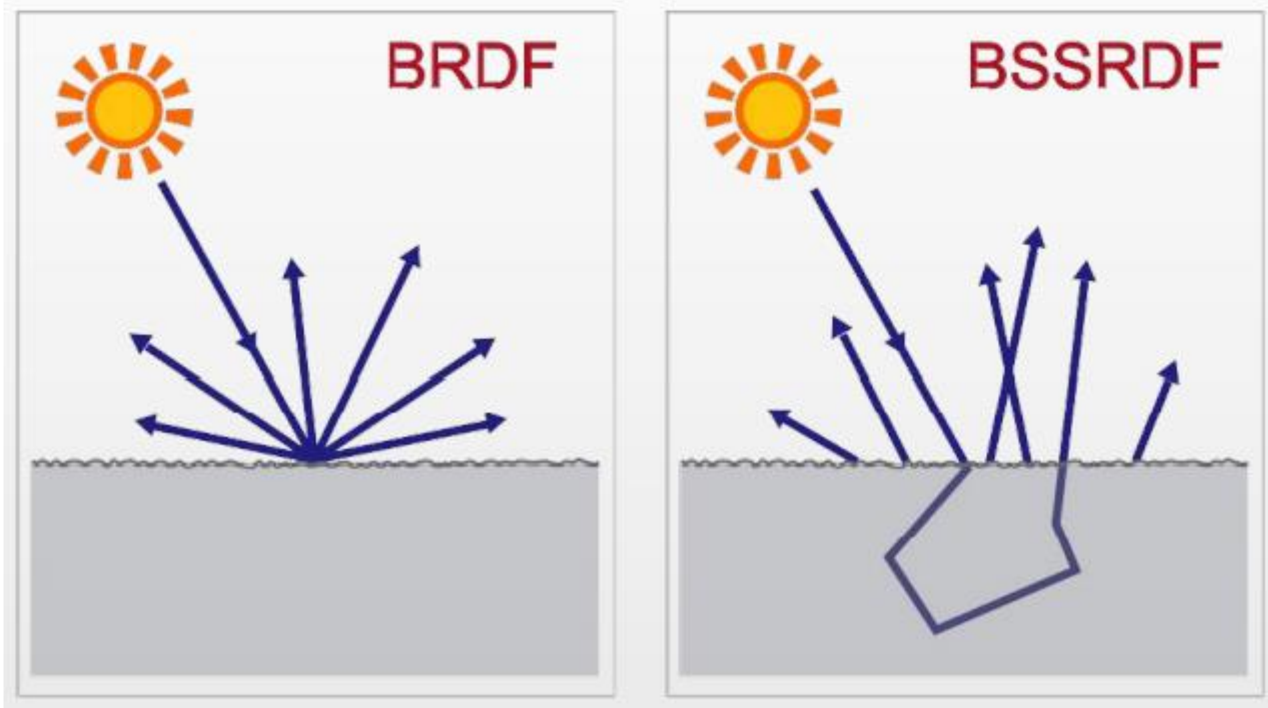
In the case you want to describe the **refraction** (*transmission*), you will have to use **Bidirectional Transmittance Distribution Function** is a function which does the same of BRDF but for the light which goes inside the material, and it is scattered in particular way.

The idea was to use the BRDF and BTDF separately, but with increase of the computational powers and the starting to *refraction* techniques, they wanted to use a term for indicate both the phenomena together. This is the **Bidirectional Scattering Distribution Function**.



BSSRDF

Bidirectional Surface Scattering Reflectance Distribution Function, it is introduced for the light which is scattered inside the material can exit from the surface not in the same incident position. The light is bounced inside the material (*diffraction*).



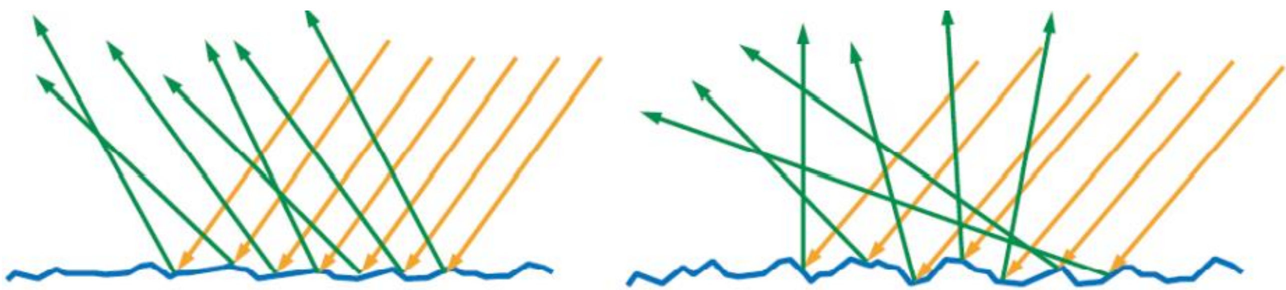
Physically based BRDF

Where "physically based" means that it si more physically correct respect at the standard way.

The assumption usually is that the material can be described at microscopic level as composed as microfacets.

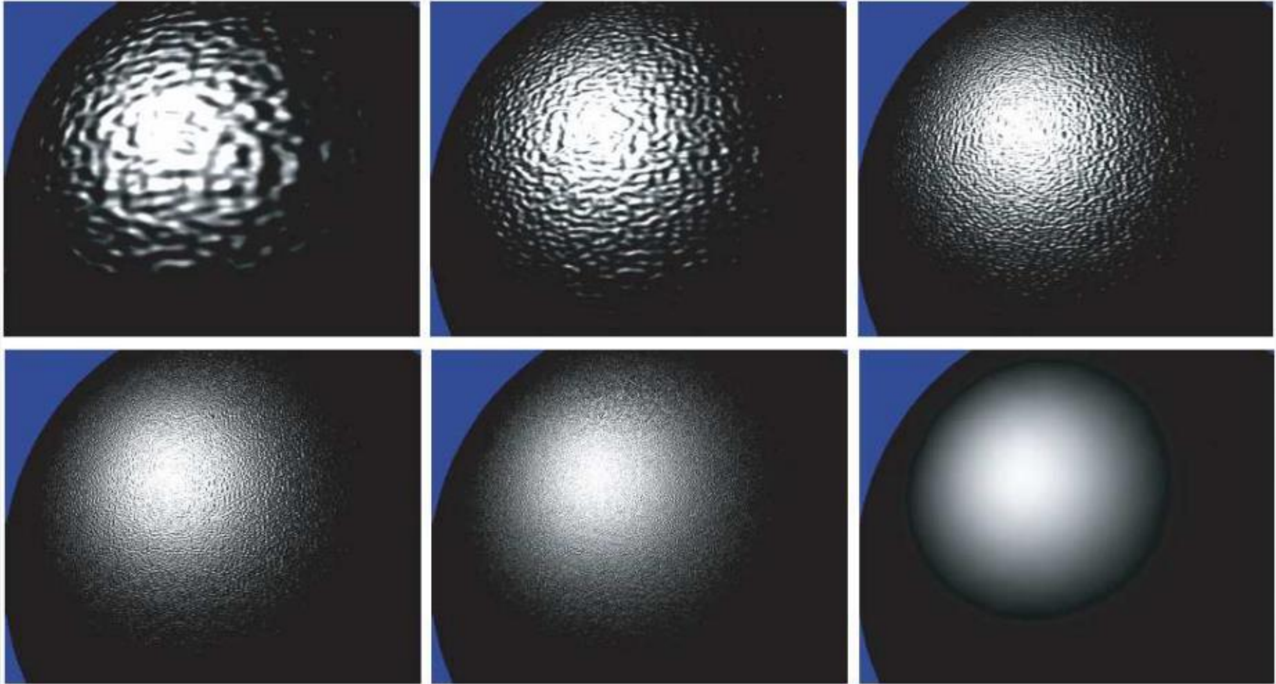
For example, if I use a microscope and I look to the surface of the material I can see **microscopic facets** (they are visible only at microscopic level or smaller) oriented in a different way, like very small mirrors, and the final reflection of the light is given how random or ordered is the orientation of this very small microfacets.

This set of small facets, there is a high number of this i cannot simulate them geometrically I can simulate them statistically. I assume that I can use statistical distribution that describes describes the overall orientation of the facets completely random or oriented in a certain direction, and I use statistical distribution to simulate the characteristic of the material.



Rougher the orientation of the faced is more scattered the light is in several direction.

Different level of microfacets on a material



BRDF

The BRDF defines how an incident light from an incoming direction is **being reflected** along an outgoing direction.

It is possible that the material is reflecting in just one direction (specular material), the BRDF can say that for a portion of 90% of the reflection the result is 0 and in the remaining portion the reflection is 100%.

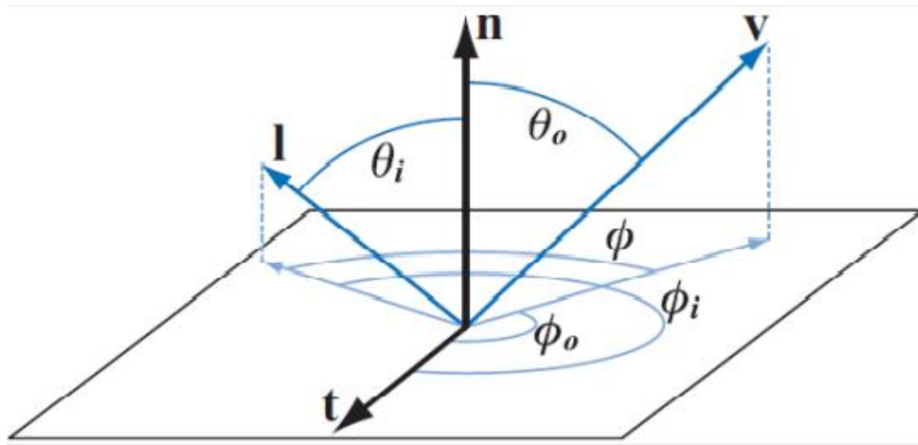
The BRDF is like giving you a map of all the possible direction, the original description of the BRDF is the function of :

- *Wavelength*
- *Position of the surface*
- Directions for the *incoming* and *outgoing* direction.

We have the formula with two parametric values that describe a ratio between two photometric physics measures ***radiance*** and ***irradiance,*** mainly it is the ratio between the ***incoming*** and ***outgoing light,*** $\boldsymbol{L_0}$ and $\boldsymbol{E}$.

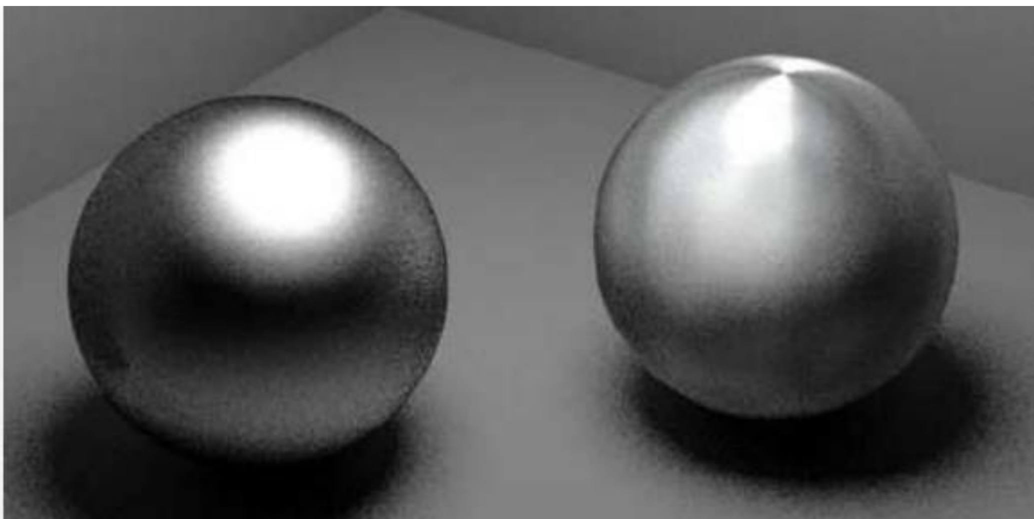
$$f_o(p, \theta_i, \Phi_i, \theta_o, \Phi_o, \lambda) = \frac{dL_o(p, \theta_o, \Phi_o, \lambda)}{dE(p, \theta_i, \Phi_i, \lambda)}$$

BRDF is **isotropic if the formula is valid for all the orientation**, if the orientation around the plane it's uniform regarding that direction.



Anisotropic BRDF

The BRDF is **anisotropic** if the behavior of the reflection is not uniform in all the direction, but it changes when the surfaces is rotated along then normal (the sphere on the right).



As you can see we have a certain area where the sphere is brighter while the other two area they are dark. So, the anisotropic BRDF means that isn’t uniform in all orientations.

Where do I found the BRDF for materials ?

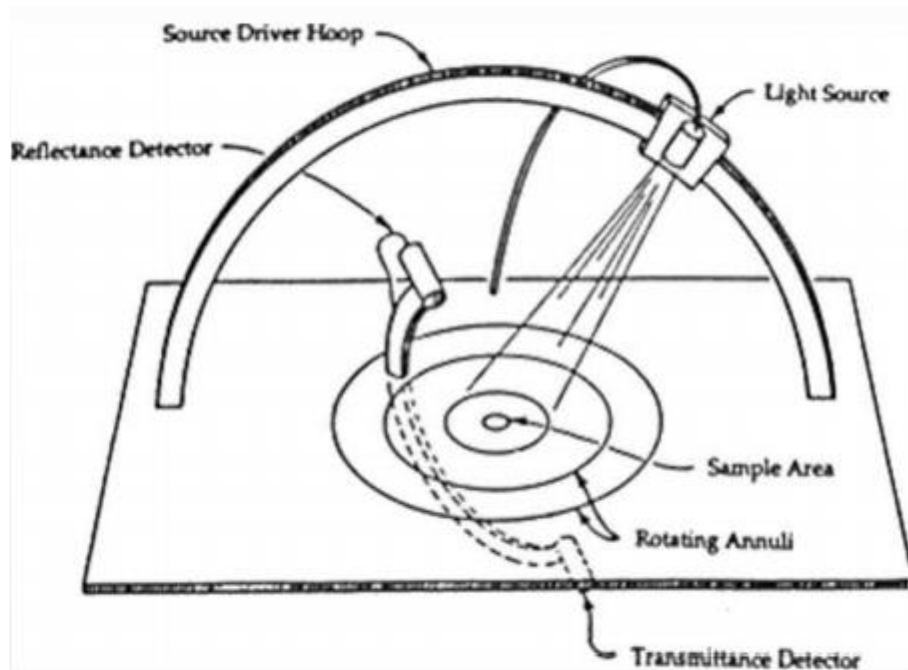
for different material aren’t known for all possible material, they are usually defined analytically so they are approximated. Then I can use the parameters and different parameters lead to an approximation of the final result of the material.

This approximation can be based on some physics criteria, or they are mainly based on an empirical model.

The BRDF can be measured with the **gonio reflectometer**, a device specifically made for measuring the BRDF.

The device consists of a light source illuminating the material to be measured and a sensor that captures light reflected from the material. The light source should be able to illuminate and the sensor should be able to capture data from a hemisphere around the target. The hemispherical rotation dimensions of the sensor and light source are four dimensions of the BRDF.

You have very large set of data of all the possible lights and direction (where *gonio* came from), object observation and object rotation. With all this data you can reconstruct a numerical method which gives you the BRDF.



Rendering equation

This process it is based on the rendering equation proposed by **Kaija** in **1987**.

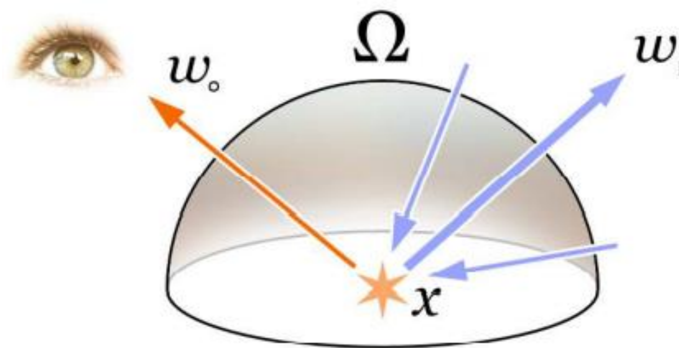
This equation describes generally **light-material interaction** for every point in the scene and for every incident light direction.

This is based on the law of concentration of energy, for every position and direction of the scene, the **outgoing light will be equal to the sum of emitted light** (light on the same surfaces of the object, most object doesn't have it) **with reflected light**. This is considered by the recursive formula for multiple reflections and refractions. In **real-time graphics** you don't use the recursion implementation of this formula, in **path-tracing** yes.

The integral is considered over the interval of the hemisphere of the incidence point *x,* of the BRDF multiplied by the incoming light multiplied by the cosine of the incoming direction and the normal. The integral considers all the possible reflection and refraction of light,

$$L_o(x, \omega_o, \lambda) = L_e(x, \omega_o, \lambda) + \int_{\Omega} f_o\,(x, \omega_i, \omega_o, \lambda) L_i(x, \omega_i, \lambda)(\omega_i \cdot n) d\omega_i$$

This isn't obviously an appliable equation in practice, but it gives a nice explanation of the big picture.



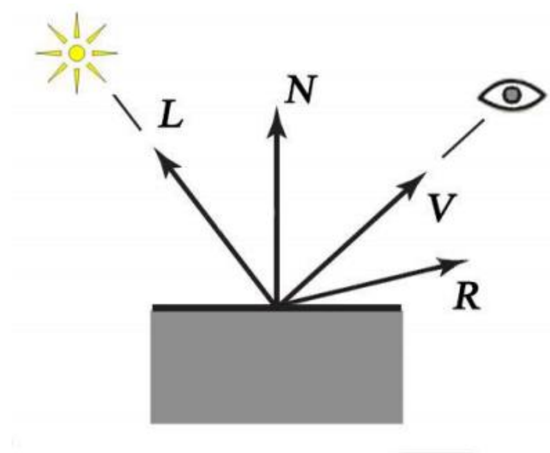
Illumination models

In computer graphics we use illumination models which are mathematical models which approximate the results of this equation.

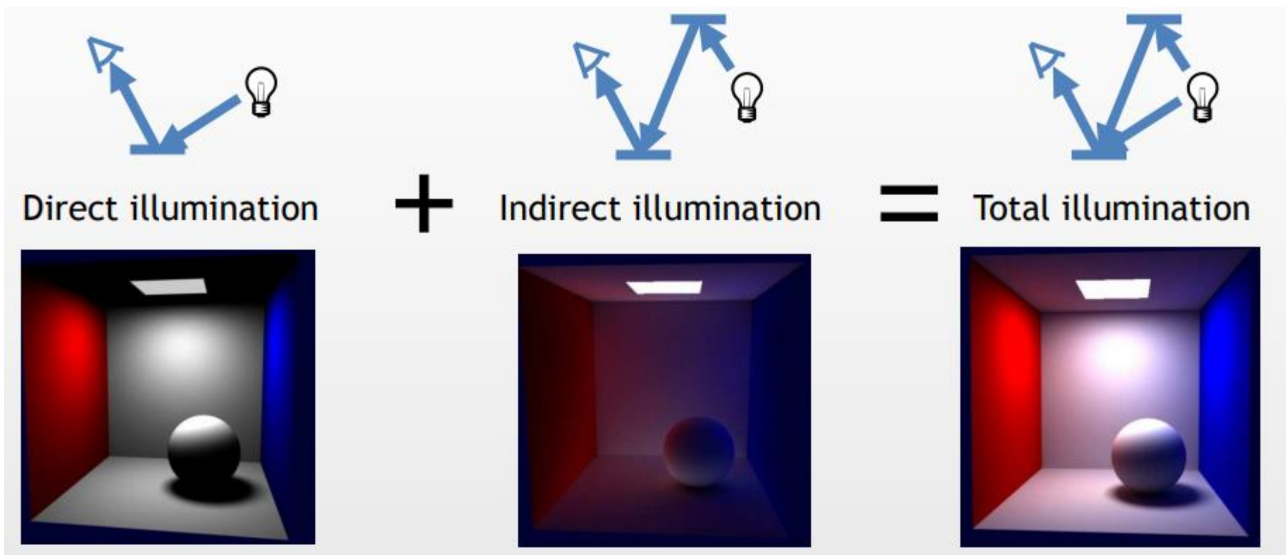
The amount of the computation is given by the complexity I want to use in my application and also the nature of the computer graphics in my application. In sense that, if the application is rendering for a movie i can use an illumination model which is an approximation but is something that will run for hours, so it must be really specific.

The illumination models can be divided in :

Local illumination models, this are used in real-time graphics, we computer only the direct illumination, we consider each point separately and you usually apply an illumination model which is a function of a certain limited set of vectors. These vectors are mainly the position, normals ,direction from the point to the light source (L), direction from the point to the camera (v), the reflection vector.



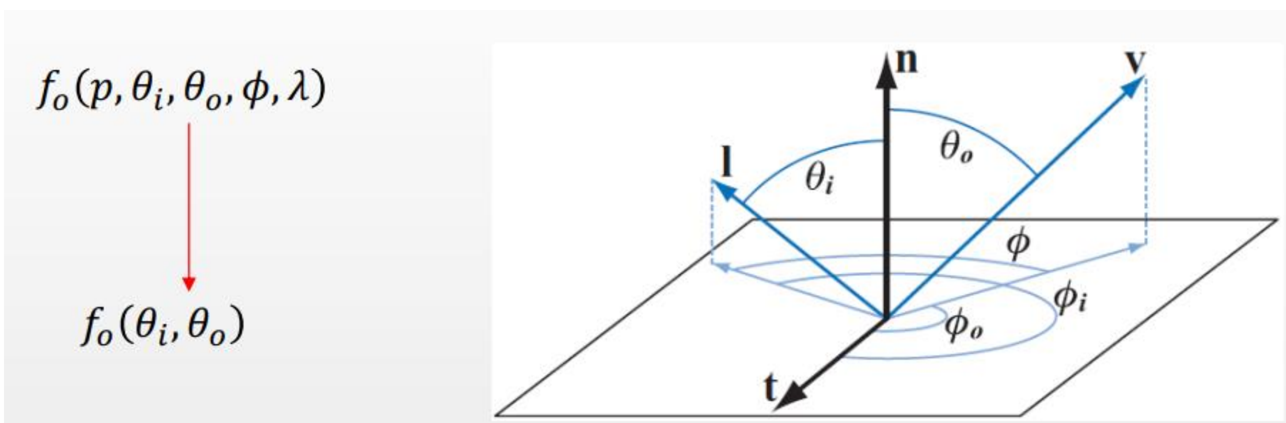
Global illumination models, this are used for the movies, where are considered both direct (illuminated the object directly) and indirect (lights reflected from other objects which contributes to the illumination of the objects) illumination.



BRDF Simplification

Mainly what we do is a more relevant approximation of the BRDF. The **BRDF** for standard real-time graphics is a simplified version of the original BRDF model where I don't consider the whole spectrum of light but only the **RGB**, and I consider the material as **uniform** (BRDF is the same on all the points of the object) in all surface point, and usually I consider the material **isotropic**.

So, I reduce the BRDF to just two parameters: incident light and outgoing light.



Empirical BRDF

This are used when the computational power wasn't so high as nowdays. Some of these models are current still used in **computer graphics** for simple objects, the requirements for these models (even if they are empirical) are :

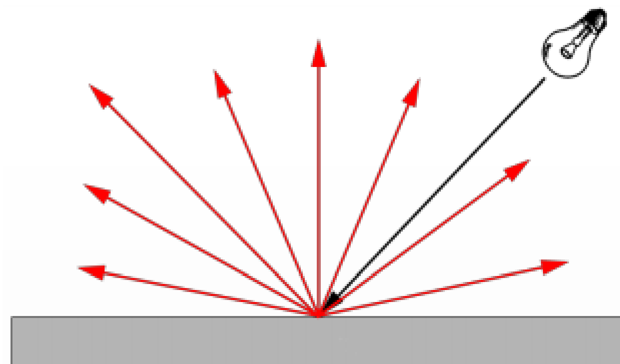
- **conservation energy**
- **separation between diffusive and specular components**
- **intuitive parameters**

Simplified models of light reflection

Diffusive

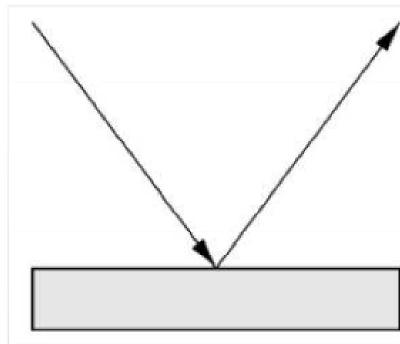
Diffusive is when the light is reflected in a **uniform way in all the direction**, you consider the hemisphere and you reflect the light in all possible direction. The only parameter which rules the

amount of the reflection is the **angle of incident light**, the more is perpendicular the higher is the reflection (vice versa more is tangent to the surface). It **isn't dependent** from the **viewpoint**.



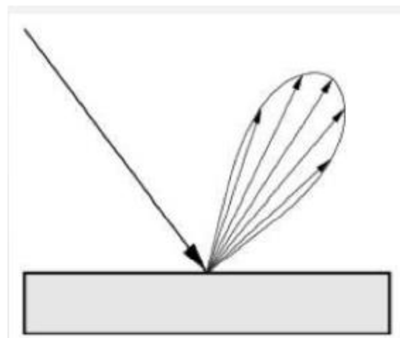
Specular

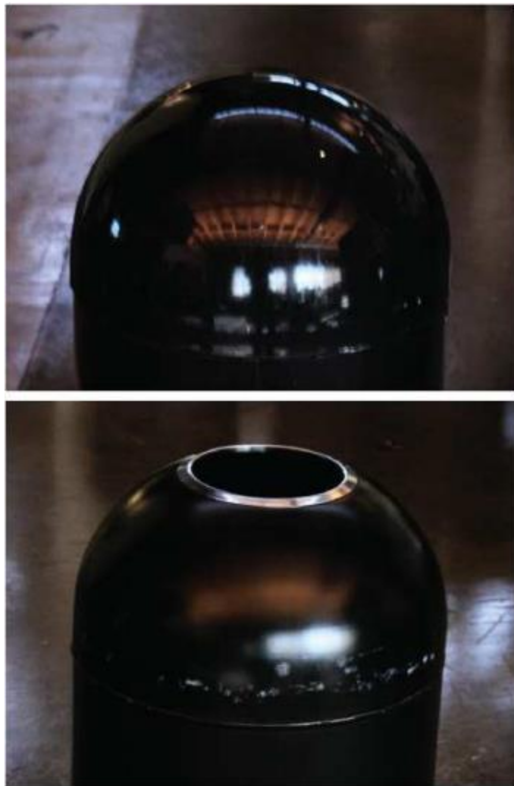
The light is reflected in only a single direction, in this case the view position is relevant. We aren't talking about a reflection of a mirror but of a material like metal or glass.

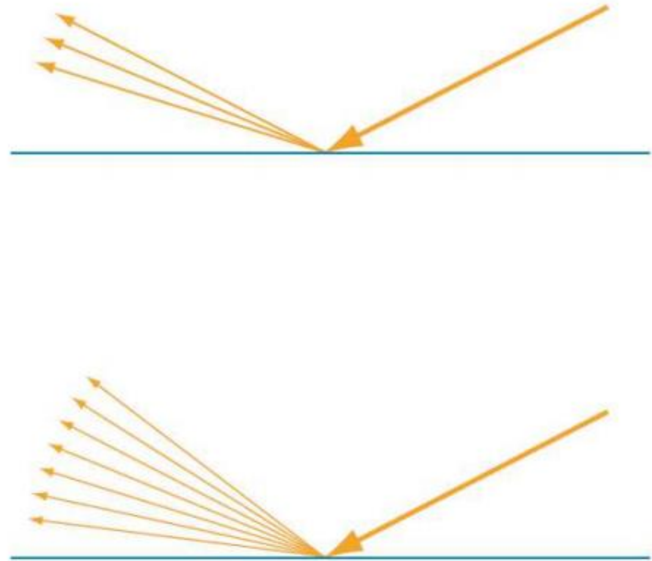


Glossy

We have a main direction which is a specular direction, but then the light is reflect as a **lobe**. The larger is the lobe of reflection more blurred is the specular reflection.







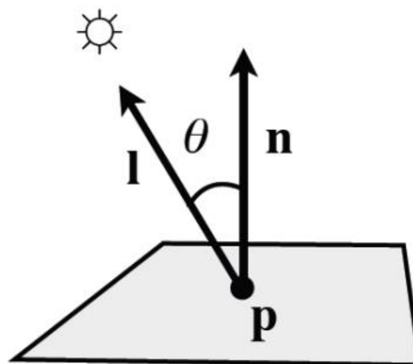
Lambert's cosine law

For the **diffusive reflection** the only parameter is the angle of the **incident light**, then we define this using the **cosine law**. The light

The first illumination model is the lambert's illumination model which is a model just for diffusive material (no plastic, no metal but more like wood material). There are two version of this model, one for trigonometry and one with vectors.

$$L_d = k_d L_i cos\theta = k_d L_i (l \cdot n)$$

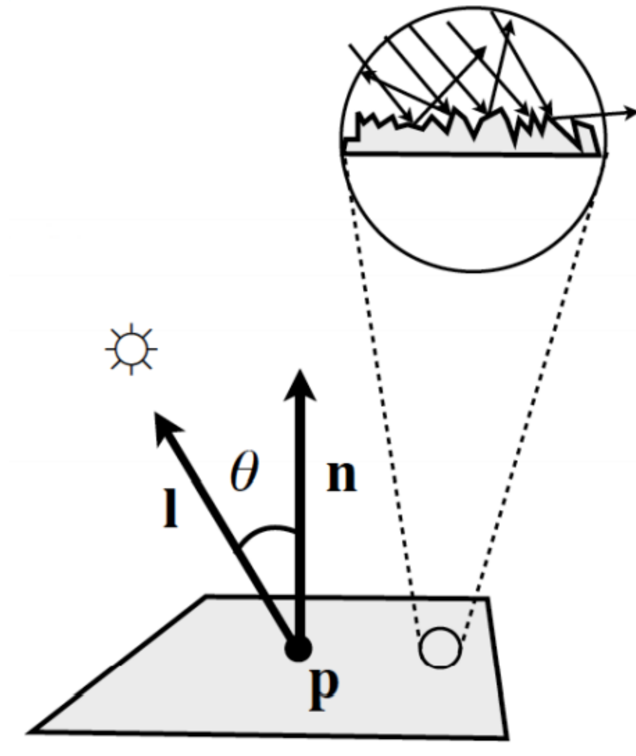
Digitare l'equazione qui.



$\boldsymbol{L_i}$ represents the incoming light from direction $\boldsymbol{l}$, $\boldsymbol{k_d}$ is the **diffusive reflection coefficient**.

This is for one light only, if we have more lights we have to do a sum and consider all the directions.

Even if it is really empirical if we want to link the **BRDF** to the **microfacets approach** (**PBR** approach), at microscopic level this **material uniform reflection** is a very **rough surface.** Because the light is reflected in every possible direction this means that all the microfacets are all oriented in all different way.

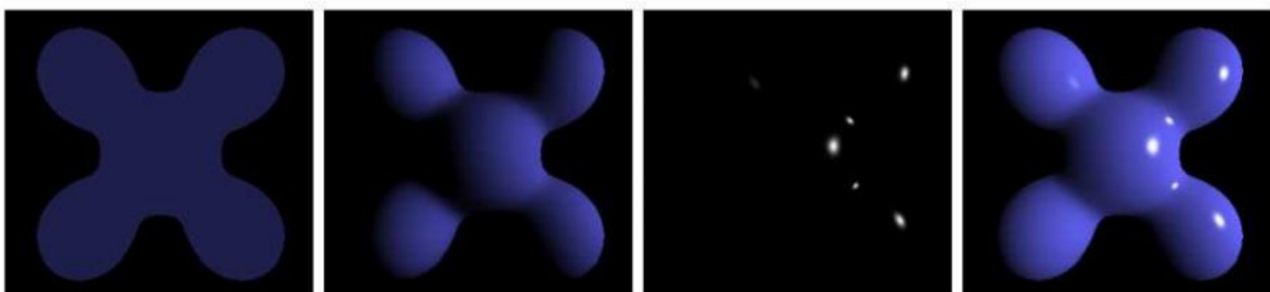


Phong reflection model 1975

Phong introduced the Phong reflection model, an empirical model used for considering also the specular material like *plastics*, *metals*, … it is not the most used (that is the **Blinn-Phong**). It has become a standard for many years because it was easy and possible to compute them on the graphics hardware that was available.

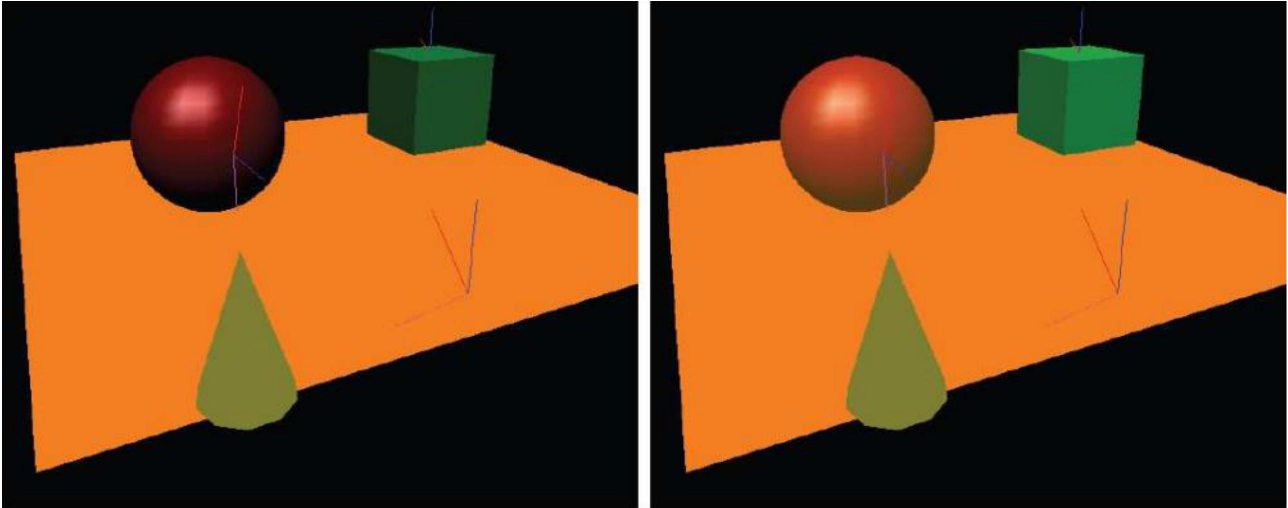
The reflection on a surface is obtained by the sum of three components :

- A **diffusive** one using the lambert model.
- A **specular** one for the "highlight" part of shiny materials.
- An **ambient component**, which approximate the global scattering of the light in the scene.



Ambient component

The idea of Phong for ambient component is to introduce a small constant to simulate some kind of **indirect light approximation**, the problem with that is given by the fact if you leverage too much this value you get a wrong *flattened* effect. This is a very rough approximation of indirect light.



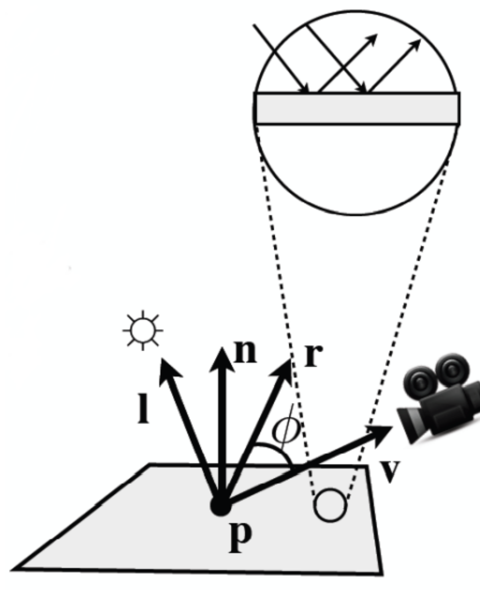
Specular component

Just for the "highlight" it is used this particular formula based on the cosine.

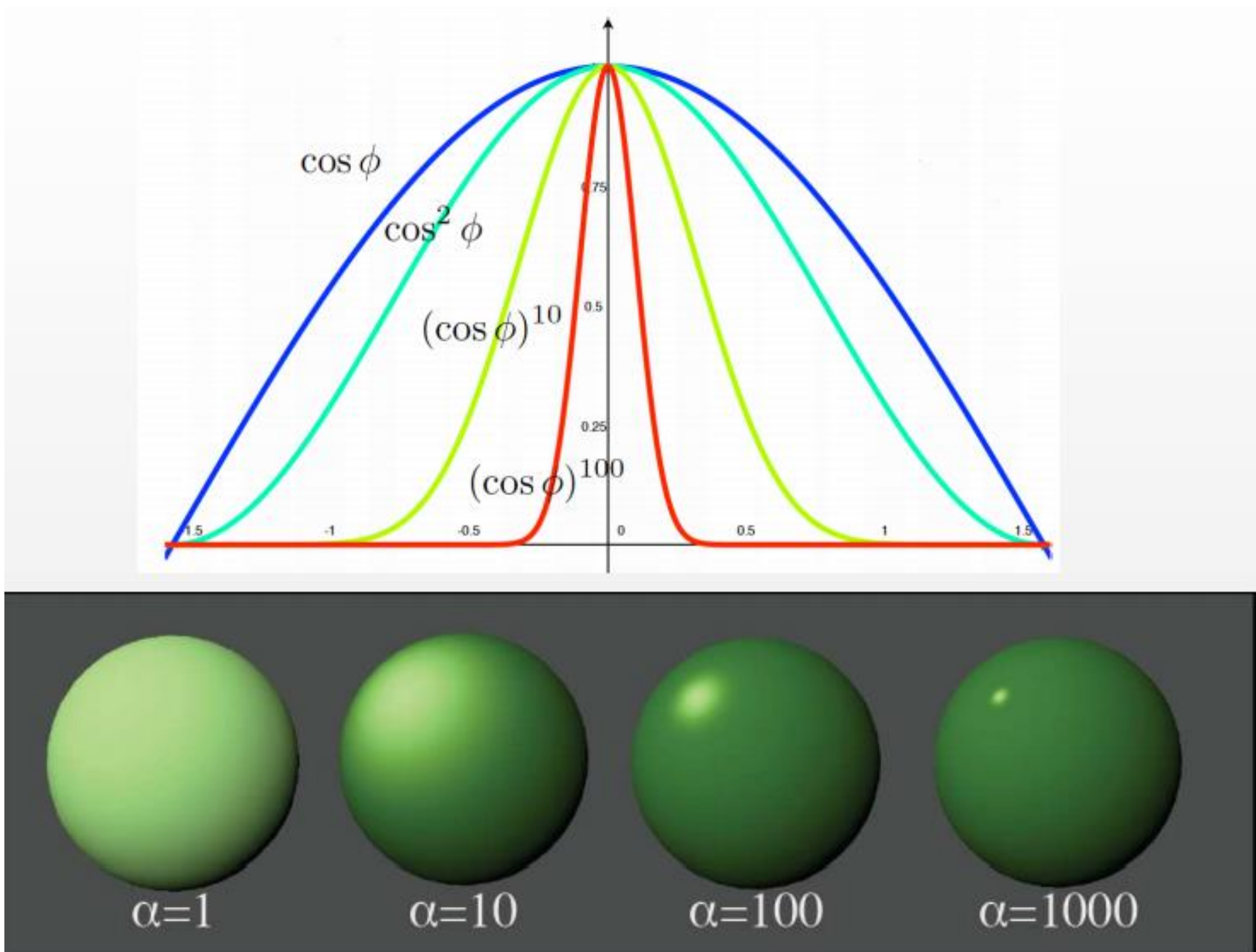
$$L_s = k_s L_i (cos\Phi)^{\alpha} = k_s L_i (r \cdot v)^{\alpha}$$

The reflection vector *r* is the mirrored version of the incident light vector in respect to the normal, this value is powered to an empirical element alpha for determine the shininess of the object, in fact is called *shininess coefficient*.

If we link this to the microfacets approach, then the specular reflection will be very smooth in the Phong, because the directions are uniform along the perfect specular direction, this means that the microfacets are oriented in the same way.



The change in value of alpha will change the graphic effect, usually between 100 and 500 we have metals, and values under 100 means that the highlight is very large.



The whole Phong model

It’s an extremely approximated version of the rendering equation, and it is valid for **just one light**. It considers the emitted light for surfaces simulating the light sources,

$$I = L_e + k_a L_a + L_i(k_d cos\theta + k_s(cos\Phi)^\alpha) =$$
$$= L_e + k_a L_a + L_i(k_d(l \cdot n) + k_s(r \cdot v)^\alpha)$$

In case of more lights, we have another formula which is more complete because it considers a sum for all the lights and there is also the decay rate from the **inverse square law** :

$$I = L_e + k_a L_a + \sum_{j \in lights} f_{att} L_i\left(k_d(l_j \cdot n) + k_s(r_j \cdot v)^\alpha\right)$$

It’s an empirical model and if you think to it, **linking to the microfacets approach** it is a very strange idea the material is **at the same time rough and smooth**.

Having together a **diffusive** and **specular** means that we can visualize the **BRDF** has a volume, this volume describes the direction and the intensities of the **overall reflections** (final result of the **BRDF**).

Phong	$\rho_{ambient}$	$\rho_{diffuse}$	$\rho_{specular}$	ρ_{total}
$\phi_i = 60^\circ$				
$\phi_i = 25^\circ$				
$\phi_i = 0^\circ$				

- The **ambient** is a simple and **small hemisphere**, this because I simply add something in all the direction.
- The **diffusive** is a **hemisphere** it means that I reflect in all the possible direction, the radius of this sphere is given by the cosine between normal and incident light.
- The **specular** is a **longer lobe** that points to the direction of the specular reflection, which changes in based of the direction of the vector for **incident light**.

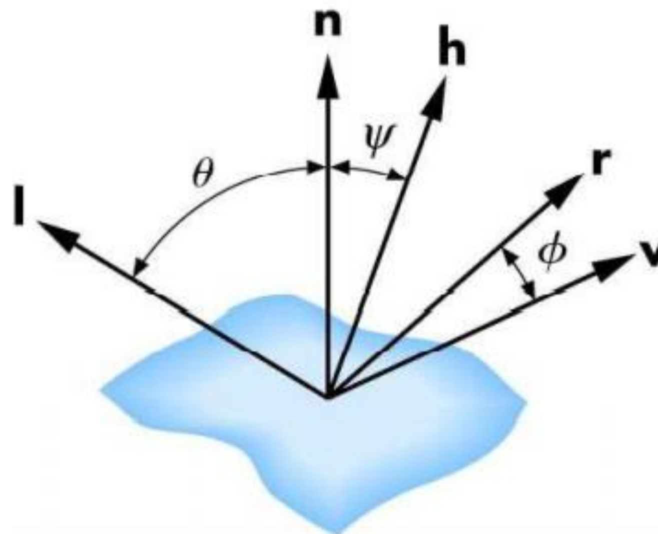
If we put everything together we have the result of the BRDF model, this strange volume is visual description of the BRDF (all the possible directions of the reflection and the intensity).

Blinn-Phong model 1977

The real **most used illumination** model is the **Blinn-Phong** model which is heavily the same but with **an optimization**. Blinn is another very famous scientist which has contributed a lot in computer graphics.

It changes the specular component part, it doesn't consider anymore the angle between $\boldsymbol{r}$ and $\boldsymbol{v}$, but it uses a new vector called $\boldsymbol{h}$ (*half vector*). This new vector is halfway between $\boldsymbol{l}$ and $\boldsymbol{v}$, and in the **new formula** will used the **angle ψ** between $\boldsymbol{n}$ and this new vector $\boldsymbol{h}$ instead of the one between $\boldsymbol{r}$ and $\boldsymbol{v}$ $\emptyset$.

$$I = L_e + k_a L_a + \sum_{j \in lights} f_{att} L_i \left(k_d (l_j \cdot n) + k_s (n \cdot h_j)^{\acute{\alpha}} \right)$$



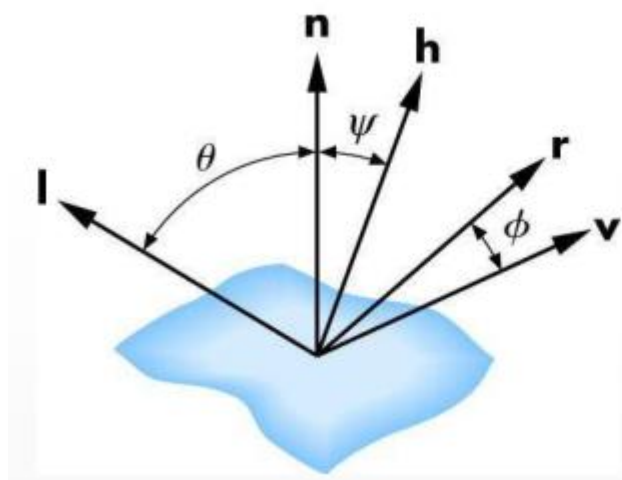
The half $\boldsymbol{h}$ is really simple to compute is just the sum between $\boldsymbol{l}$ and $\boldsymbol{v}$. The angle $\boldsymbol{\psi}$ is always **less or equal 90°**, in this way you can avoid **checking for negative dot product**, and it is easier to calculate then Ø because you can avoid calculating $\boldsymbol{r}$. Since $\boldsymbol{\psi}$ cis smaller then , there is the need for a different value of **shininess** $\acute{\alpha}$.

It has been proposed in **1977** (it has been implemented when the computational hardware wasn't like nowadays, and they had to care for everything). The techniques was developed for **optimization reasons**. However, this is the most used illumination model.

Physically based BRDF models - GGX

Or in this specific case we should talk about physically based BSDF, because we are considering something which manage also the refraction. At a certain point when the computational power and the possibility of the graphics programming allowed to develop our own illumination models and to performs in the shaders, research suggest finding more complicated illumination model.

This new model has been proposed :



$$\frac{F(\boldsymbol{v},\boldsymbol{h})D(\boldsymbol{h})G_2(\boldsymbol{n},\boldsymbol{l},\boldsymbol{v})}{4(\boldsymbol{n}\cdot\boldsymbol{v})(\boldsymbol{n}\cdot\boldsymbol{l})}$$

It is based on the model we have already seen; it relies on the same background but for the specular component is absolutely more complex.

The specular component is based on three values :

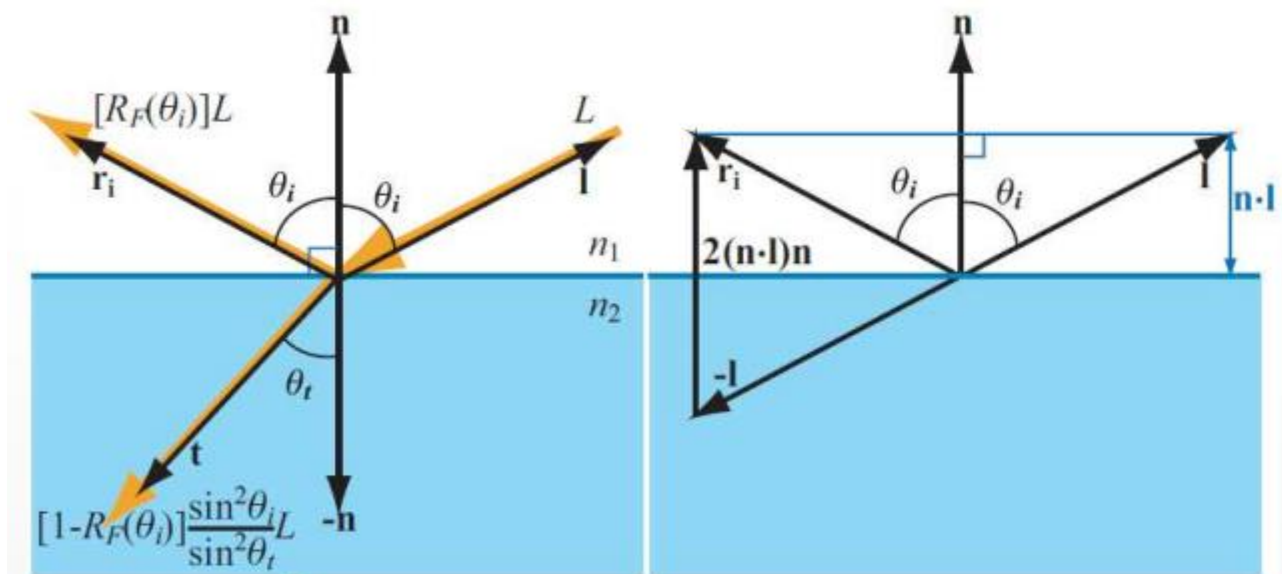
- *Fresnel reflectance (F).*
- *Microfacets distribution* (D, overall description of the kind of reflection, rought or smooth).
- *Geometry attenuation (G).*

This equation has been proposed by **Cook-Torrance 1981**, however in the years has been used to enhance the original proposal of Cook-Torrance and actually it has become a sort of template, where different F, D and G are being sued to enhance the original model.

What will be illustrated right now isn't the original model, but a template called **GGX**, which is the most used right now for **physically based materials**. They mainly change the D and G factor, the F remains almost the same.

Fresnel Equations (F)

These equations come from physics; they are equations proposed for describing the behavior of the light when the light moves between two media with two different **refractive indices**. The light is incoming from the right $\boldsymbol{L}$, part of the light is reflected, and part of the light is transmitted inside the material (*water*), and the direction of the transmission is given usually by the ratio of the **two refractive indices**.



Part of the light is reflected and another part of the is trasmitted inside the material, **the direction of the material is given by the ratio between the two refraction indices** which are typical parameters of the two media.

For the law of the conservation of energy the sum of the reflected and transmitted light is equal to the incoming light $\boldsymbol{L}$ ().

The Fresnel's equations says that the **transmitted light** (or ***refracted***) is given by :

$$1 - R_f(\theta)\frac{\sin^2\theta_i}{\sin^2\theta_t}L_i$$

Where $\boldsymbol{R_f}$ is the reflection coefficient, which is a parameter that describes **how much of a wave is reflected by an impedance discontinuity in the transmission media.**

While for the **Snell's law** we have this equivalence :

$$n_1 \sin(\theta_i) = n_2 \sin(\theta_t)$$

With this equivalence I can change how I can retrieve the **transmitted component** which is no more based on the sines of the angle and use two known parameters the **refraction indices**.

Given this equivalence now I can simplify the **Fresnel's equation** using the **Schlick's approximation**. We know that for the Fresnel's equation we need to calculate and to store samples **between 0° and 90°** (from the incoming L light quadrant). Where the $\boldsymbol{R_f}$ coefficient, is the amount of the reflected light for every possible angle of incidence, so I need to store all the value of Rf for every angle usually between **0°** and **90°.**

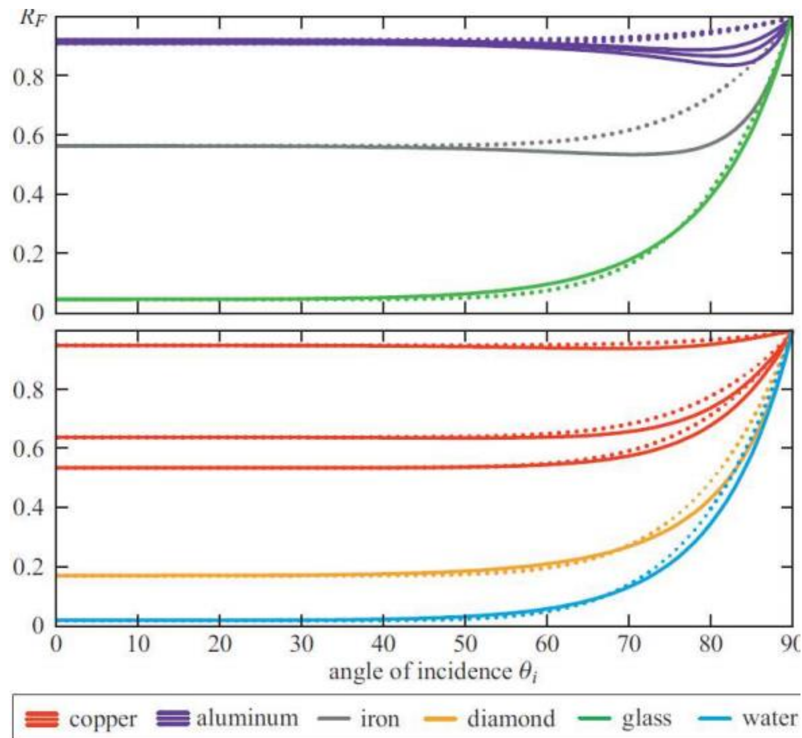
The Schlick's approximate the $\boldsymbol{R_f}$ by using only the values at **0°.**

$$R_F(\theta_i) \approx R_F(0°) + \left(1 - R_F(0°)\right)\left(1 - cos(\theta_i)\right)^5$$

Given this we can simplify even more, even if this one depends at the value at 0° we still need that value. I can simplify because the value of $\boldsymbol{R_f}$ on **0°** degree **is a determined ratio that depends on refraction indices that we have.**

$$R_f(0°) = \frac{\left(1 - \frac{n_1}{n_2}\right)^2}{\left(1 + \frac{n_1}{n_2}\right)^2}$$

And also, in the shaders often you use this angle between $\boldsymbol{v}$ and $\boldsymbol{h}$ instead of the $\mathbf{cos}\ (\boldsymbol{\theta_i})$, the ratio is computed between the refraction indices, and with that you can compute the simulation of the **Fresnel's equation.**



The only materials where the approximation is a bit rough is aluminum and iron.

Microfacets distribution (D)

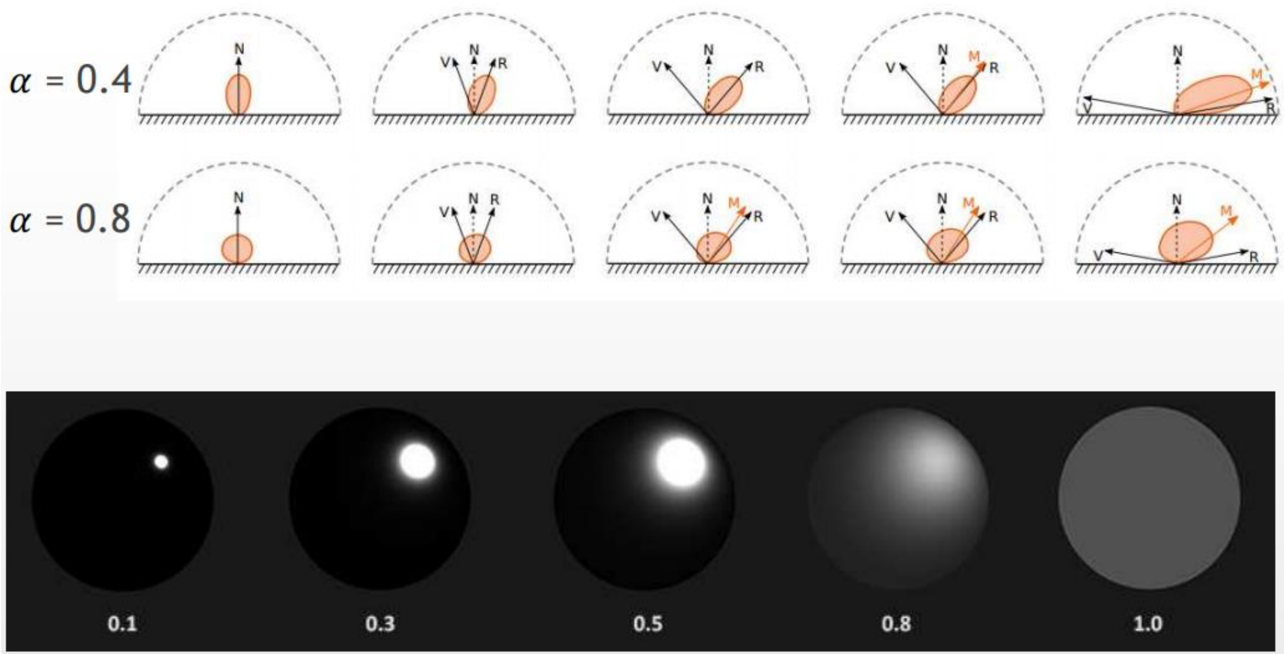
This is the description of how the microfacets are oriented, if they are randomly oriented the material is completely diffuse if they are oriented in same we have a specular reflection.

For the estimate we use a statistical distribution if the material is smooth or rough, the original Cook-Torrance use a different distribution. The **GGX** model uses this distribution which actually is considered the name of the whole illumination model which is the core of the model (from the Trowbridge-Reitz scientists, but GGX is simpler to remember).

$$D(\boldsymbol{h}) = \frac{\alpha^2}{\pi\left((\boldsymbol{n} \cdot \boldsymbol{h})^2(\alpha^2 - 1) + 1\right)^2}$$

It is mainly based on an alpha value, which is a value between 0 and 1 which describes if the surface is **smooth** (when 0) or **rough** (when 1). In the original paper for the GGX they make a mistake instead of the pi there was written 4, then it got corrected, some texts still have this problem.

The only parameter is alpha, the other ones are already present.

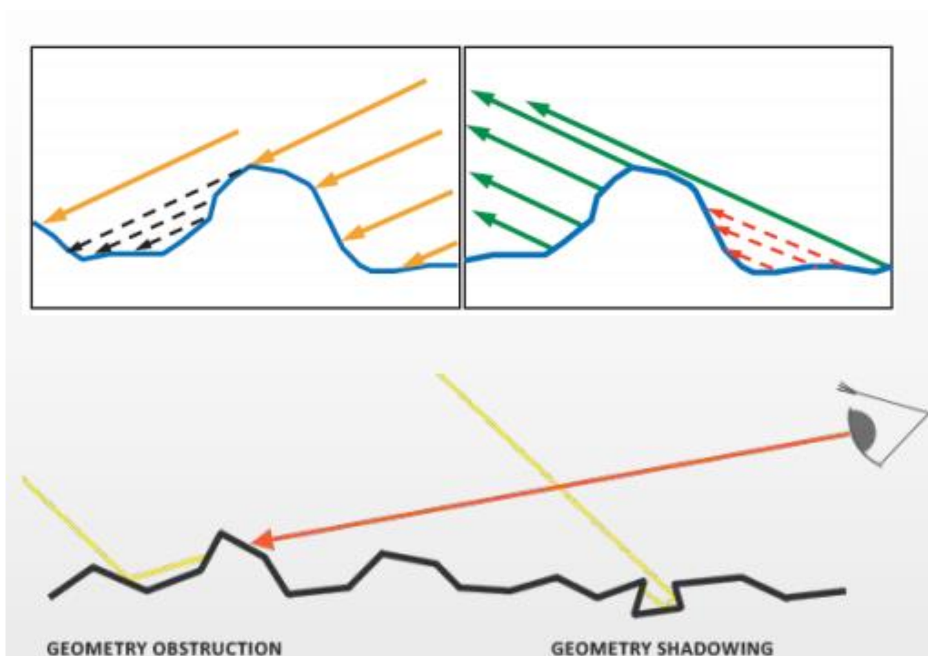


This formula provides different effects regarding the issue with , in the Blinn-Phong you can mix the different components and you have different materials, but the range of different materials is limited. With this GGX approach you can tune small changes in the overall visual effect of the material, it is a really fine and small changing control, the material doesn't change instantaneously.

Geometry attenuation (G)

At microscopic level we have something which is done by small facets that are oriented in different ways. If the light coming on the microfacets, it may happen that the light will get partially occluded by the microfacets itself.

In some cases, this occlusion is called **geometry obstruction** and in other cases is called **geometry shadowing**. The have added in the FDG formula this weight in order to attenuate the reflected light by means of internal self-reflection.



This is the original formula, which is called $\boldsymbol{G_1}$, everything is based on $\boldsymbol{n}$ and $\boldsymbol{v}$. This is the Schlick-GGX method which was used from Pixar.

$$G_1(\boldsymbol{n},\boldsymbol{v}) = \frac{(\boldsymbol{n}\cdot\boldsymbol{v})}{(\boldsymbol{n}\cdot\boldsymbol{v})(1-k)+k} \qquad k = \frac{(\alpha+1)^2}{8}$$

But in the **GGX** version we are using the $\boldsymbol{G_2}$, because it combines two $\boldsymbol{G_1}$ functions one for **geometry obstruction** and one for **geometry shadowing**.

$$G_2(\boldsymbol{n},\boldsymbol{l},\boldsymbol{v}) = G_1(\boldsymbol{n},\boldsymbol{v})\ G_1(\boldsymbol{n},\boldsymbol{l})$$

Is an overall darkening of the overall reflection.

Empiric BRDF - Ward anisotropic model

This is a **different method** (and for anisotropic) which is presented from Ward (worked at Dolby Digital, …) when was working at university he produced and proposed prototype for the gonio reflectometer and measured a very large set of anisotropic materials. Then proposed this empirical model for approximate values for anisotropic materials.

It has 4 parameters :

- Diffusive component p_d
- Specular component p_s
- Standard deviation of orientation of microfacets α, this actually consists of two components:
  - The **roughness index** for **anisotropic materials** consists of **2 indices** ax and ay.

For describe these two indices we introduce the **direction of the tangent vector** for the ax index, and the **bitangent vector** for the ay index (which is given by cross product **normal** and **tangent**).

The formula :

$$\rho_{an}(\theta_i,\varphi_i,\theta_r,\varphi_r) = \frac{\rho_d}{\pi} + \rho_s \frac{1}{\sqrt{cos\theta_i cos\theta_r}} \frac{1}{4\pi\alpha_x\alpha_y} e^{\left[-2\frac{\left(\frac{\boldsymbol{h}\cdot\boldsymbol{t}}{\alpha_x}\right)^2+\left(\frac{\boldsymbol{h}\cdot\boldsymbol{b}}{\alpha_y}\right)^2}{1+\boldsymbol{h}\cdot\boldsymbol{n}}\right]}$$

There is **no link with physical models**, is just a mathematical elaboration.

Other models

We saw the most common, there are others for diffuse reflection :

- Minnaert model

- Oren-Nayar model

And others for anisotropic reflection :

- Ashikhmin-Shirley model
- Heidrich-Seidel model

Texture mapping & Procedural Textures

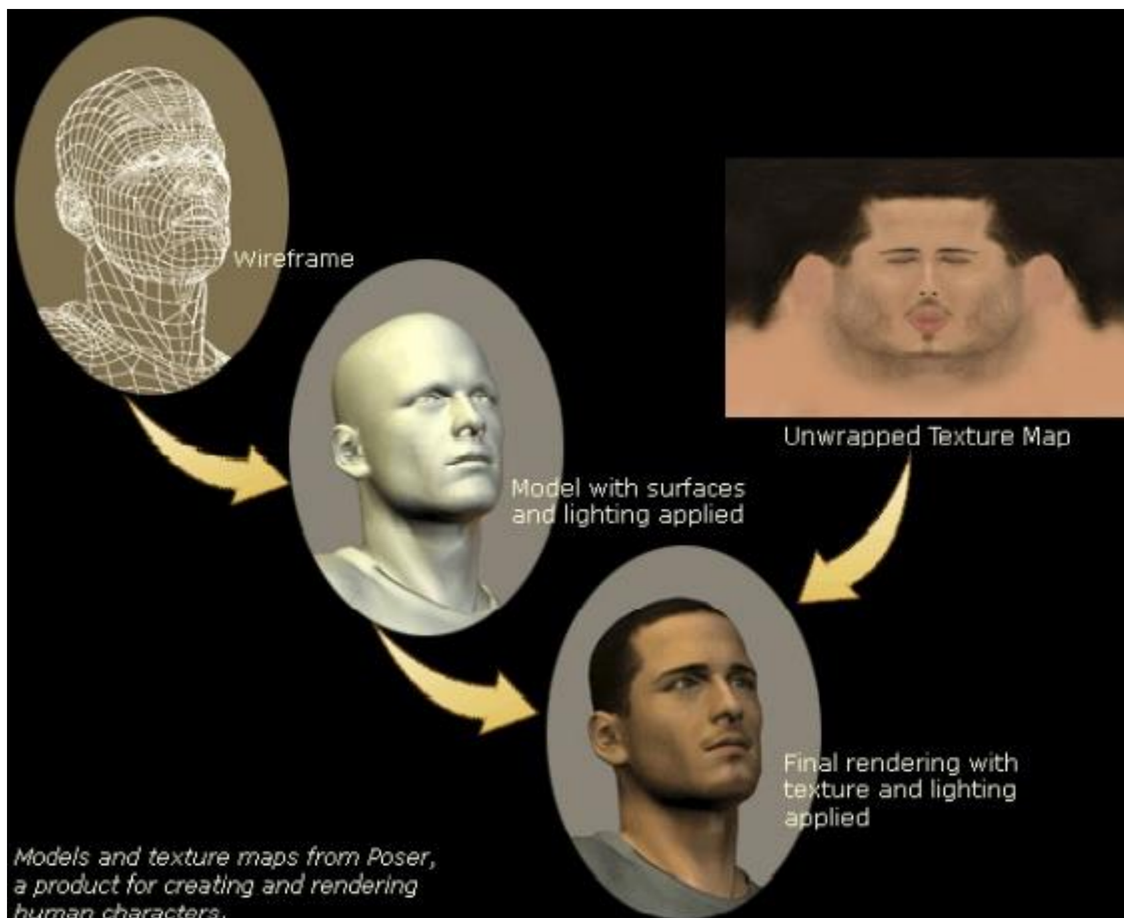
Texturing is the use of images of different nature to enhance the appearance of objects, regarding color, details but also using the information in the image to guide the effect of the illumination model.

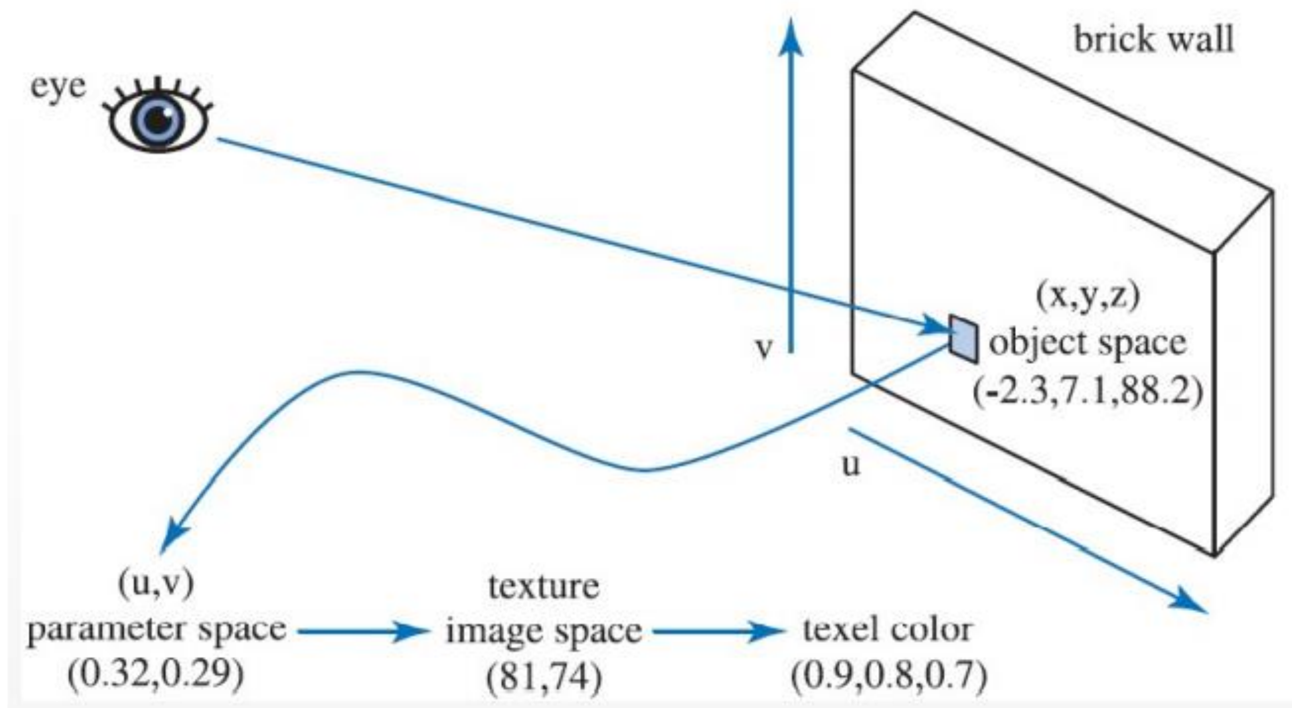
Image's assets can save in external files on disk and in a similar way to *mesh objects* be loaded during the **Application Stage**, where the images will be loaded in memory and then will transferred on the **GPU memory**. But also, images **can be generated procedurally on the fly**.

Texturing approach

In my mesh a correspondence **between every vertex in 3D with a 2D pixel of the texture**, this pixel is called ***texel***. When I need to use the textures, I use this mapping (or correspondence), which is a set of two coordinates (u, v) for read the values of the ***texels*** and in someway applying that on the mesh.

The *texel* value is used in the computation of the final color of fragments, this *texel* can be also combined with other colors.

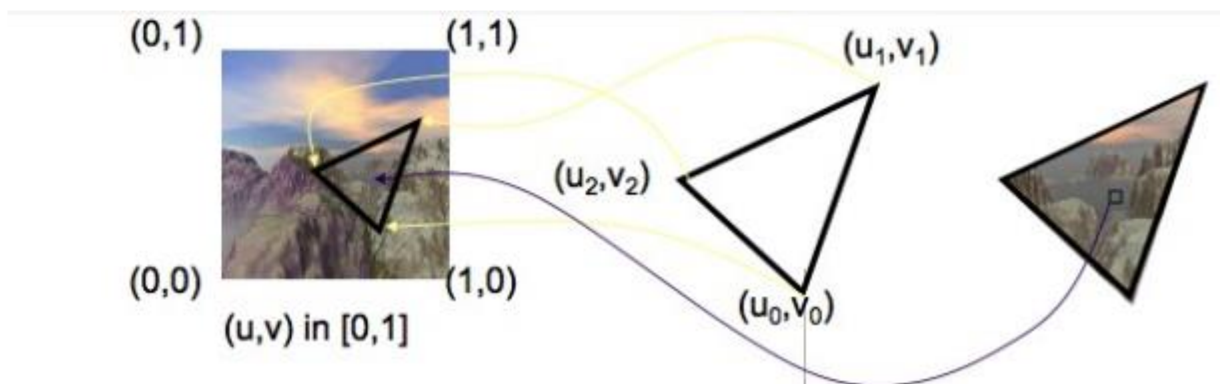




This means that for each vertex I have the three coordinates (x,y,z) in 3D space but also the (u,v) texels which are two coordinates. **The (u, v) coordinates are inside a range between [0, 1]**. When I need to do texturing I take the value in this range and I convert them in integer values that are inside the range **[0, rows]** and **[0, cols]** for scaling the coordinates on the actual dimension of the image then I read the color (and in the end some computation will be applied on the mesh).

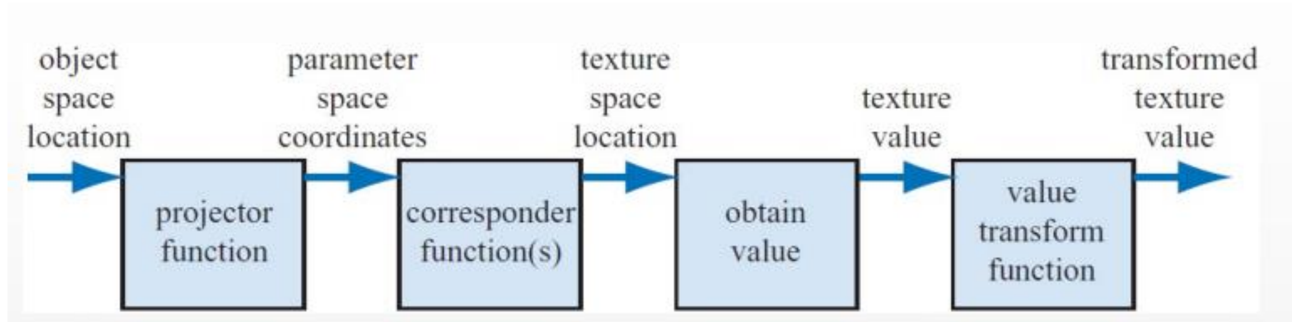
It is possible to have different set of (u, v) coordinates this because is possible to have different textures.

The idea is when I have to process one triangle, I need to know triangle which part of the image is assigned to the triangle.



We can express the process and the different option using a **texturing pipeline**, which is a **sub-pipeline** just for texturing.

Simplified texturing pipeline



Projector function

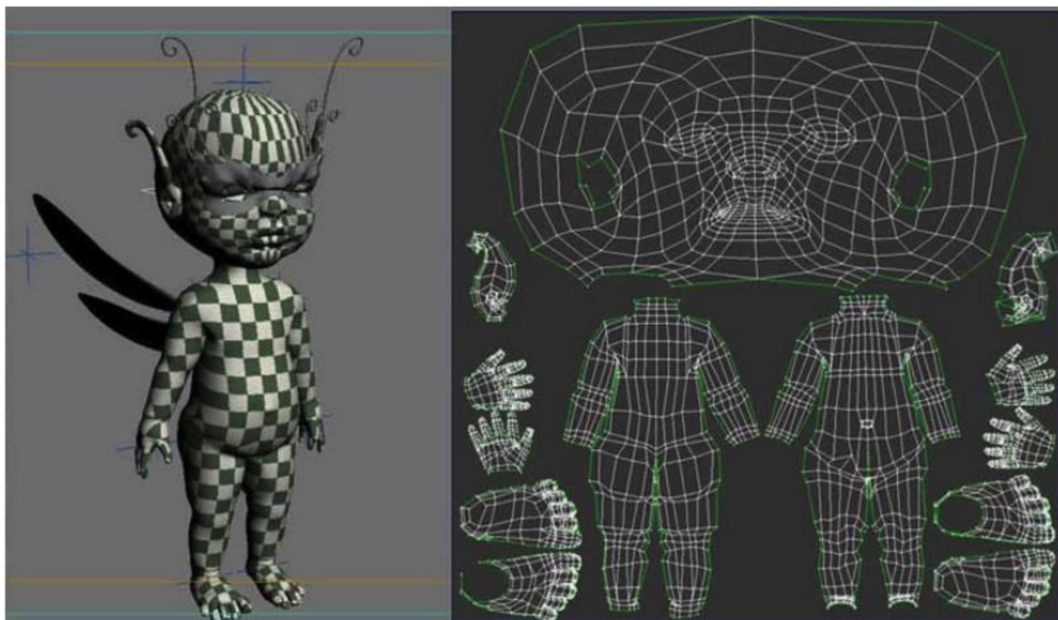
The first stage is the **projector function**, in this stage it is created a relationship between the **3D object reference system** and the **2D texture reference system** (this is a mapping between 3D vertices and 2D texels)**.** The letters *u* and *v* are the most commonly used for expressing texture coordinates (called *UV coordinates*), but there is also (s, t).

These **UV coordinates** can be expressed manually by the developer, but in case of more complex models they are already stored in the **3D Mesh file**, so they are usually part of the model object (COLLADA, .obj, …), in case of manual assignment through modeling software is called **mesh unwrapping**.

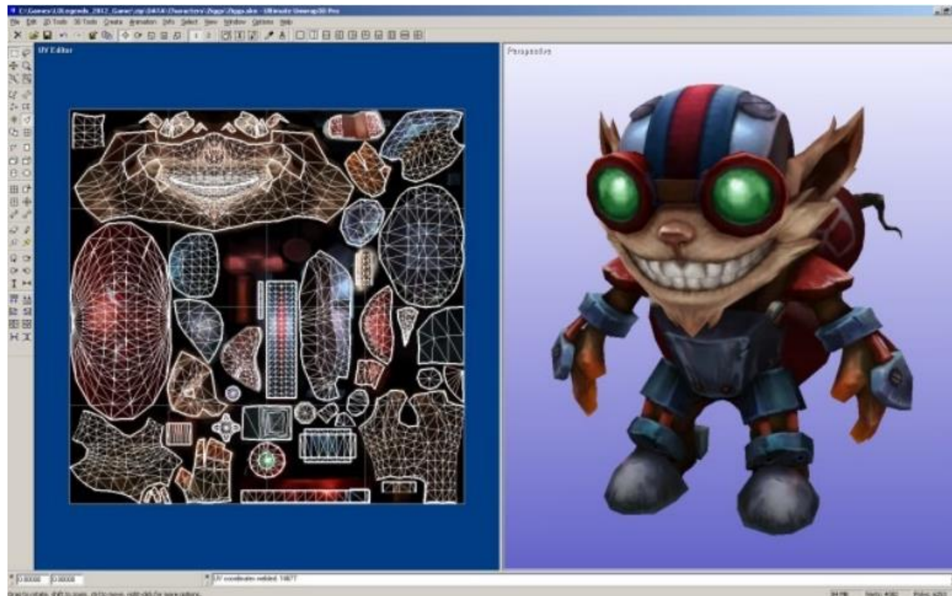
From the developer side there is the need to make a correspondence for each **triangle** with a portion of the **image described by the texels.** These texels values are not applied until the **fragment shader**, during the **rasterization** the **UV-coordinates** are **interpolated** on the **mesh**, then during the fragment shader the textures are read and applied.

Projector function : mesh unwraps

This is a process which is not usually done by the developer, it occurs at the end of the **modeling** where you model is defined. After that, the modeler will create the **UV-coordinates** using **visual tools** which gives you possibility to select line of edges and using this set of vertices to unwrap the mesh on a plane. The **plane** is the texture, usually this kind of tools will give you a **visual pov** to see where the unwrapped mesh on the plane is distorted or not.



Usually, the model will be broken in different parts, so that you can apply different colors. Actually, this process is used when you have to create the texture by **digital painting**.

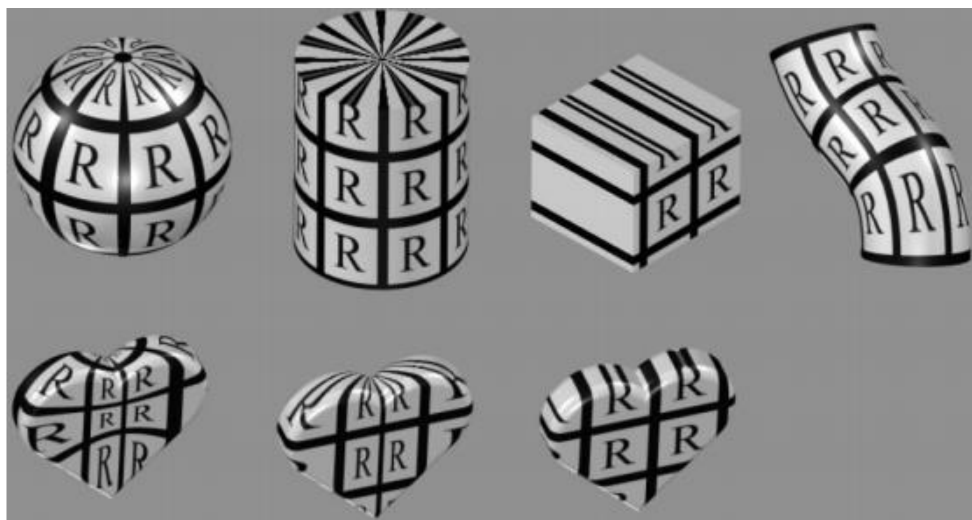


This is why you see these textures as packed in a small square, where **digital painters color the textures directly in the modeling software**. In this case the texture is created during the creation of the asset, so is not a real image until then, then will be saved near the mesh object file.

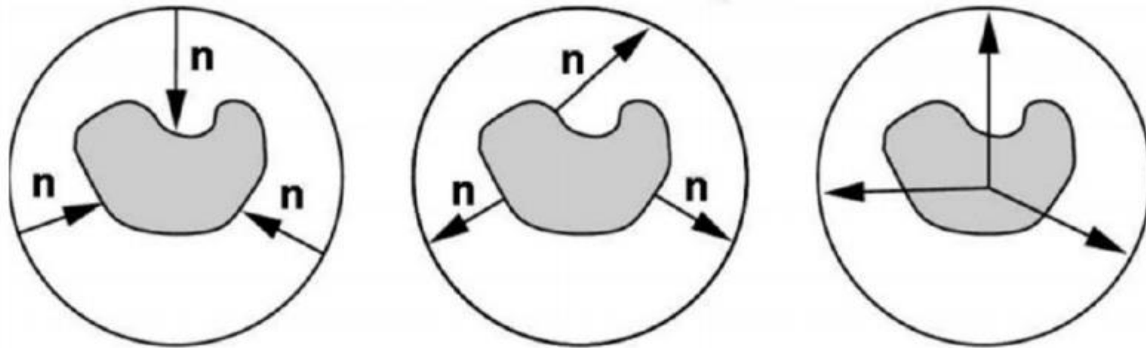
Then the *Vertex-Texels* correspondence will be made automatically by the **mesh unwrap technique**. For objects created in this way as developer we have to do nothing for the projector function, because it is already done for us.

Projector function : intermediate surface

In case of simpler models, like *spheres, cubes, cones*, … the **UV-coordinates** can be generated procedurally. The **2D texture image** is mapped on an **intermediate surface** (like a sphere), there are mathematical function in projective geometry that tells how to map a 2D plane (the image) on a 3D object, then the **UVs** are mapped from the intermediate to the final surface.



After that the mapping to the final object happen in different ways using the intermediate surface.



Considering a cylinder for an intermediate surface, and that there is the final object to be mapped inside, I can have three different kinds of mappings :

1. I can go from the cylinder to the center of the object, where the color of the object be mapped on the internal normal of the cylinder.

2. I can consider the **normal of the object vertices** and from there i externally read the color of the cylinder intermediate surface.

3. I can start from the center of the object and I go in a **radial way outside** and again when i intersect the cylinder I read the color on the cylinder.

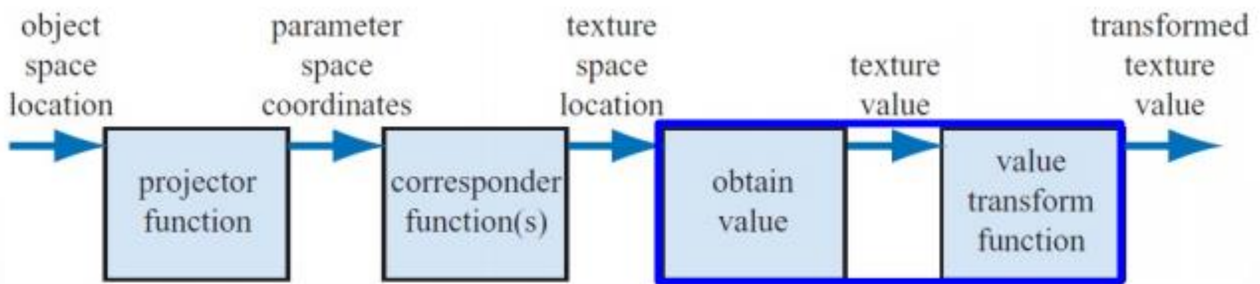
Corresponder function

Once we have the UV-coordinates, there is the second stage of the texture sub-pipeline, the **corresponder function.**

We now have to read from the texture, we can add some **flexibility** during the mapping of the texture of the object. The texture is expressed in $[0,1]$ range. In the texture space we may "think" to have a space with two axis and the texture between 0,1. But, if I try to express the UV-coordinates **out of this range** I have to apply some kind of **mapping policy** the provides me different kind of effects when the range is exceeded .

- **Repeat** : outside the range I just repeat the texture.
- **Mirror** : outside the range I apply the rotated and mirrored texture.
- **Clamp** : outside the range I assign the value of the last pixel on the border.
- **Border** : outside the range i use a specific color.

Texture values

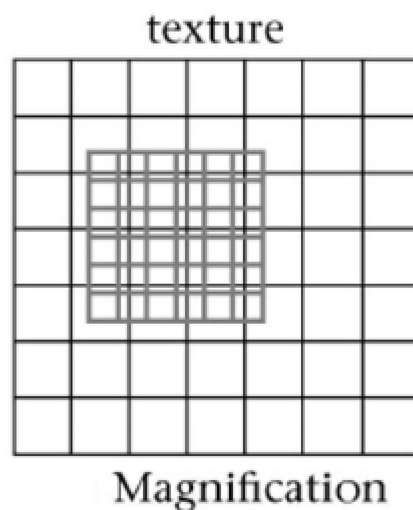


Before reading the final color from the texture I read the value from the image, and then i can also apply other processing before actually using the value. This kind of transformations are linked directly to the **Image Processing** field.

In the most common case, a 2D image is "***attached***" to a surface, after it is being loaded from disk. But in the case where the surface covers an area that is larger or smaller than the texture dimension, there is no more a direct correlation between **Texel-Fragment**.

Magnification

In this case the texture image is smaller **must be enlarged** to reach the match between texels and fragments, the smaller grid in light grey is the texture.



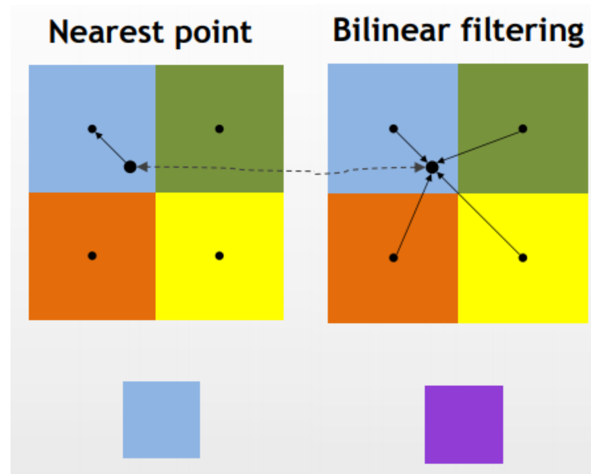
We solve it using filtering, we get this technique from image processing field.

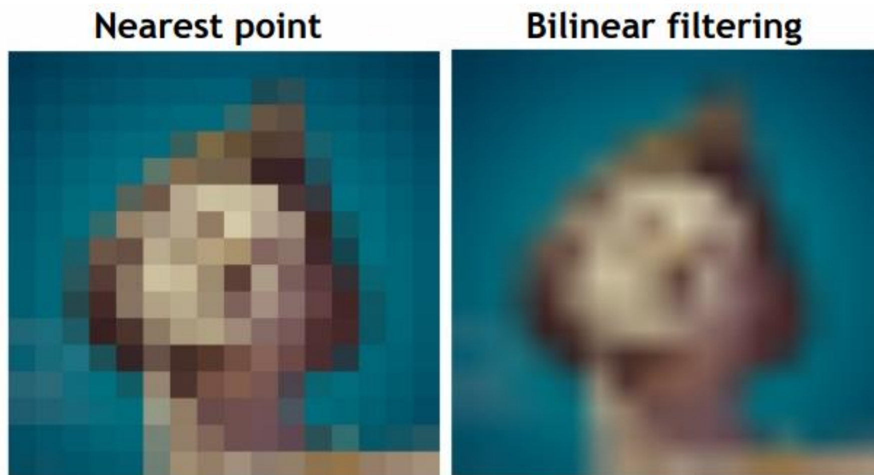
During the mapping from UVs $[0,1]$ range to the discrete $[w, h]$, there is a **really low probability to match a texel center**.

There are two commonly involved techniques to fix this problem, the **big black dot** represents the **UV-coordinates in the texture image space**, and the **four colored squares** represents the **texels** :

- **Nearest point** (or **nearest neighbor**) : where I choose the closes texel to the sampled one, the problem that in a rasterized image (so a true color image) this led to a very strong aliasing. Since that there will be a strong difference of colors near fragments, in the color will be much more shaded.

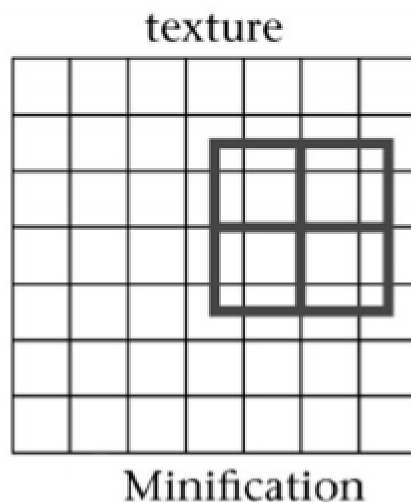
- **Bilinear filtering** (or more, but you will pay it), this is a more realistic approach, where a **weighted average between the 4 nearest texels** occurs and lead to a new pixel.



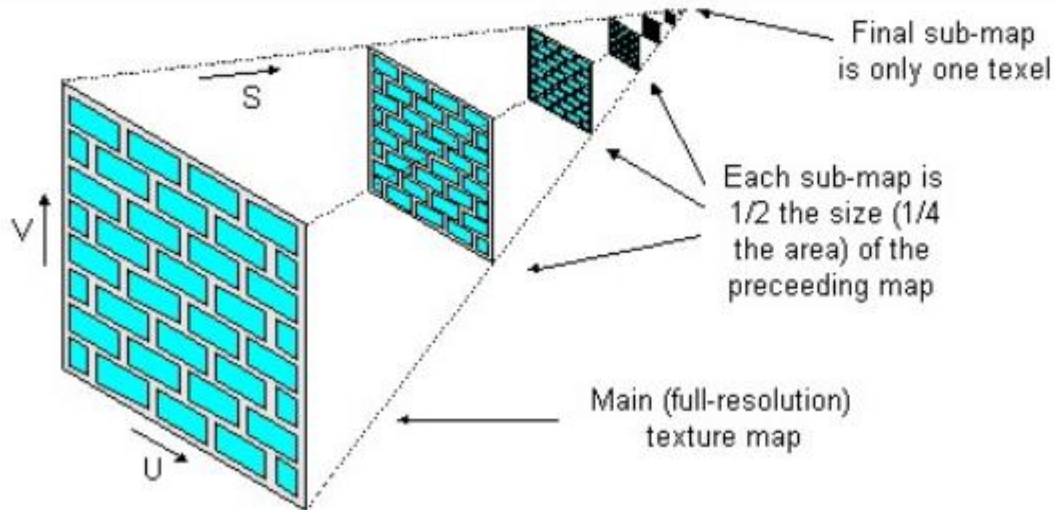


Minification

In this case one texels corresponds to many fragments, for example the object is far from the camera and the dimension of the texel is bigger than dimension of the object. One texels cover more than one fragment.

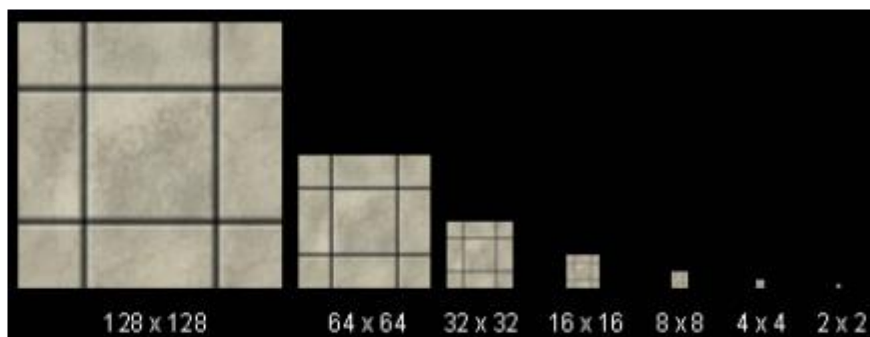


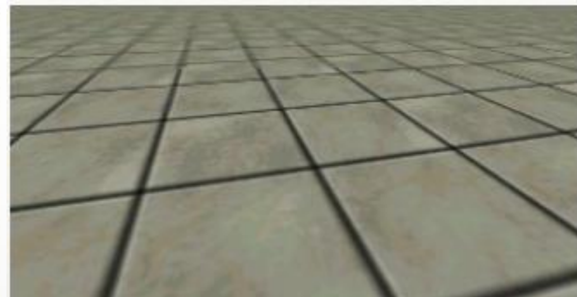
In this case we solve this issue by using a more sophisticated approach, **the mipmaps**. I have my textures and instead of subsample and minify at runtime, i precompute different version of the image at reduced dimension. This operation will obviously occur offline, during the asset preparation.



In this case i'm using more data, but they can be organized in memory in a particular way which lead to store **only 33% more memory** of the original buffer.

During **runtime**, in case of a basic approach without mipmapping it will result in lot of gaps and jagged lines effects**, with mipmapping there is a better effect which is not perfect but lead to much less gaps and overall jaggies from resolution reduction.**





without mipmapping with mipmapping

Even in the case of mipmaps is possible to apply filtering transformations, we are talking about **mipmapping filtering**. These filters are applied to **smooth the transition between different mipmaps**.

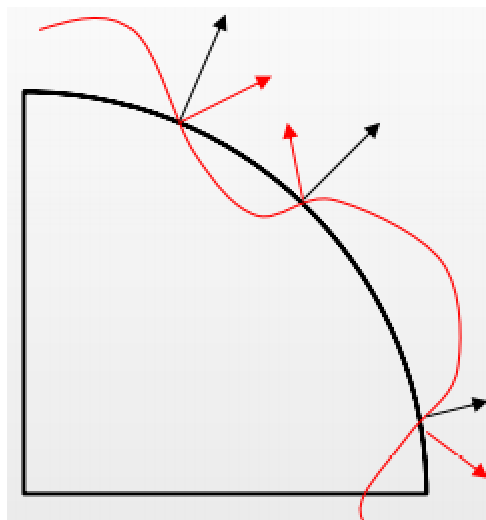
This means that i take two mipmaps and i apply a filtering technique in order to select a color value which is a blend of the mipmaps.

- The **Trilinear filtering**, this means applying the bilinear filter to the current mipmap and *next one*, after that i interpolate between these values.
- The **Anisotropic filtering** enhances and changes the filtering considering the orientation of the camera to the surface.

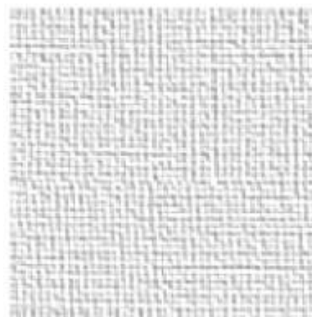
Regarding the techniques that changes the color effects of the image itself we finished, now let space to the techniques that are able to change the rules of the illumination model

Bump mapping

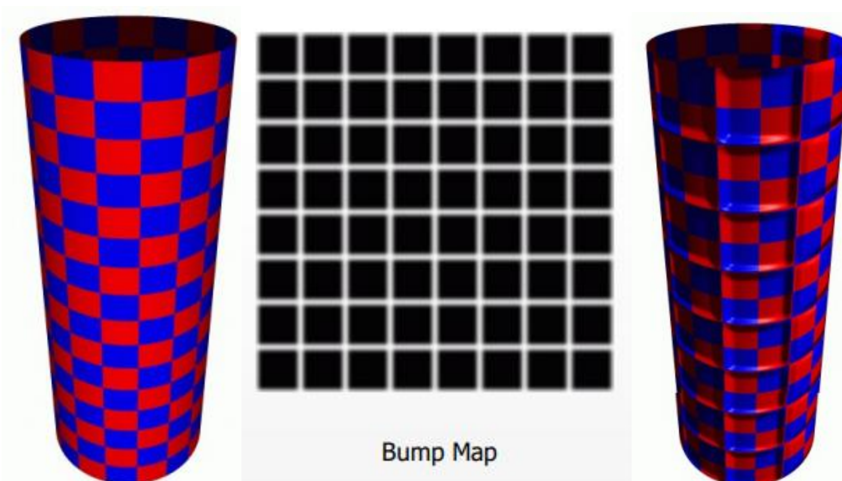
This is the simplest technique; It is used to **simulate wrinkles and perturbations without doing modifications to the actual mesh surface**. What actually happens is that the surface normals are perturbated before the application of the illumination models, for changing the normals values it will be used an heightmap (a grayscale image that contains the normals changes).



I can attach a texture which has some information of the normals of my mesh and the final effect will be a perfectly smooth mesh which appears after the application of this texture and the illumination models as full of wrinkles and bumps.



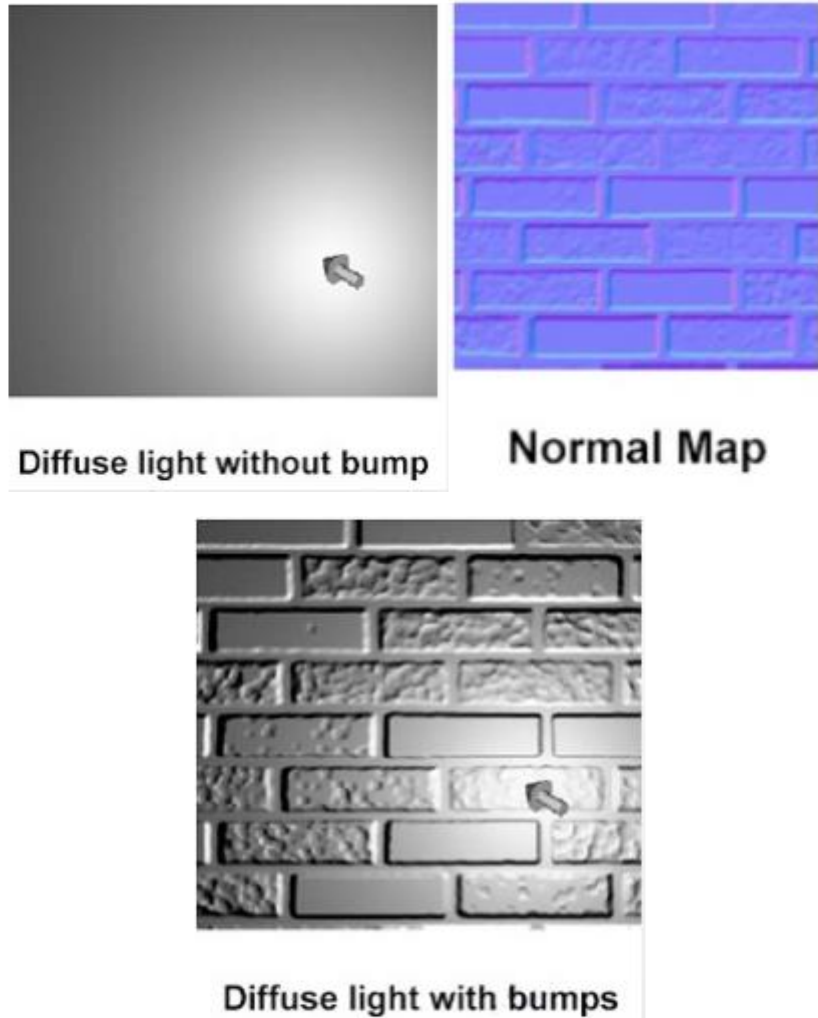
A **bump map** is an image which is composed of only gray values between $[0,1]$, and these values are **weights** that are going to be applied to the original normals of the mesh, in order to change its direction. Then during the illumination model, I will read these new values, making it believe that the normal are going in particular direction described from the texture.



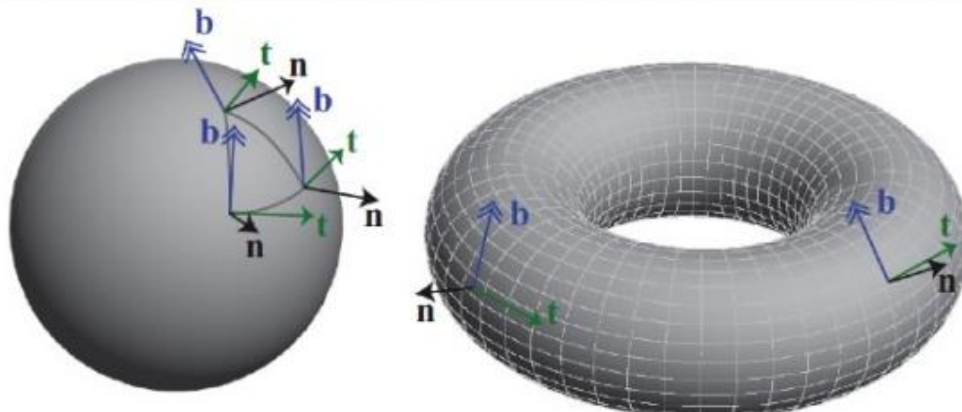
But in reality i'm not moving any vertex I'm **just changing the direction of the different normals**, but the surface is the same as described in the mesh model.

Normal mapping

This technique is an advanced version of the bump mapping, in this case the normals are not perturbed but **substituted** with a new vector. The XYZ components of the new normals are stored as **RGB colors** in a new texture which is called **normal map.** The result is really effective.



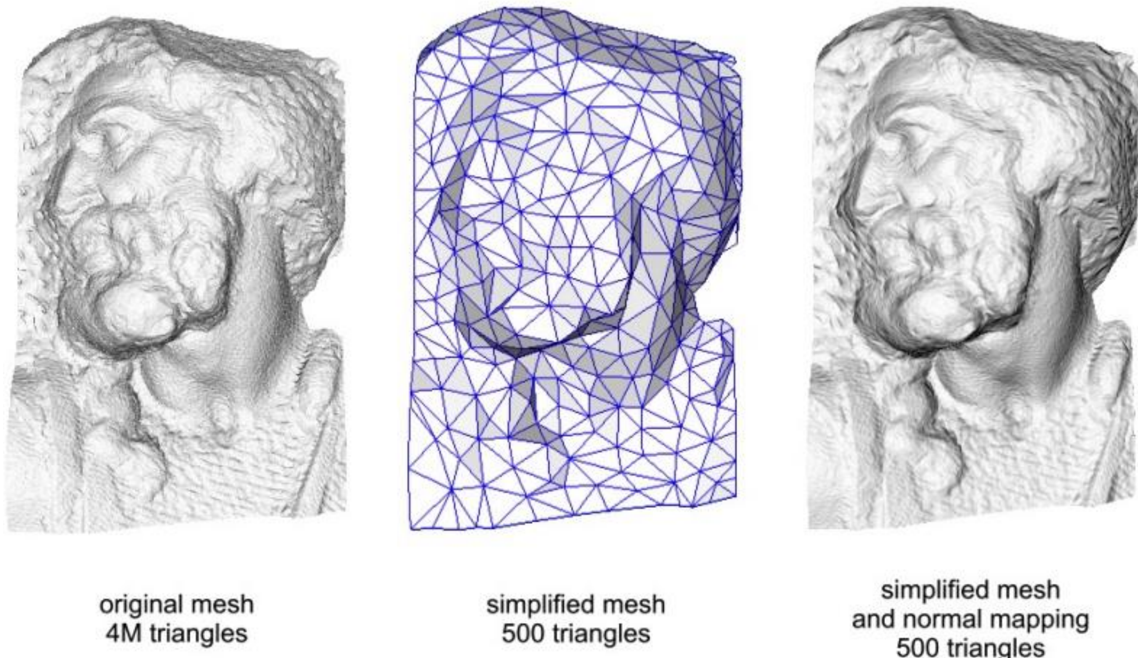
The normal maps have usually this kind of colors, this because we want to pass a **vector** which is a **substitute of the normal** and we need to express this vector in a reference system. The normal maps tend always to have this purplish color, because the larger values are usually stored in the blue channel of the RGB.



The new normal XYZ components are expressed in *tangent space*, this is a local reference system composed by the **normal**, **tangent** and **binormal/cotangent/bitangent** (which is the cross product between the other two).

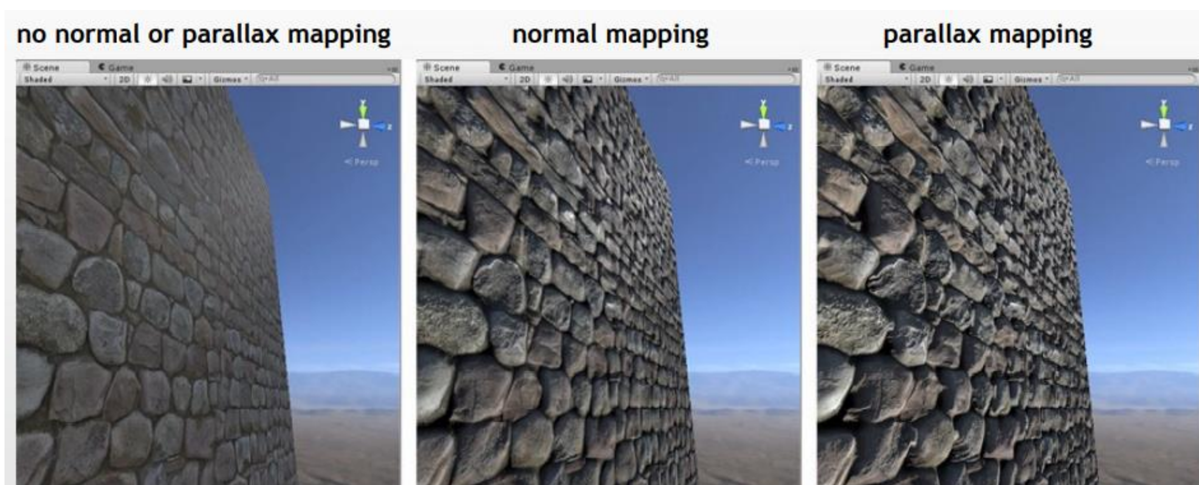
After that, the new normal must be transformed to **world space** before being used from the illumination model, so the same transformations that are going to be applied to the model are going to be applied on the **normal map**.

The normal map is **pre-calculated using a high-poly** version of the **model**. In this way we can reuse the high-poly normal map on a simplified mesh gaining an extremely good approximation of the high-poly model.



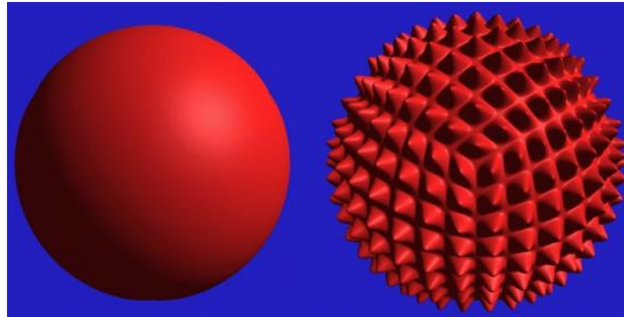
Parallax mapping

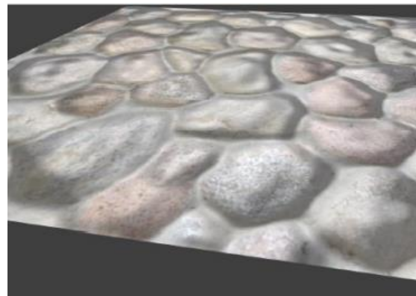
This technique is used to enhance apparent depth in textures as *stones*, *walls*, *etc* … The texture coordinates in a point are displaced by a function of the **view angle** and the value of the **height map/normal map** in that point.



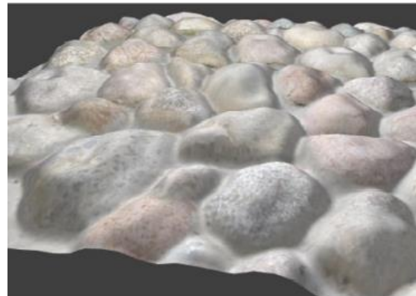
Displacement mapping

This technique uses the texture to actually **modify the surface vertices**, the RGB values are used to displace the XYZ local coordinates of vertices (this is really different from **parallax mapping**).





Bump mapping



Displacement mapping

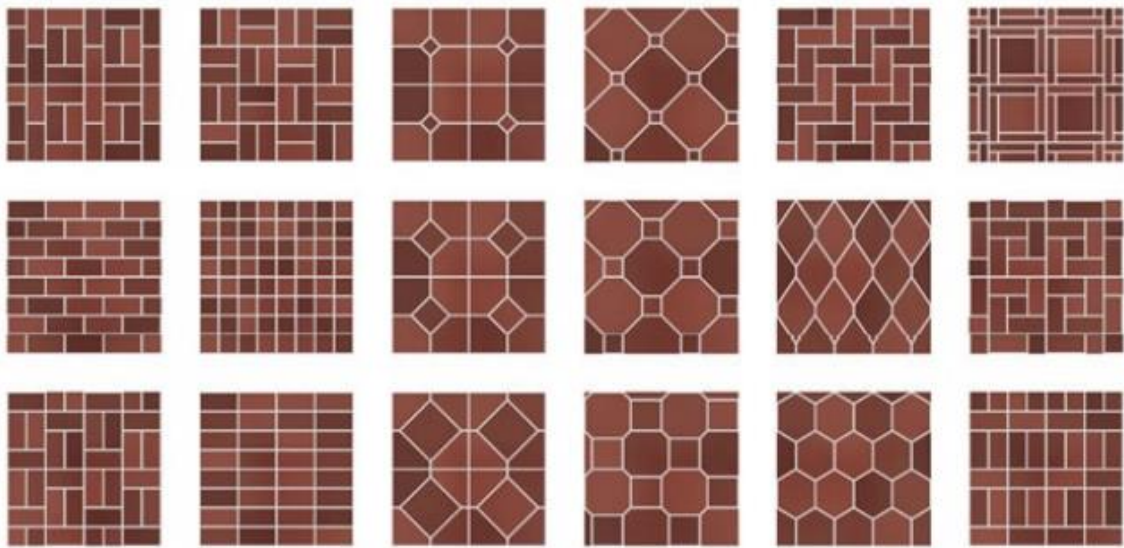
Procedural textures

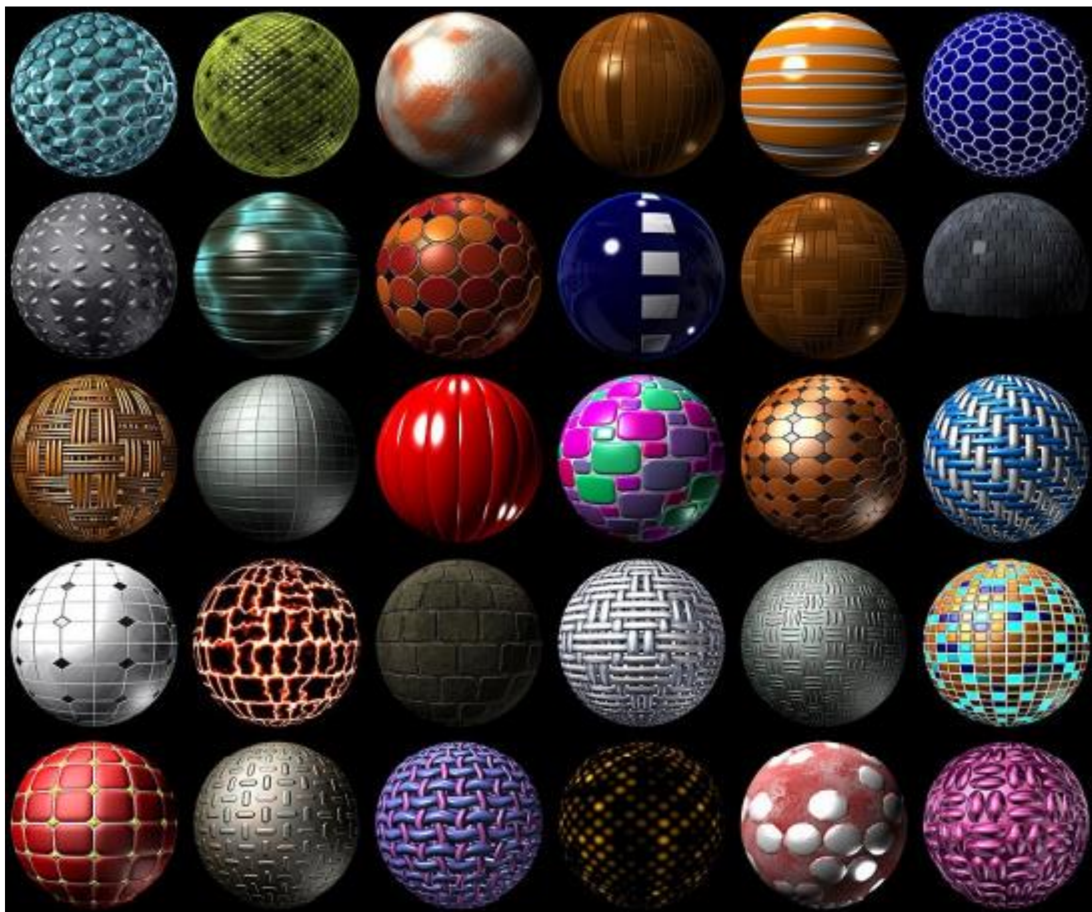
There are algorithms that can generate a texture *"on-the-fly"*, usually we create **regular patterns** (*walls, …*) or **random patterns** (marble, woods, …).

The computational resource is required to have different versions of the same kind of pattern in order to avoid repetition and to enhance variability.

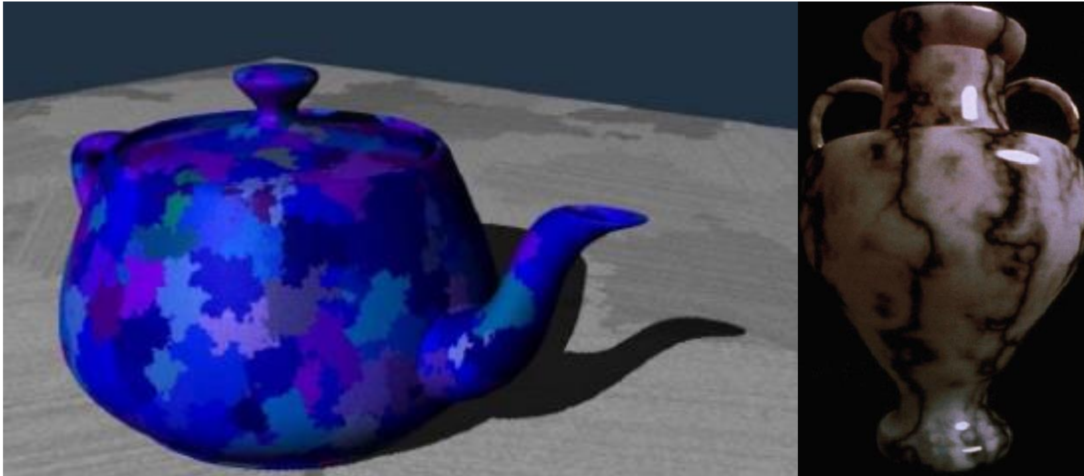
These patterns are generated using the UV-coordinates, on the basis of that we use thjis values in a function that decides which part of the pattern you are on the subject and other parameters like the vertex coordinate together with other specific textures.

Regular patterns (*stripes, squares, regular polygon repeated, …*)





Irregular patterns (*veins in marbles, wood, and other visual randomic effects, …*)



Procedural texturing was largely involved in the first years of computer graphics, this because of the limited use of the external images for textures (the other alternative was solid colors).

It is still useful today, this because brings no issues related to resolution (it is created on measure of the image), no issue related to magnification, minification of the texture image.

I avoid issue of repetitions of textures and border of textures, and with a good procedural technique due to parametrization you can reuse the same approach several times adapting to different situation.

With **embedded systems** and **WebGL**, where the environment has **limitation** on **memory** and **data transmission** (not really a problem nowadays) the use of a procedural approach brings the possibility to determine the materials from the client-side using some lines of code instead of a full **RGBA** image.

Issues of procedural texturing

It's not like painting, a mathematical check will occur, this will bring an **unpredictability** of the result that could bring some serendipitous discover.

Reading from texture memory have his issues because it depends on the **bandwidth** of the **GPU** and from the BUS, but with the current time this is less an issue then in the past. **Anyway, the procedural texture creation involves algorithms, so by nature they are more expensive than reading from texture memory.**

There is also a problem with aliasing, which is not trivial in some cases.

Procedural patterns GLSL

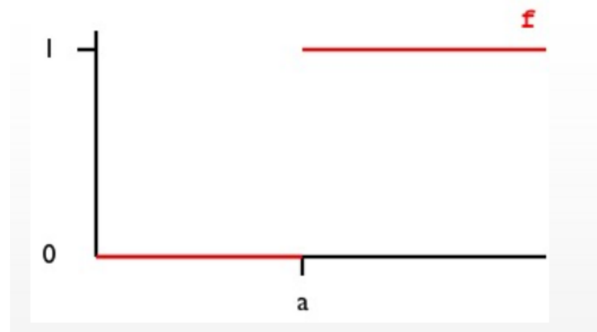
The idea is to have a whiteboard with the domain $[0,1]$ on the u axis and $[0,1]$ on the v axis. You apply functions using the (u, v) values as parameters and these functions assign as a result a grey level between $[0,1]$. Actually, you are **creating a map**, this is kind of a map which may be black, white or gradients values, and you can use this value between $[0,1]$ to interpolate among two colors.

In case of **regular patterns** are used **functions like *step, sine* and *distances***. In case of **random patterns** are used ***noise functions* and its variants**. The **UVs** of each vertex are used as **parameters** inside this **function.**

One function provided by the shading language is the $step(a, x)$ function, which it's kind like an if-else statement.

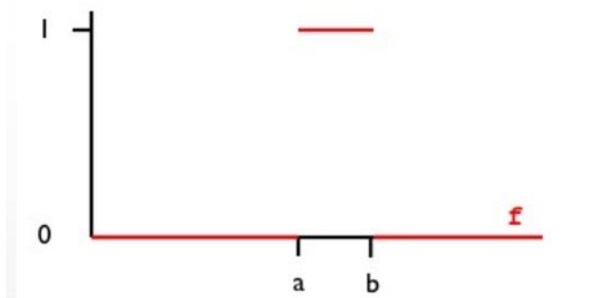
$$\text{If } x < a \;\; 0.0 \; else \; 1.0$$

```
float f = step(a, x);
```



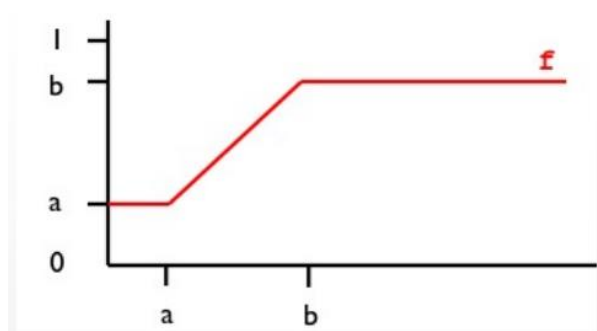
A combination of steps is called pulse, but this isn't predefined in the shading language. You can create a pulse by subtracting two step functions (maybe using the *preprocessor define*). A possible resulting **map** would give someone with the repetition *black, white, black, white, …*

```
float f = step(a, x) - step(b, x);
```

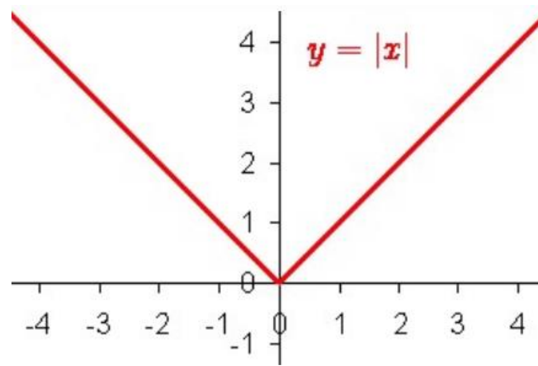


Another function which is used in **GLSL** is $clamp(x, a, b)$, it performs a **linear interpolation** on a specific value x between two further intervals a and b.

```
float f = clamp(x, a, b);
```



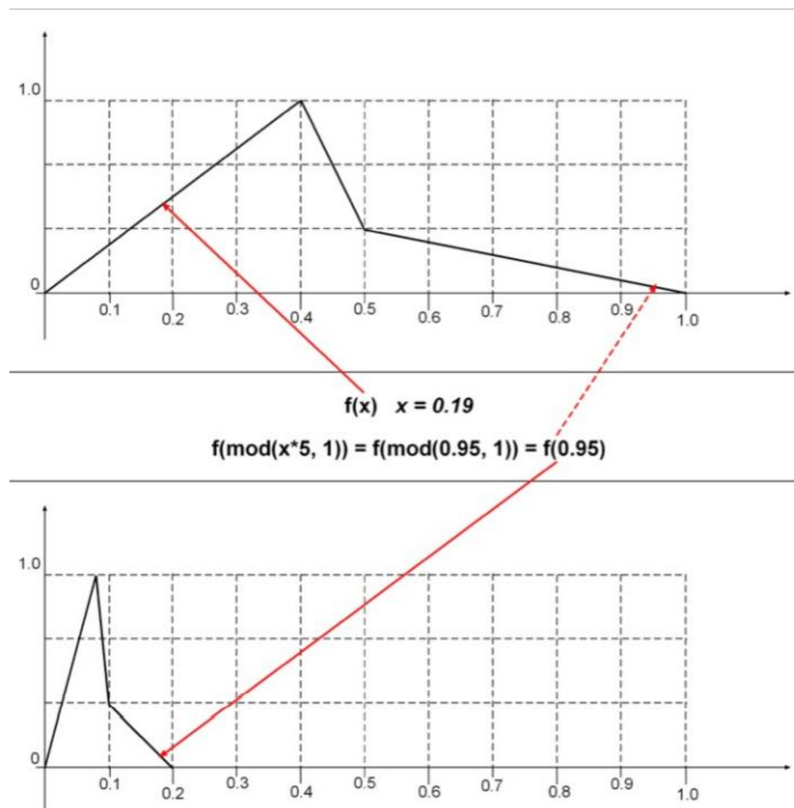
Another is the absolute function $abs(x)$ which gives you a symmetric function.



On the basis of this functions, you can apply repetition of the pattern as many times you want, this by using the $mod(a, b)$ function. The **mod** function takes two parameters and returns the ***remainder*** of the Euclidean division between a and b.

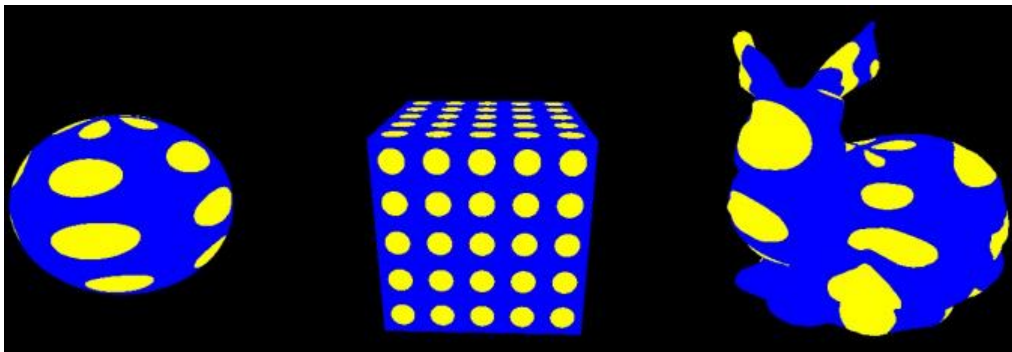
In the case where the second parameter b is 1.0, the function will return the **fractional part** (*mantissa*) of the first parameter a.

We can perform a trick and multiply the first parameter by a multiplier, and then use the result as the first parameter of the mod function, with the secondo equal to 1.0. The result will be the value of the future steps on the domain respect from the actual x, this means that will be performed a compression along the domain in the range $[0,1]$.



With regular patterns we have some issue with **aliasing**, because when we have **straight lines** or **curved lines** we fall in the jaggies issue during the rasterization stage. We have to assign a grid of colors for curved or diagonal lines.

In particular in the $step(a, b)$ function, when I suddenly change the frequencies of the information, this problem occurs also in the *mod* function at the ends of the first interval.

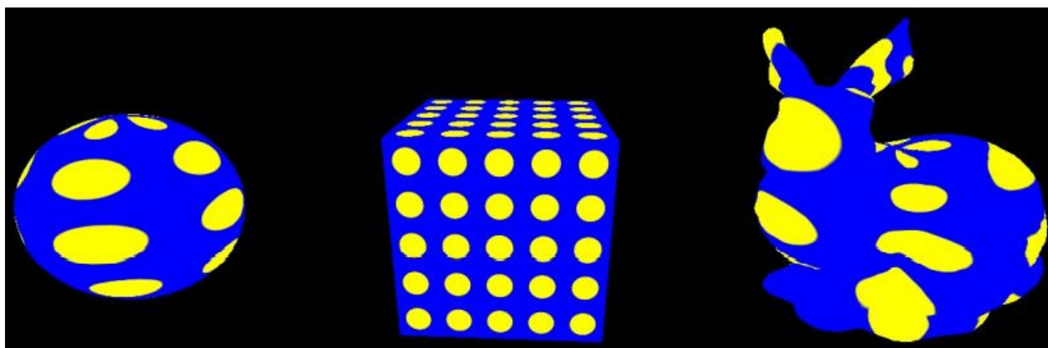


Approaches, to avoid *dangerous* techniques listed in previous slides u

The first approach to avoid "**dangerous**" techniques listed before, I can use the more sophisticated blending functions like $smoothstep$ or mix or we use a specific feature of **GLSL**.

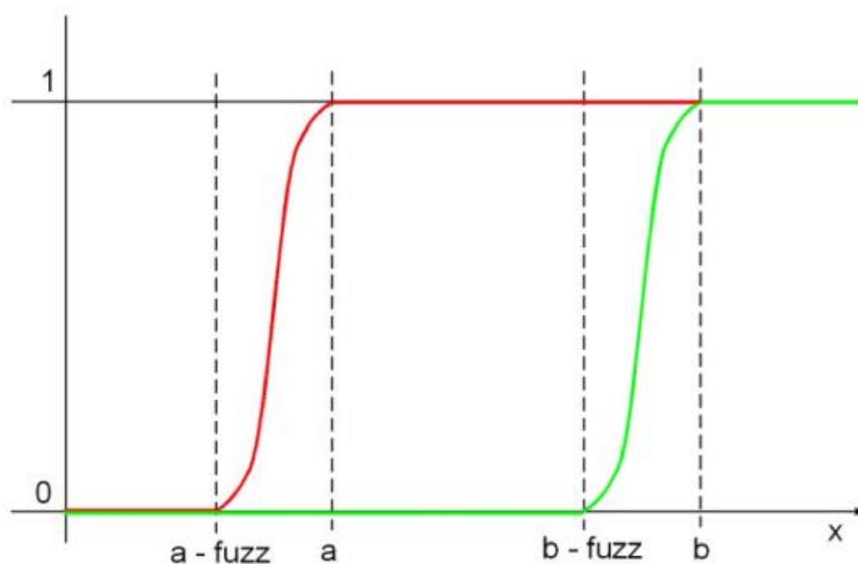
The $smoothstep(a, b, x)$ function guarantees a softer approach in respect of the brutal $step(a, b)$ function, the first two parameters are always the two edges of the interval, the x is the value where the **linear interpolation** of the value will occur in the middle.

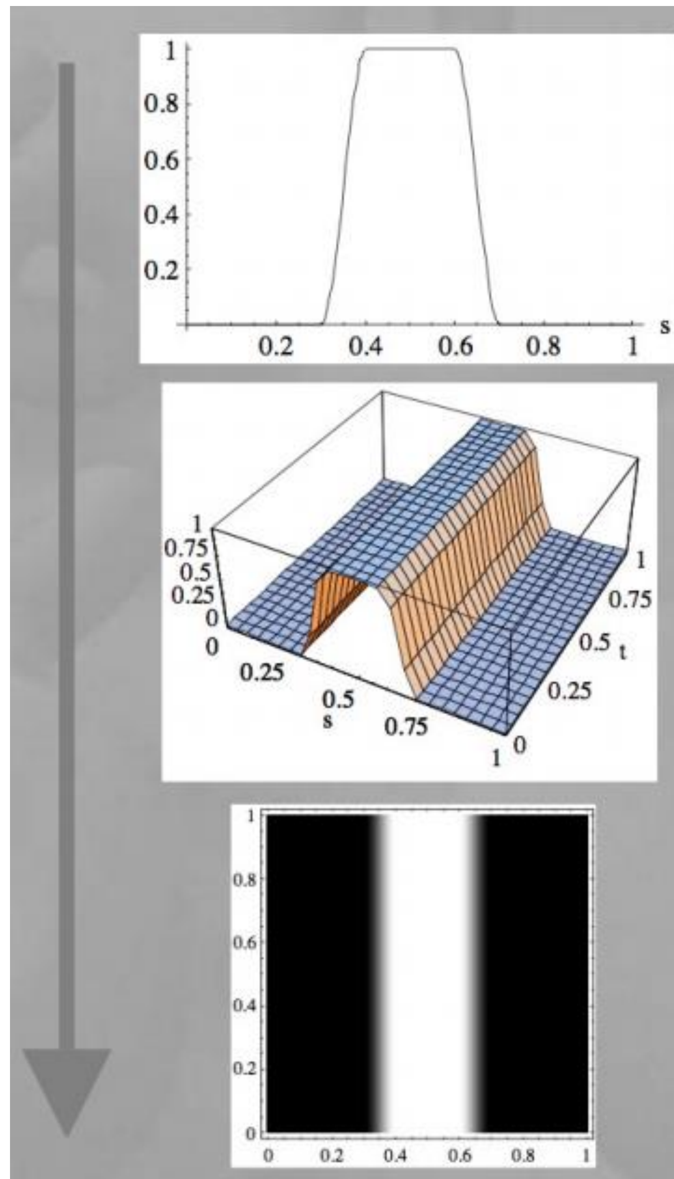
In reality the function is used as $smoothstep(a - fuzz, a, x)$, where the first value is decremented by a **small transition** interval called $fuzz$. This function will give a much better result.



By saying this we can redefine the $pulse$ function as a subctraction of two $smoothstep$ functions.

$$pulse = smoothstep(a - fuzz, a, x) - smoothstep(b - fuzz, b, x)$$

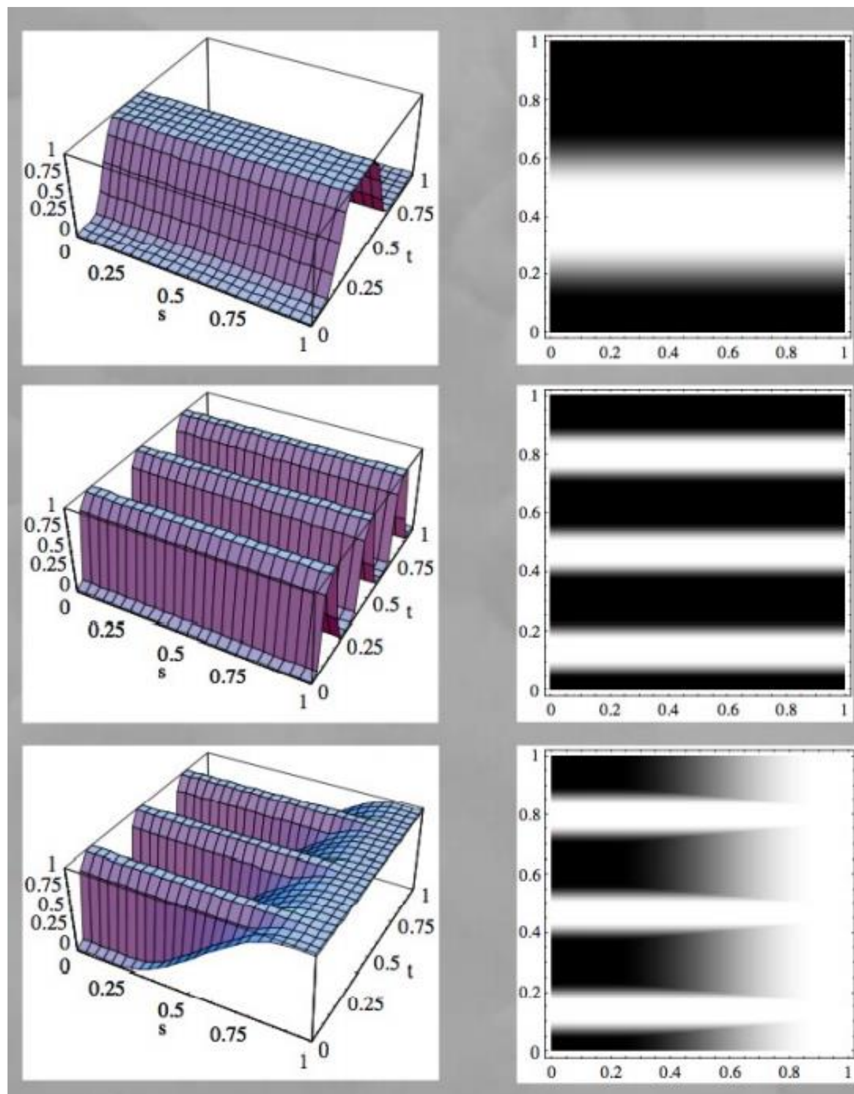




The last image is the map that will be used to assign the colors, that will be used in the shader that will read from the texture and then applied it.

Combination of functions

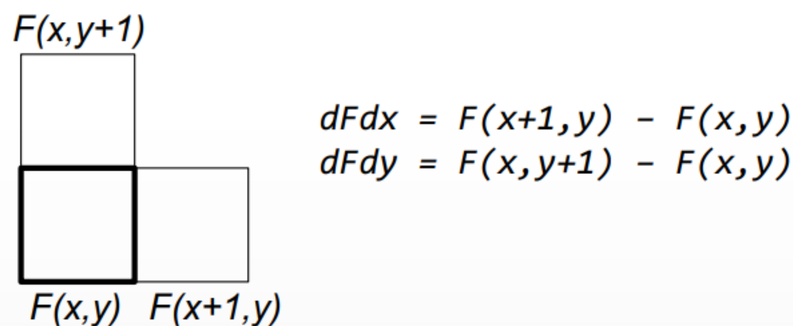
It is possible to combine different functions, more you combine hardly will be to predict the behaviour of the result. For example, is possible to apply a mod function to a pulse function and obtain a repetition on the t axis.



It is possible also to perform more complicated things like using that function and apply to that another smooth step but this time on the horizontal axis s.

Antialiasing: analysis of fragment neighborhood

In **GLSL** is possible to use two functions $dFdx$ and $dFdy$, they both take the value of the fragment, and they check the difference between the fragment on the right and the one on the top.



This is the only exception in the pipeline where the fragments are not elaborated completely independently.

In this way we can adapt **dynamically** the fuzz transition of the smooth step, because I can see the difference between the fragments and i can apply a different fuzz which performs the best in that situation.

```
// threshold is constant, value is smoothly varying
float aastep(float threshold, float value)
{
    float afwidth = 0.7 * length(vec2(dFdx(value), dFdy(value)));

    // GLSL  fwidth(value) is abs(dFdx(value)) + abs(dFdy(value))
    return smoothstep(threshold - afwidth, threshold + afwidth, value);
}
```

As it is possible to see in the code it calculates the $fuzz$ offset dynamically in base of each fragment distance, and then use this value for the smoothstep function.

This function is applied for every fragment, and it checks the difference between the two near functions, and it adapts **dynamically** the antialiasing function. This is a very effective method, and $dFdx$ and $dFdy$ are computationally more expensive of other **GLSL** functions, because they access to the information of the near fragments.

Now let's move on random patterns.

Noise

In case of natural surfaces, they present irregularities like *veins, stains, stripes, …* this kind of random patterns are commonly based on **noises**. With "**noise**" we refer to a set of mathematical functions used to **perturb in a pseudo-random manner a procedurally generated pattern**.

There are different noises, but they usually share the same properties :

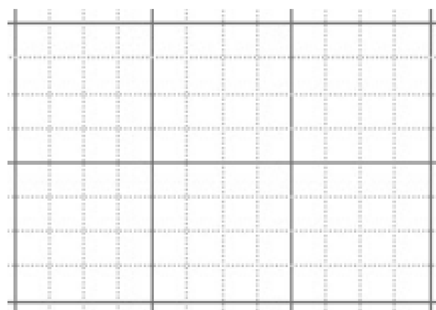
- *Repeatable,* they are defined in ranges like $[0,1]$ or $[-1,1]$, there is no regular or repeated patterns must be shown. They are **isotropic** (means that they are invariant to rotation) and they can be scaled in many dimensions (1,2,3, …).
- *Continuous functions which appear random.*

Perlin Noise

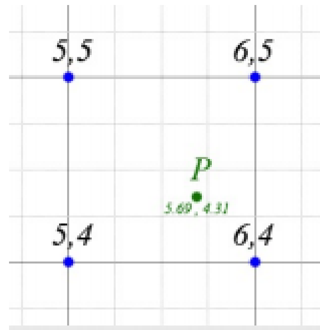
This is the most common kind of noise, proposed in **1985 from Ken Perlin**, used in the original movie of **Tron**. Until the end of 2007 and so on it put online a set of slides explaining the procedure of Perlin noise.

How it works:

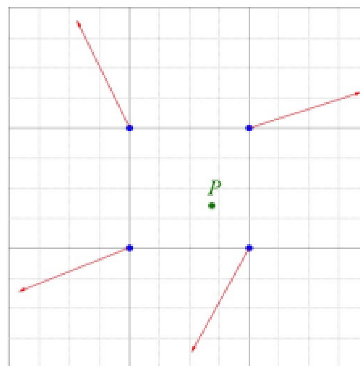
We start from a regular grid of points, which is called "*lattice*".



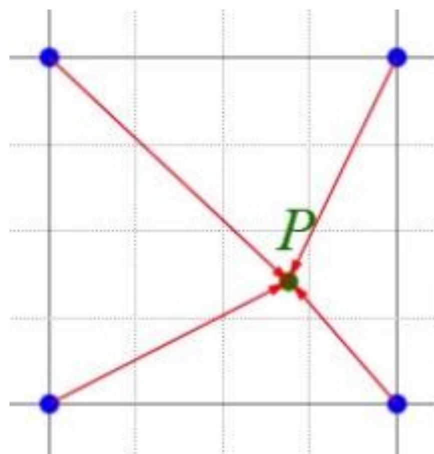
Then for an input point $P(x, y)$, you consider also the four points on the corners where the cell of the grid is placed.



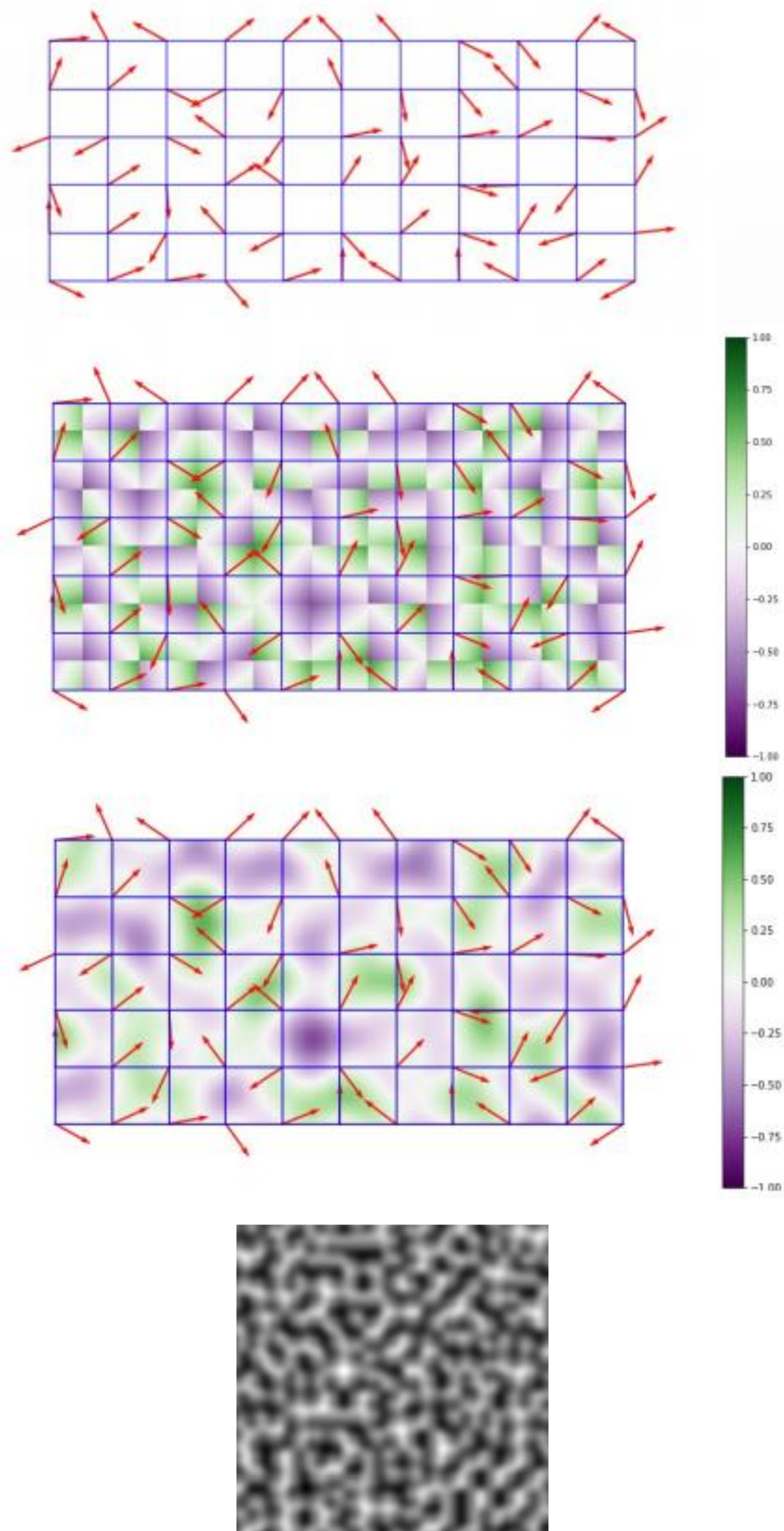
Then there will be created a **pseudo-random gradient** associated to each one of the corner-points.



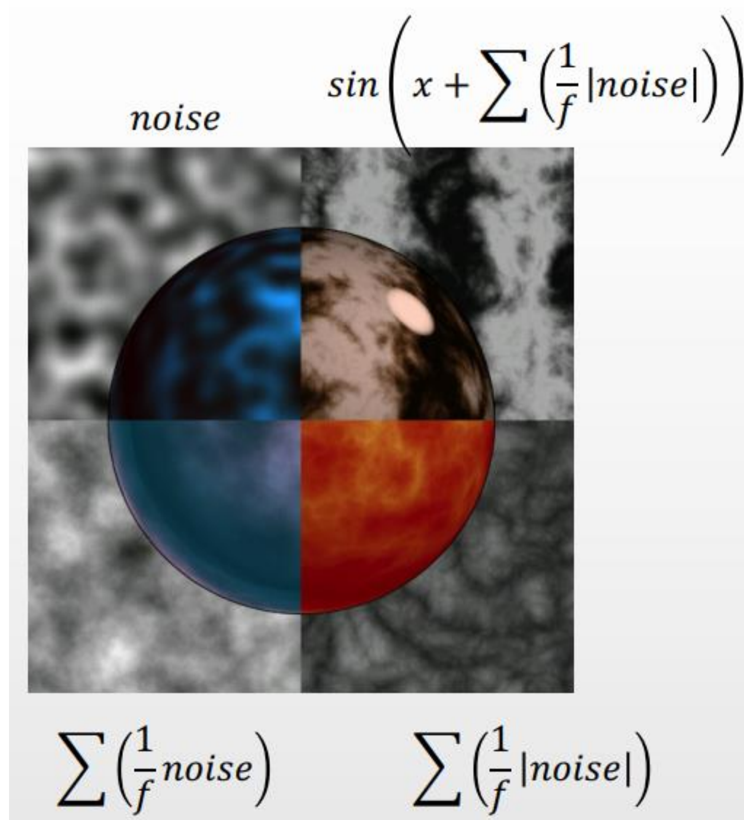
Also, will be created a new set of **four vectors** that starts from the points on the corners and points to the input point P.



The **noise** is **created** by the **combination** of the **dot product** between the **internal vectors** (gradients) and the **external vectors.**



It is possible to achieve different version of the noise by applying different functions or displacement values to the original noise (which is used as a basis for other noises).



Turbulence

The idea of summing different execution of noise with different parameters is called **turbulence**. Different noise executions are called ***octaves***. As said previously, these octaves will have different parameters.

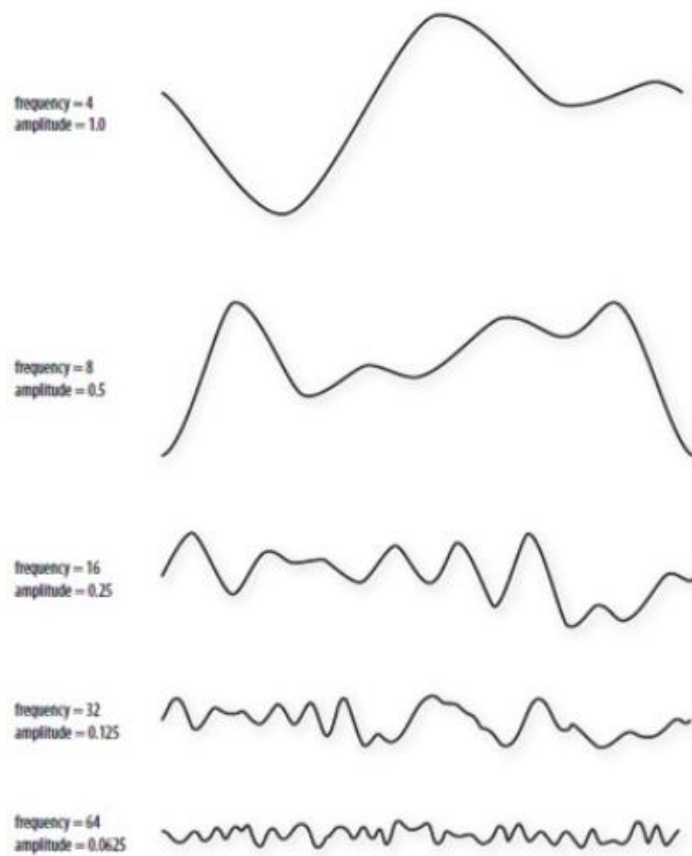
This is taken from **Fractal Brownian** motion :

$$Noise(x) = \sum_{i=0}^{N} \frac{1}{2^i} \cdot noise(2^i \cdot x)$$

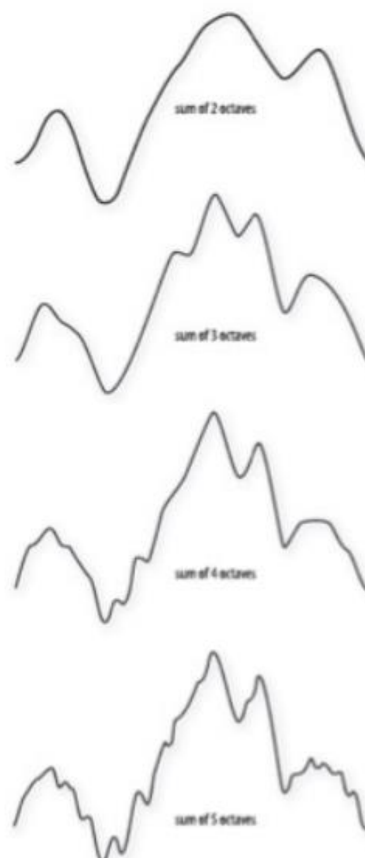
Is possible to consider also an **Amplitude-frequency system**, which gives a greater control and flexibility of the final effect.

$$Noise(x) = \sum_{i=0}^{N} \frac{amplitude}{2^i} \cdot noise(2^i \cdot x \cdot frequency)$$

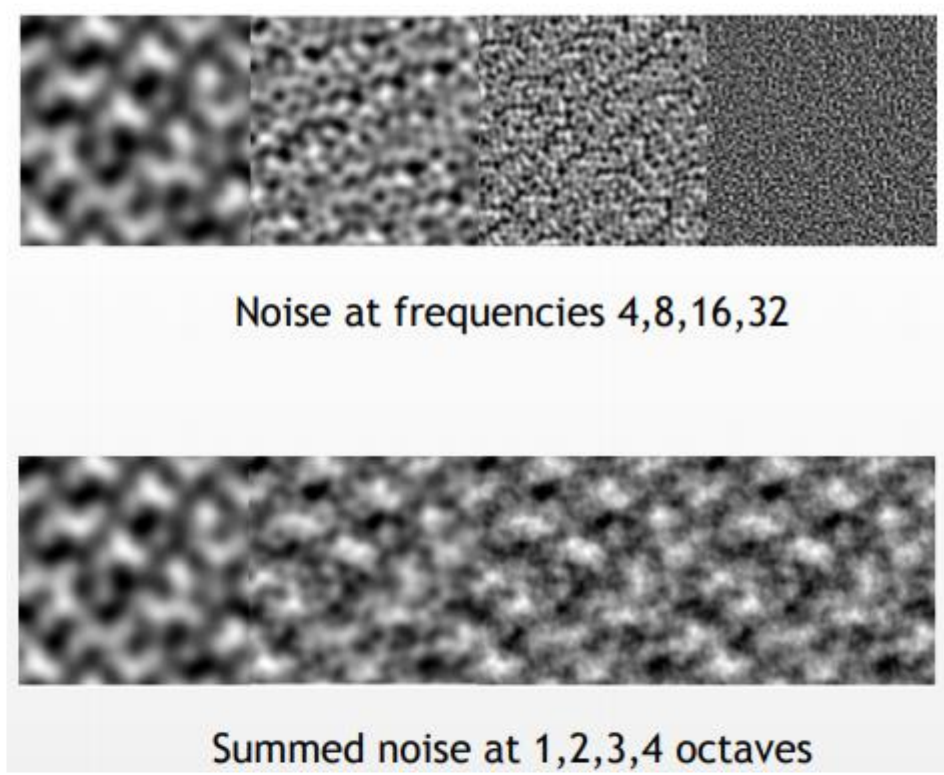
This because we can use two parameters to edit the noise function



Then the octaves with different amplitude and parameters will get summed



2D representation



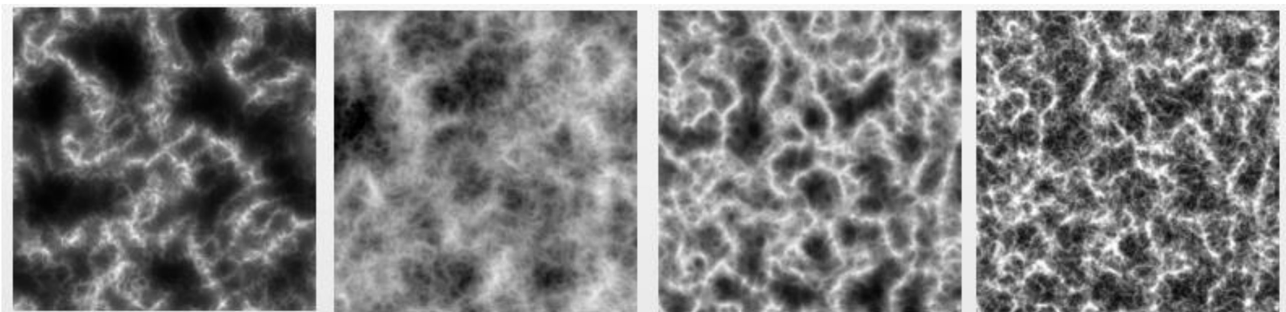
This system offers a very large "**gamut**" of possibilities. There is other expansion of the basic Perlin noise, the simplex noise and the multifractal.

Simplex noise (variant of Perlin Noise)

Proposed by Perlin, is a more complex extension of the Perlin noise and it is multidimensional. Proposed to **avoid directional artifacts**. In some cases, it could be visible some part of the noise which seems more oriented in a specific direction rather than completely random.

Multifractal (variant of Perlin Noise)

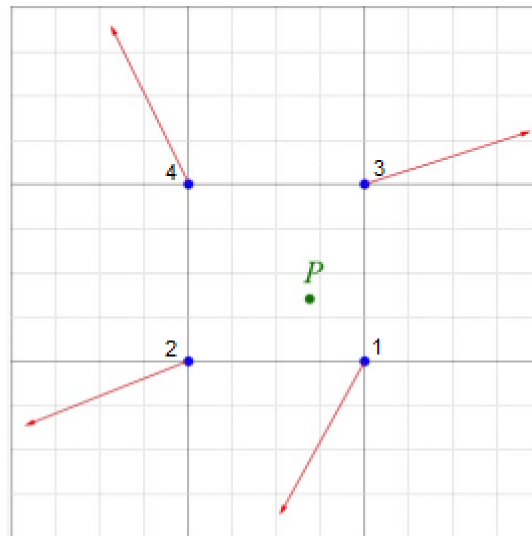
The functions are coming from the **fractal field** of mathematics, in this case there is more variability and contrast and a more ridged effect.



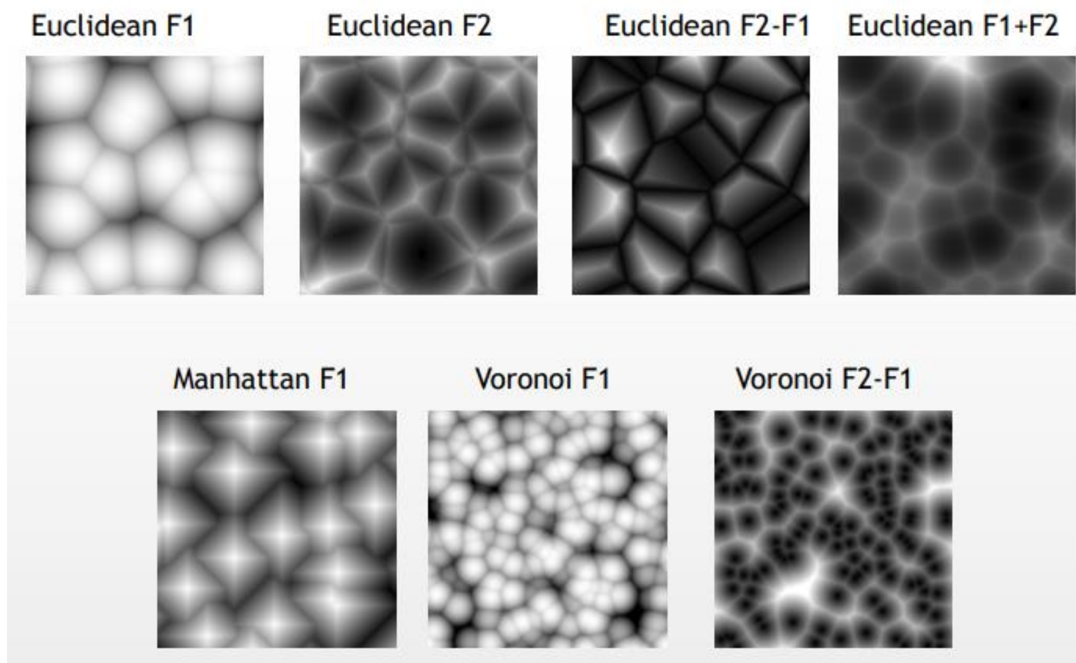
Worley noise

Also called **Cellular Noise** (or **Voronoi**), has a completely different approach.

Given a grid of points and an input $P(x, y)$. The distance between the input P and the $n^{th} - closest$ point F on the grid as the basis for noise calculation.



I use this distance as a basis for the noise calculation. This noise is **more computationally expansive then Perlin Noise**, but we usually don't apply the sum octaves. It is mainly based on which **closest point to consider and the function used for the distance** (*Euclidean, Manhattan, Voronoi, …*) and I can also consider more closest point and I can apply operation above them.



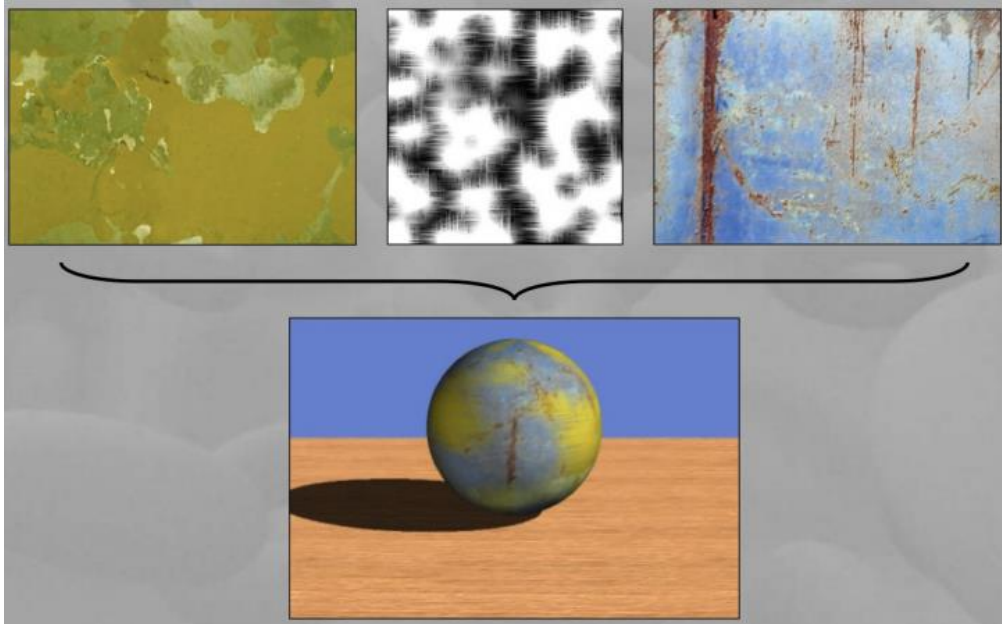
GLSL and noise

You may think that we have a predefined noise function in **GLSL**, well no. This is a very interesting situation, because in the specification you set some rules and then the developers develop the hardware.

They never implemented a noise function; they never reached an agreement of which kind of noise to implement as default in the noise function. In some implementation of **GLSL** we have a function call $noise()$ but it **returns zero**, it never got implemented.

Layered shader

We have two layers with different materials, and then we have something to blend them together. **This is the result of the procedural function**. Both layer and mask can be generated procedurally.



Global illumination techniques

When we talk about **global illumination** in videogames we may be sure that we are not talking about the global illumination as in the movies, is not the same of **full path-tracing** approach of movies.

In real-time graphics we are talking about the same visual effects given by global illumination like *shadows, reflections, indirect illumination components,…*

In the **real time** you always rely on **local illumination model**, all these effects contribute to the realism of the image are usually added through **additional techniques**.

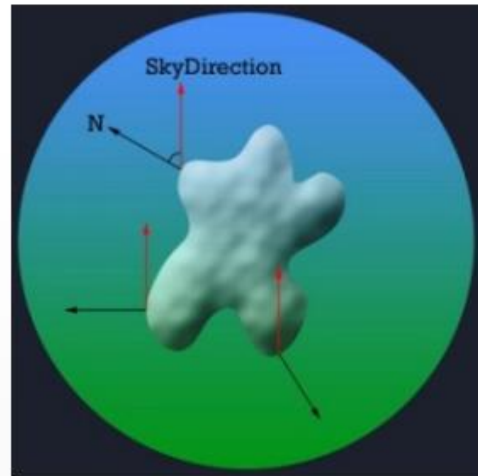
The **illumination models** compute locally the color, then the visual realism is obtained through additional techniques like *shadows, reflections, simulation of indirect illumination components* **are introduced as additional techniques** (and not a global illumination engine like a *path-tracer*) since they are not directly linked with the **illumination model**. This different use of techniques provides different levels of realism.

The state of the art in the last years was the use of **textures mainly to add the effects rather than the illumination**, often textures who are **pre-baked**.

Hemisphere lighting

Very simple way for **simulate global illumination**. In the real world the light bounces on the ground and illuminates you from the bottom, for example if you are in a garden with the grass, the light coming from the bottom is **greenish.**

It is possible to simulate this by considering your environment as being inside two hemispheres, one with the color of the sky and one with the color of the ground. You then decide the color of the object by interpolating between the sky and the ground color.



$$C = a \cdot skyColor + (1 - a) \cdot groundColor$$

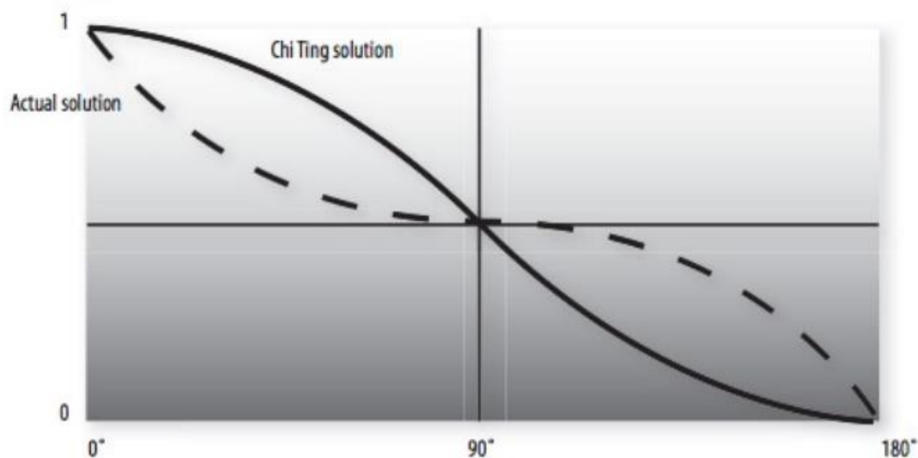
$$a = 1.0 - (0.5 sin\theta) \qquad if\ \theta \leqslant 90°$$

$$a = 0.5 sin\theta \qquad if\ \theta > 90°$$

It is a simple linear interpolation between sky color and ground color.

However, to compute it faster and to avoid the ***if-statement*** in the shaders (which is not computationally efficient to have in the shaders, since it is GPU-code), instead of calculating the a considering the sin you consider the cos which is really fast because it is a dot product.

The cos will provides the same are under the curve but will be computed faster then the sin (*dot product).*



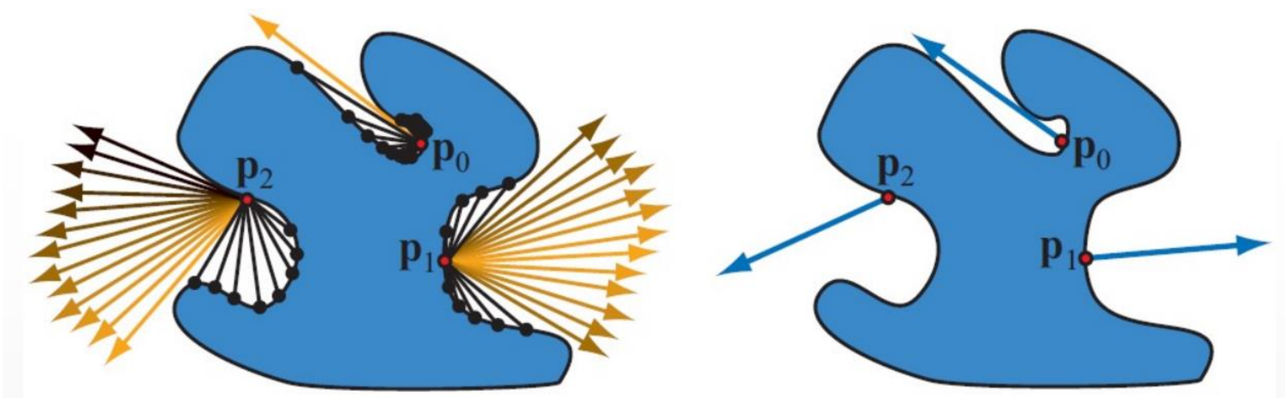
It is more related to **diffusive reflection**, it can be integrated with other light sources like then sun, lamps and so on.

One of the issues of the **hemisphere lighting** is that the **self-occlusion** is not very well simulated. The self-occlusion is the idea that an object always **cast a shadow on itself**, and in some cases the phenomenon is called **ambient occlusion**.

Ambient occlusion

In the bunny we consider **every point as independent**, where the fragment of the ear is computed in the same way of the fragment of the paw. But what if the ear produces a shadow on the paw of the bunny, now the color of that fragment is dependent by an effect given from the ear fragment.

The **ambient occlusion** tries to darken some area of the object in the scene because we have a self-occlusion, and it is used for render in a better way the shading of the object.



If we treat the fragments of an object as they are all **mutually independent** (image on the right), then this means that we have an effect which is not really realistic.The problem of the local models is that they don't consider the **inter-reflections**, the ambient component in the **Phong model** is a **very rough way to approximate the indirect illumination**.

The **classic ambient occlusion** technique was developed by Landis in 2000, at Industrial Light & Magic, to improve the quality of the environment lighting used on the computer-generated planes in the movie *Pearl Harbor*.

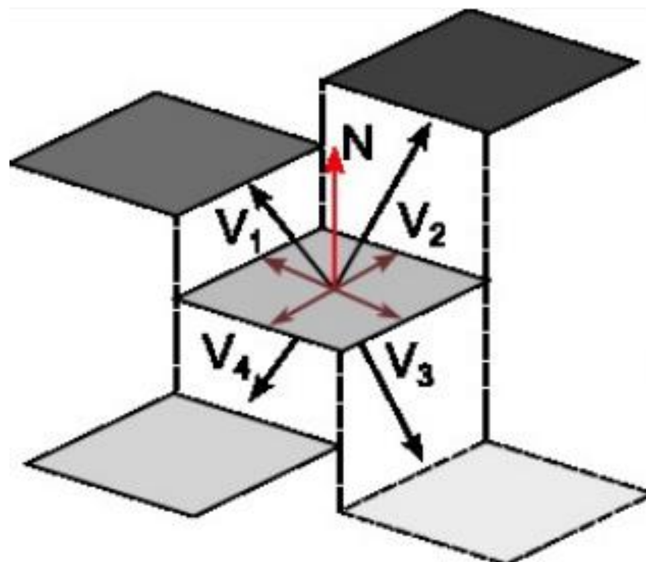
does these sequential operations for a single point (*image on the left*) :

- **Ray-casting** of lot of rays on the hemisphere (from the side where comes the light) where is centered the point.
- **Calculate the distances of the intersection points**.
- **Weights these distances** and **combine** them in an **attenuation factor**.
- The attenuation factor is used during the computation of the illumination model.

For example, $\boldsymbol{p_1}$ is a bit more open the $\boldsymbol{p_2}$ and $\boldsymbol{p_0}$ the idea is that i combine al lthe directions i wconsider the main overall vector given by the avarage of everything, and we use that vector for the computation of illumination.

SSAO (Screen Space Ambient Occlusion)

Screen Space Ambient Occlusion, as the name says this technique is screen space based, and it has been proposed by **Crytek**, it occurs during the **fragment shader**.



We have this potential fragment (small pixels oriented to the camera), I have the depth of the fragment (obtained during the Z-buffer depths), I have these different depths of fragments. If I take this fragment and I compare the depth with the neighbor fragments, I can consider how this fragment is occluded or not to respect to the others.

This technique is based on an **analysis of the depth buffer** of the scene, and it computes this ambient component in a dynamic way (not like the Phong) by adjusting it in base of every fragment of the scene.

The fragments to test are selected with a spherical **kernel** which is randomly rotated at each step.

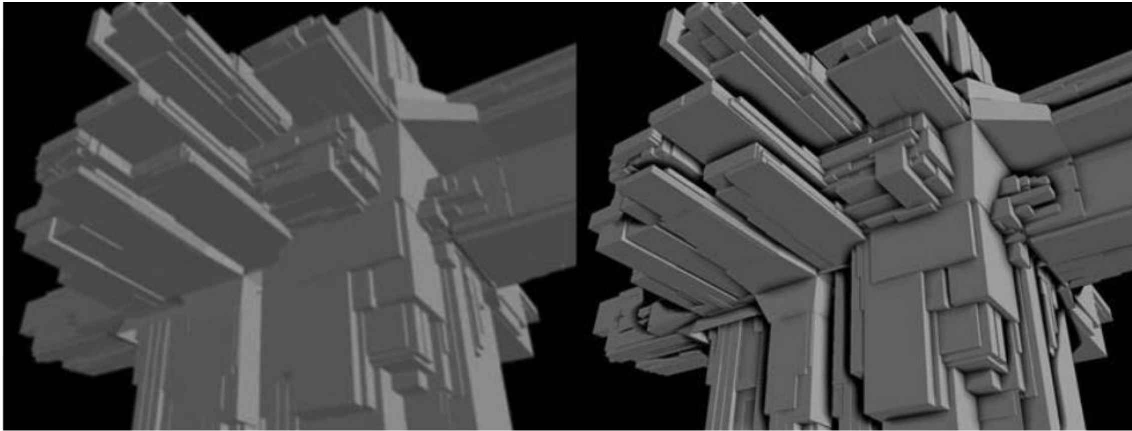


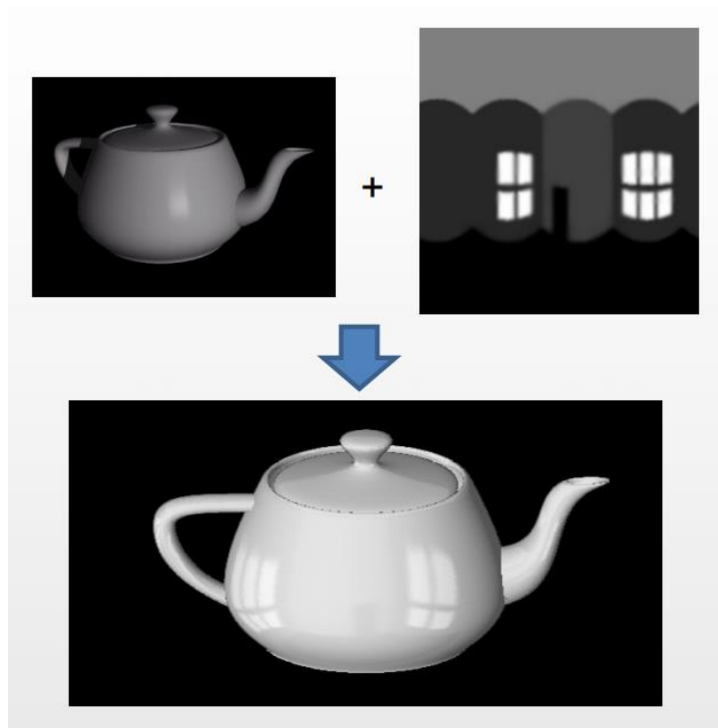
Image-based techniques

These techniques are based on the use of **additional textures**.

Environment mapping

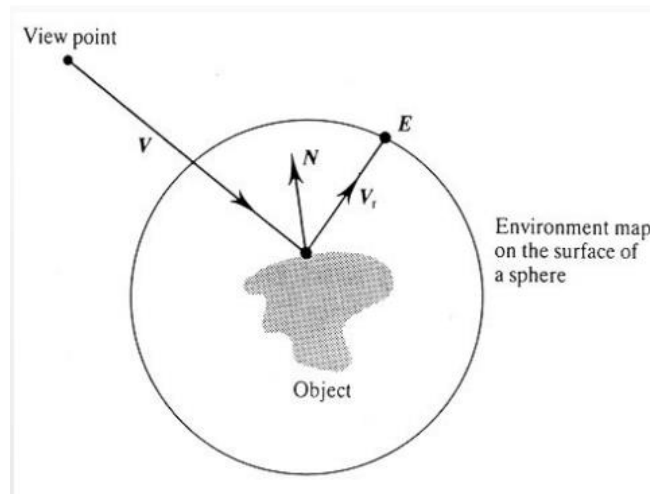
This was proposed as soon as the computational power of the graphics hardware has increased in time. It can be used both for **virtual scenes** that for **real scenes**, let's start with the first one.

I use a **texture which describes an environment**, and i sample this image which represent the virtual environment of where the object is placed, then i sample this texture to simulate **specular effects** of the object. We are not talking about specular lights; we are talking about the mirror effects of an object.



This kind of reflections **natively** generated by **path-tracing**, in that case most of these techniques are not added in the separate moment, they are computed in the overall rendering process. Anyway, I need an in image which is in a format of panoramic image or more peculiar formats than that.

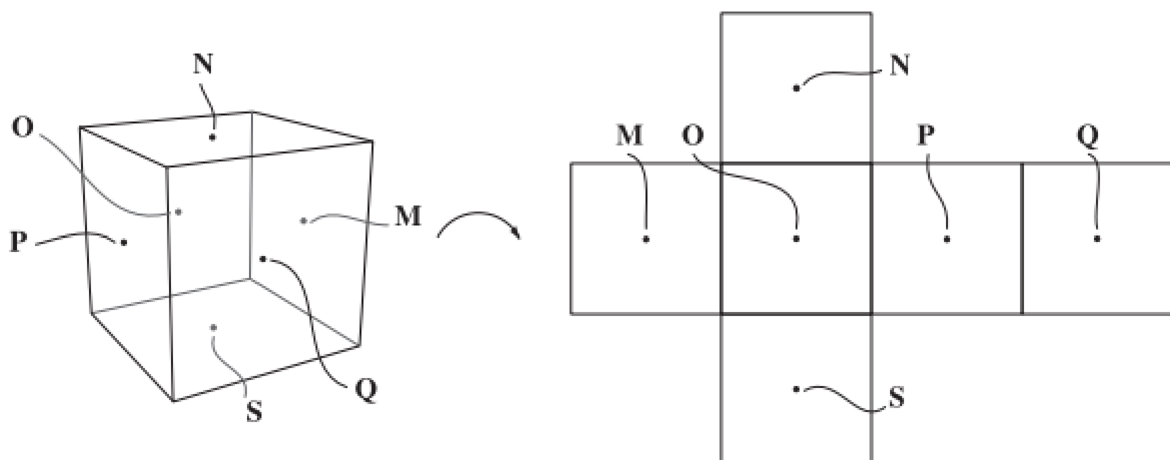
In this technique we don’t assign the texture to the object, we don’t rely on **UVs**, we consider the panoramic image as **mapped on a sphere** and we use a **reflection vector** and his components as **UVs** to read from this sphere/cube.



We compute the **reflection vector** starting from the **view vector**, the reflection vector is used for **sampling the environment**. Then we use that value to bland it with the result of an illumination model. The map can be **pre-filtered** to apply blur, in this way is possible to simulate more glossy effects.

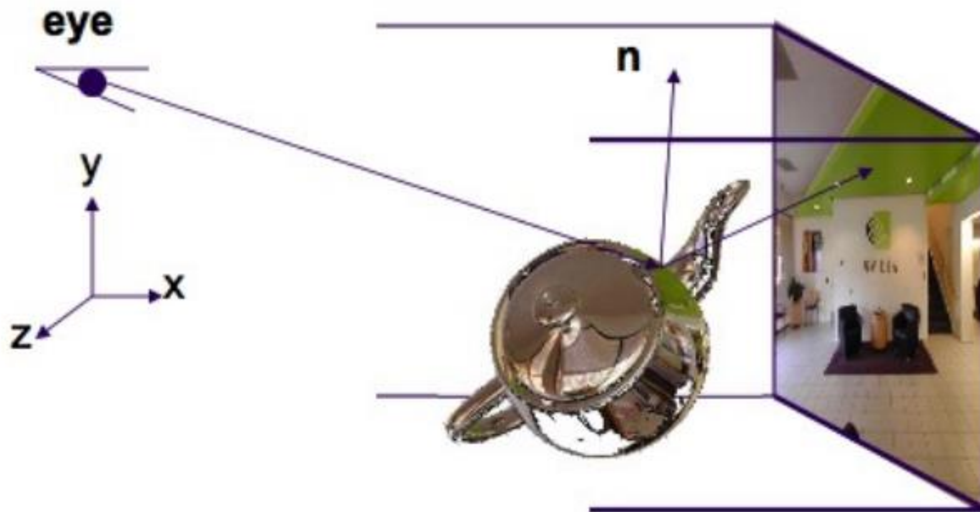
In **1986**, **Chris Green** introduced the *cubic environment map*, usually called *cube map*. This method is far and away the most popular method today, and its projection is implemented directly in **hardware** on modern **GPUs**.

The **cubemap** is created projecting the environment onto the sides of a cube positioned with its center at the camera’s location. The images on the cube’s faces are then used as environment maps. Usually, a cube map is often visualized in a “*cross*” *diagram*, i.e., opening a cube and flattening it onto a plane.



Anyway, on **hardware cube maps are stored as six square textures**, in this way there is **no wasted space**.

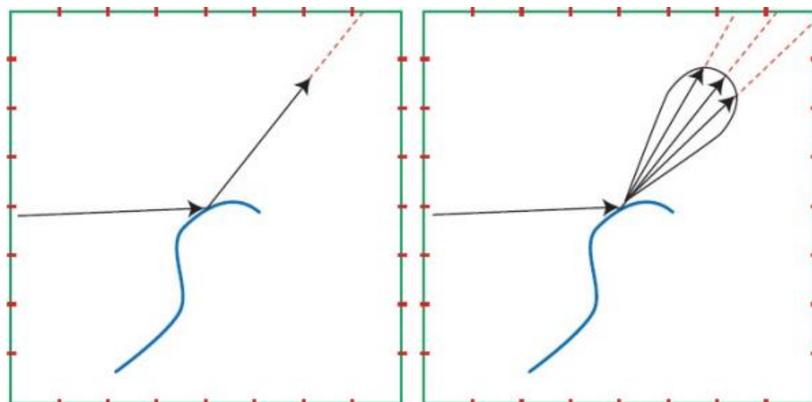
It is possible to create cube maps synthetically by rendering a scene **six times with** the **camera** at the **center** of the **cube**, looking at each cube face with $90°$ **view angle**. Cubic environment mapping is **view-independent**.



This environment map can be an image of a **real scene** or it can be a reflection of complete virtual environment. In case of **real scenes**, the first time that the environment map was used for real scenes was in Terminator 2. for real scenes

Now we have lot of different techniques, they are all based on **photographic acquisition**. We can take lot of pictures and then **stick them together** with an **imaging software**, or we can use **lens** with a very large **FOV** (*fisheye*), or i can use a **mirrored ball**.

Once i have the environment map i can do a fully reflective effect but is possible also to simulate a blurred effect, there are two possible approaches.



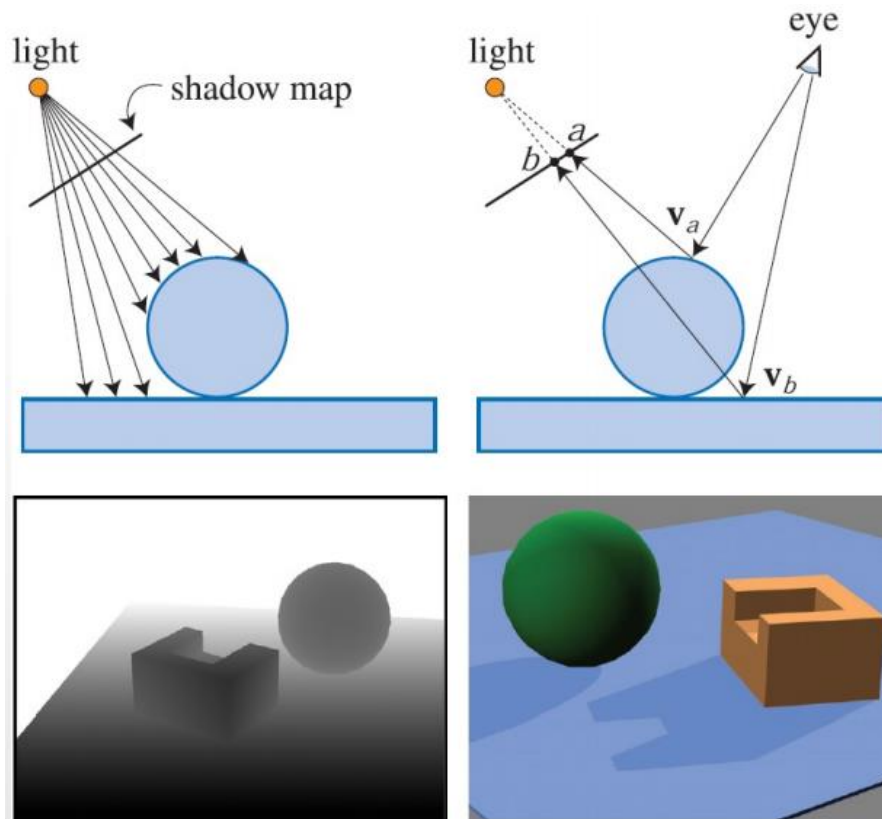
1. I can take **one texel** and i use that value, for the blurred effect theoretically i can apply a blur directly on the map, anyway there are issues at the borders of the cube map.

2. I can use a **gaussian function** or **cosine function** to sample the map, this means that I don't sample just one direction but i also sample some other texels and I average together resulting in a blurred effect.

Shadow map

Shadows are something which is really relevant to the realism of a virtual scene, but **they aren't computed natively by local illumination model**, this is calculated in a separated way.

This technique is based on two passages, I will use my GPU pipeline two times :



1. **Move the camera in the position of the light**, and store everything from the POV of the light, **saving the Z-buffer image**. What is in the shadow of the scene, won't be visible in the depth buffer image. An object is considered fully illuminated if there is nothing between the light and the object, this means that everything which is visible in the depth image (light **POV**) is illuminated by the light.

2. I use my real camera, and when i have to compute the illumination in each fragment i compute the **illumination model**, then i check if that position in the scene is **present or not in the auxiliary buffer** from the **POV** of light, if it is present the point is illuminated otherwise it is in shadow.

   Will create the final frame using the data saved in the first step for the final effect.

Pros

- Shadow map creation has a **linear computational cost**.
- **Constant-time access to shadow map**, once it is created you will have just to read a texture.

Cons

- Has all the techniques that use **more than one GPU pass**, we are using the **GPU two times for one frame**, this means that everything must be well done and implemented.

Percentage Closer Filtering (PCF)

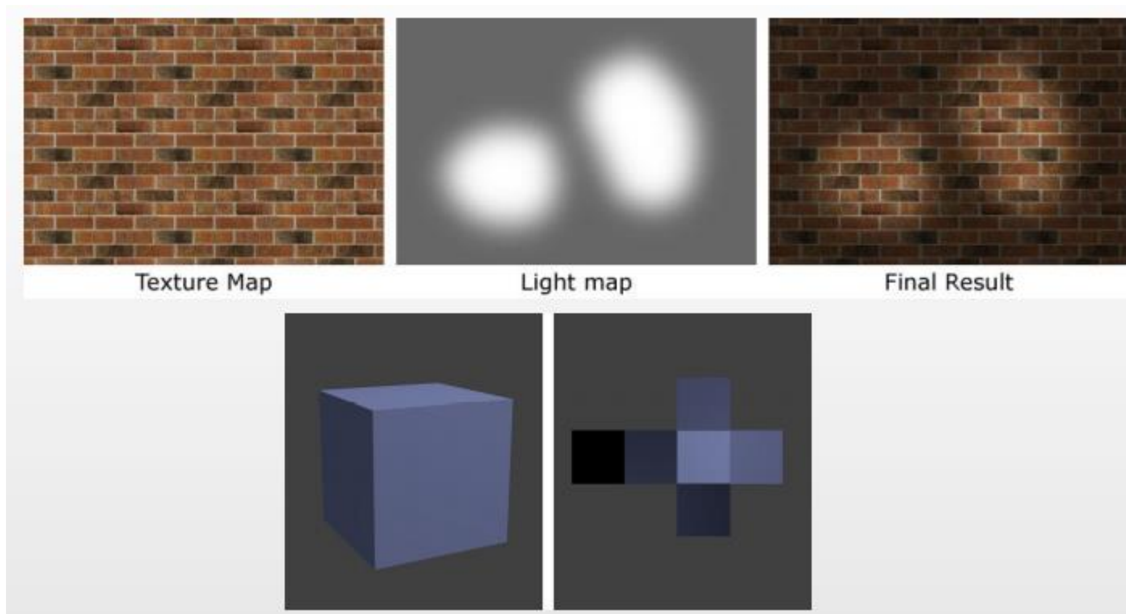
In normal shadow mapping we have jagged effects, since we magnify the texture when we apply it on the scene.

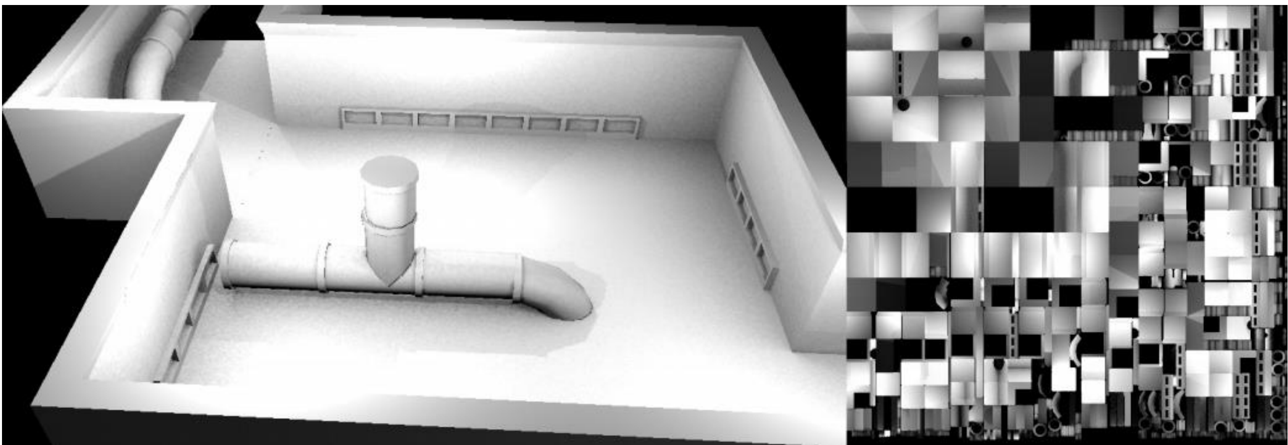
It is possible to apply some blurred effect, or we can apply the **PCF**, which consists in a **filter**, it samples more time the shadow map pixels and it average them together. Won't result in the best possible effect but is much better than normal shadow mapping.

Light maps

If i know that my scene has lot **of static meshes and lot of lights which are static**, then there are lot of illumination components in my scene which will be the same in every frame, this will be a waste of time to compute again at every frame.

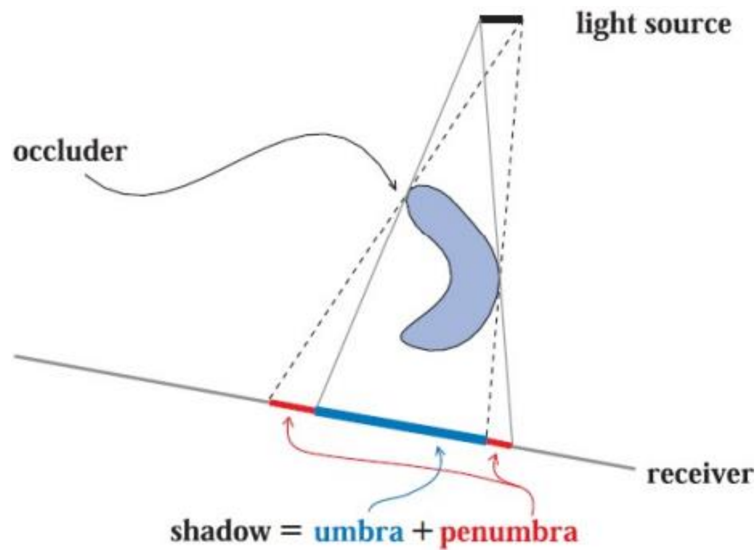
So, i **can save during the creation** of the asset a **pre-baked texture with the light**, and then when I'm in **real-time** i read from this texture and i apply the value to simulate the illumination and maybe i apply in **real-**time the effects which are **dynamic**, and I combine them with the baked and precomputed data.



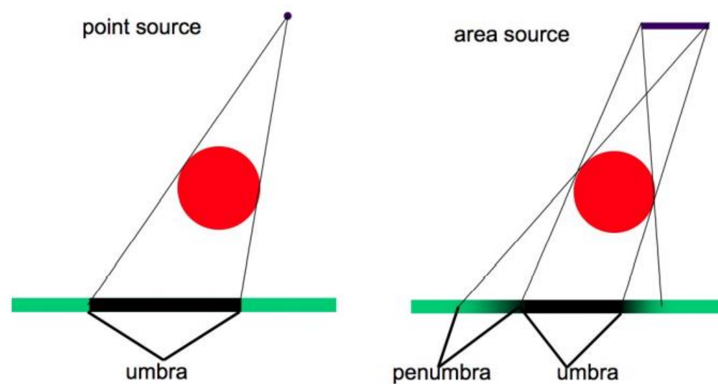


Shadow (definition)

The object which is casting the shadow is called **occluder**, the surface where the shadow is casted is called **receiver**.



Then usually the shadow is composed by **umbra** (the full shadow) and **penumbra** (soft shadow where part of the light arrives).



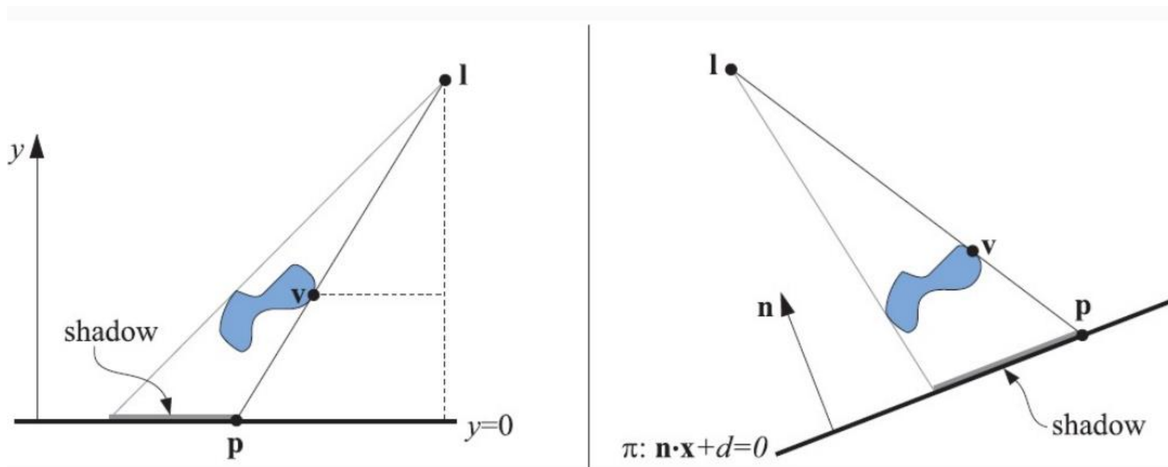
In case of **point light** we can only have **hard shadows** (umbra), we talk about **soft shadows** (penumbra) with **area light.**

Projection-based shadows

We can use **projection-based** approach to shadowing, exploiting the concept of the **umbra penumbra** in the shadows.

We **can determine the range of directions which tightly enclose the occluder**, the **minimum left** and the **maximum right** directions, that will get projected on the receiver (the image on the left).

So, we can create an actual shadow polygon by projecting the vertices of the object on the receiver plane given from the **POV** of the projection point which is the position of the light. In this case we have hard shadow, but it is possible to apply some blur later.

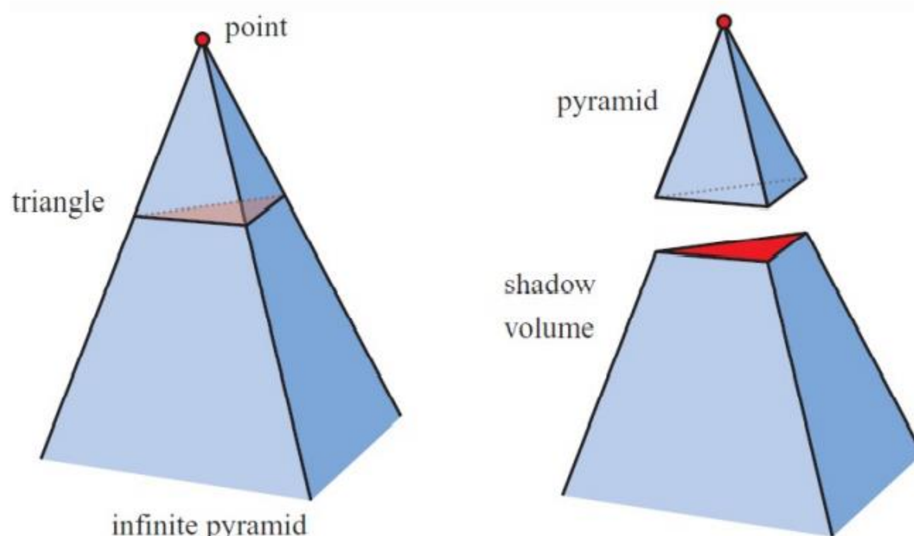


Shadow volume

Not so different as a concept, presented by Heidmann in 1991, it can be used on any GPU, the only requirement is a stencil buffer. This is not an image-based method, like the shadow mapping, this means that avoid the sampling problems producing correct sharp shadows everywhere. Anyway, **shadow volumes** are rarely used nowadays due to their unpredictable cost.

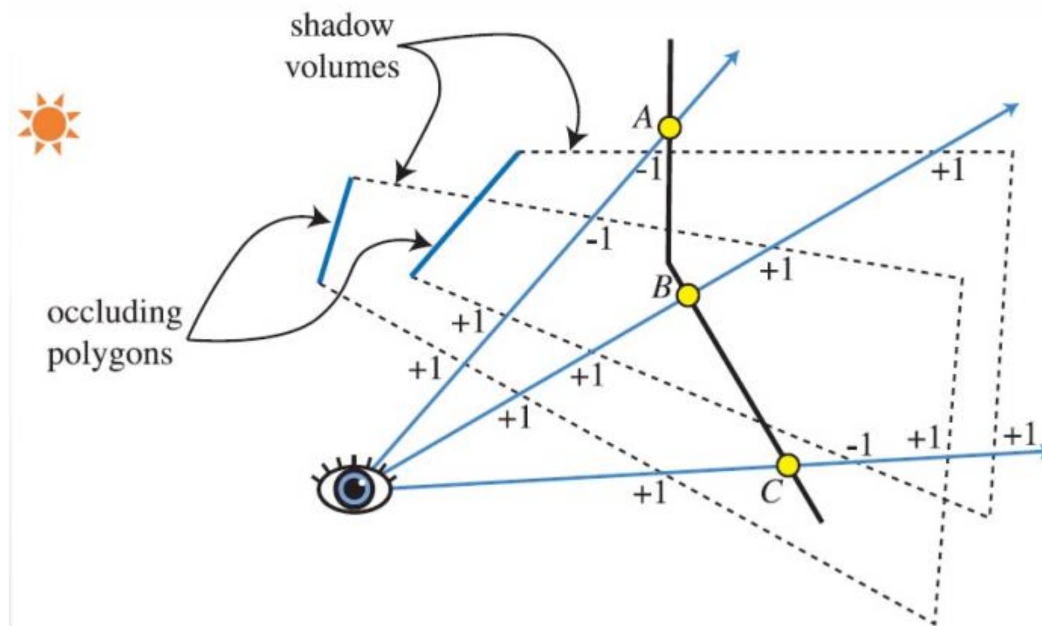
We can consider the **shadow as a volume**, the shadow is made by the **light source** and the **occluder**, initially we have volume of light which is expressed as a pyramid.

Then when an object is placed inside the volume that will cut the volume in two and the volume after the **occluder** will be called **shadow volume**.



After the creation of the shadow volumes from the POV of the light source, we move to the POV of the camera.

The idea is to consider a **ray and a line from the camera to the point to be rendered**, and to count **how many times the ray crosses the faces of the shadow volume**.



Every time the view-ray intersects one side of the truncated pyramid of the shadow volume from the front-face (in respect to the viewer) will occur an increment of the counter by 1. Every time time the **view-ray** intersects a back face i decrement the counter by 1

We keep doing these computations until the view-ray hits the object that we want to display in the final pixel. If the counter is greater than 0 , then that pixel is in shadow otherwise not (is not in the shadow volume).

Anyway, doing that with the rays is time consuming, there is better solution that involves the stencil buffer, which will do the count for us.

Image based lighting (IBL)

In the case of illumination another approach that has seen a lot of interest in the last years, is the use of **image-based lighting** effects. With diffusion of digital images, with possibility to acquire high quality images we have also proposed the use of images as illumination sources.

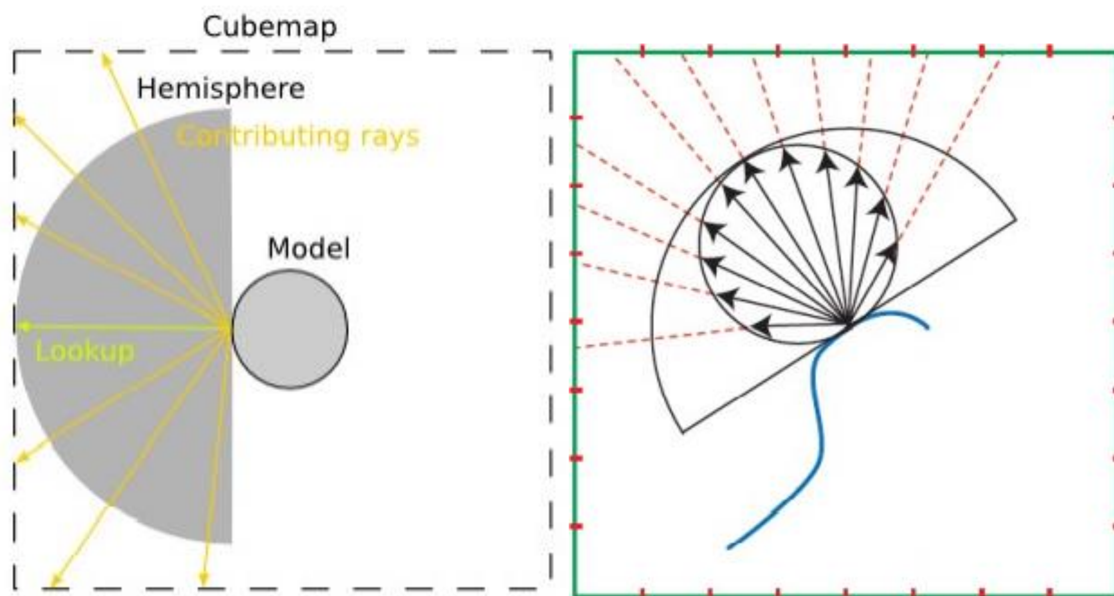
This 3D rendering technique involves capturing an **omnidirectional representation** of **real-world** light information as an image, typically using a 360° camera. This image is the projected onto a dome or a sphere analogously to **environment mapping** but for describe the illumination.

External images used to describe/calculate illumination of a virtual scene. These images are supposed to store in every pixel a value which is approximating the **real-life** value of the scene.

With the introduction of **HDR** (*High Dynamic Range*) images, which have in every pixel's values greater than the classical $[0, 255]$, they are supposed to store in every pixel a value which is approximating the real light value (the *luminance*) of the scene.

Using that value of luminance, we can use it for illuminating the scene, having as a result virtual objects which seems to be present in real scenes. This is a very effective technique, with very realistic illumination effects.

The idea as the **environment map** I have this **HDR** image, which is called **Irradiance Environment Map**, which is a **Cube Map**, but it is used in a different way respect to the **reflection map**.



What the technique does, is to take every point on my object and to sample the environment map in all the possible direction inside the hemisphere centered on the surface normal of the object.

Every direction will point to an **environment map**, but as said before it won't sample the **reflection vector** it will sample along the normal and all the images of the hemisphere, then I will integrate all the direction as possible centered on the surface and average them together.

It is not something that we can apply computationally because it is too expensive, usually what happens is that i take the environment map, and I pre-apply a filter (**pre-filtering**), which is usually something like **Gaussian filter**.

Then i read from this filtered-environment map using the normal as **UVs** in that case the **texel** is the result of a filter.

Current state of real-time Global Illumination

Actually (2021), we are living in a **transition time**, for the **real-time raytracing**, this means that the present and the future must be considered together until the future will predominate.

They are merging lot of techniques together, for complex scenes with high number of light sources and high number of static and dynamic 3D models they are not possible.

Usually in complex scenes we have lot of precomputation (**pre-baking**) or everything which is static or diffusive, *for example* **deferred shading** is used a lot for scene with **lot of lights** and **lot of dynamic lights**.

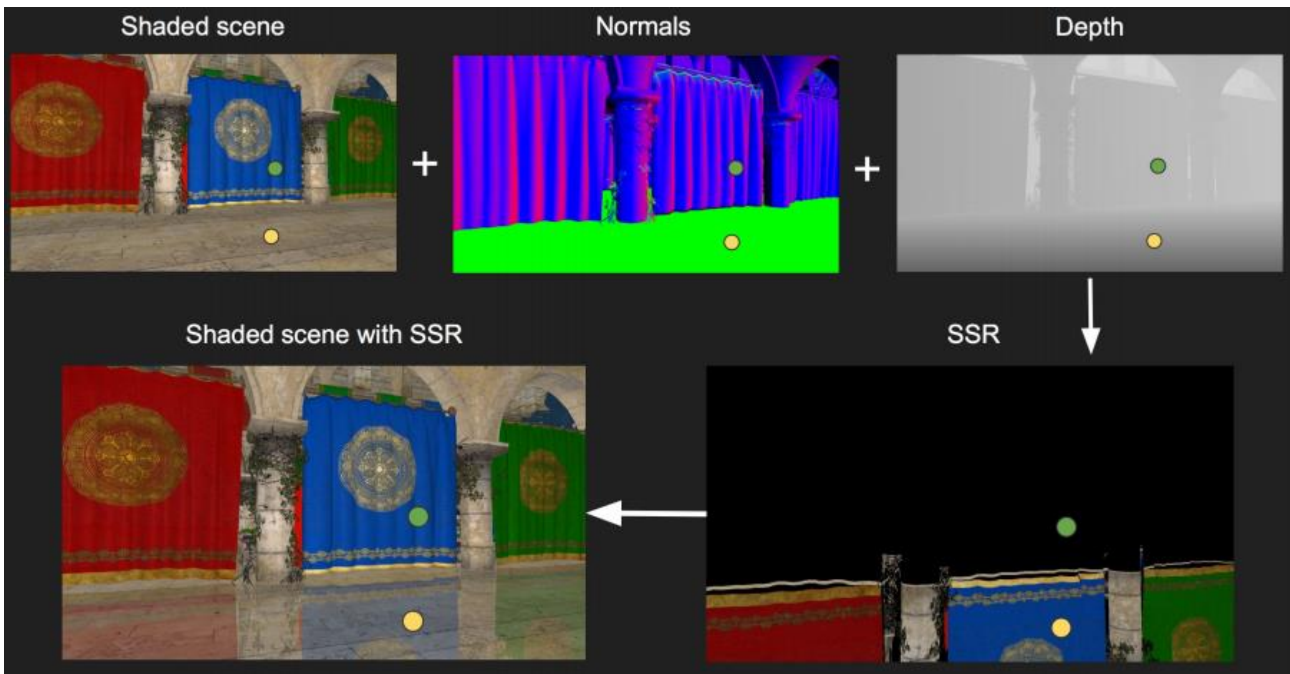
The current **Global Illumination** approaches try to merge and balance different techniques :

- Pre-computation/**pre-baking** of static and diffusive components (like lights and shadow maps).
- Update of baked data only when needed.
- Real-time calculation of dynamic components.
- Relevant use and optimization of **deferred rendering**.

In case of Specular Indirect Lighting are used Environments maps and **Screen Based Reflections** (**SSR**). In the case of Diffuse Indirect Lighting the techniques involved are Light maps (in case of static geometry) and **Irradiance volumes**/**VPL**/**VXGI** for dynamic components.

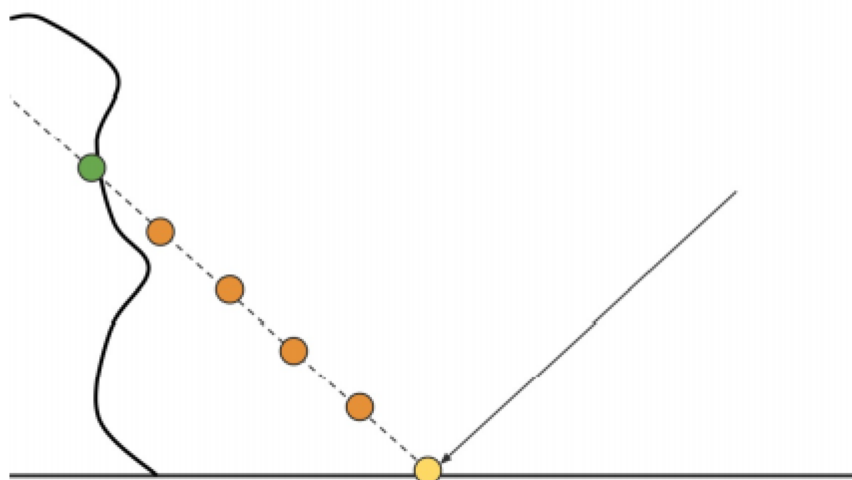
SSR, Screen Based Reflections

The idea, we know that the reflection given by the **environment maps** are limited in case of **dynamic objects** and with **self-reflection**. It is also **time consuming recompute the environment map especially with complex scenes**. The use of **SSR**, use the *fragment shader* and the *deferred rendering* technique to dynamically calculate perfect/glossy specular reflections in these situations.



It uses **deferred rendering** to save data of a certain number of textures, we save **normals and depth in Gbuffers**. In this case the floor is made of very lucid marble and reflects the green point on the yellow point.

I take the fragment and i consider in **screen space** (the fragment and normal is pointing towards the camera), i take the direction of the reflection and i consider in those directions the fragments which are intersected (at this point fragments are small squares placed in depth in front of the camera), **i take all the fragments intersected and i take all the colors and in the end i average everything and i got my reflection.**



We havea good result even for dynamic objects, which is something more convenient then having the reflection map to have updated for each 6 views.

Stochastic Screen Based Reflections

Frostbite has proposed an improvement of this technique, which is more complex, the approach is the same, but we compute also **depth mipmaps.** I use the original approach for the basic reflections, and i compute a hierarchical tracing of rays inside the mipmap of depth for subtle and sharp reflections.

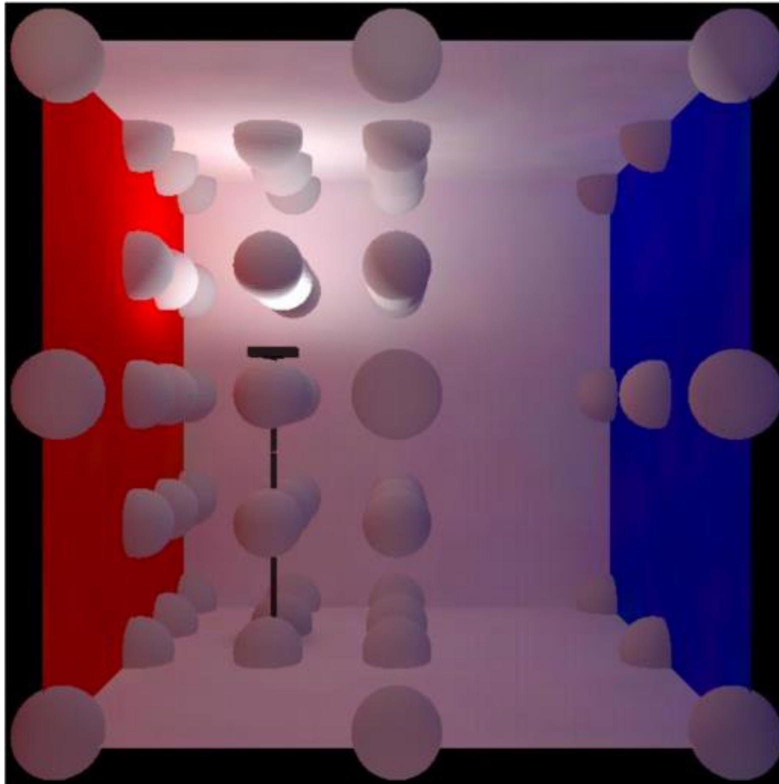
I use also the **BRDF** of the material to determine where to consider the directions of rays, and there is lot of complex optimizations to re-use point of intersections. The first one is the first pass; the bottom light added a rugosity of the terrain for increase the overall quality of the image (look at the finger of the hand of the character).

Irradiance volumes

This technique not so novel but is still used as base for more complex considerations about **global illuminations**, since **light maps works only for static objects**.

The idea is to **subdivide the scene using a grid**, and to generate a cube map for each "static position" defined by a sphere. During rendering when i have an object in certain position of the scene i take the positions on the grid close to that point and i take the cube maps close to that, and i use this **cube maps** to **compute the illumination.**

Is something which has been extended in different ways.



Virtual point lights (VPL)

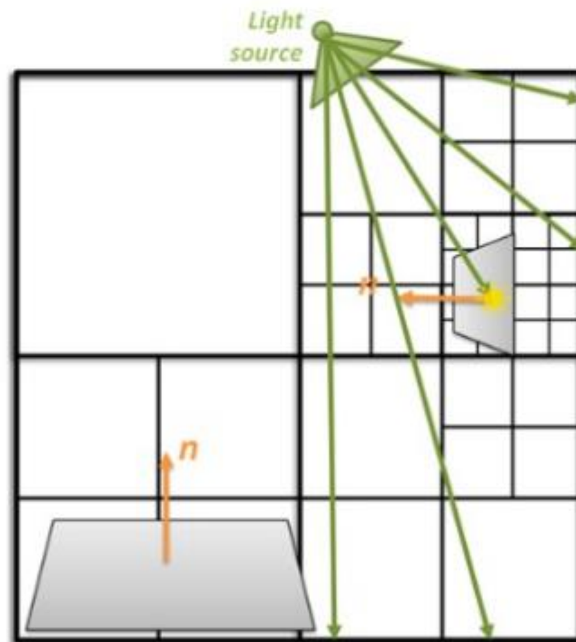
The most used in Global Illumination models combine **static** and **dynamic techniques** for the calculation of **indirect diffuse illumination**.

In Global Illuminations the bouncing of light is directly calculated by the implemented rendering engine, in real-time is being simulated by placing **VPL**, which are **classical point lights** but that **aren't linked to the actual real position** of the light sources, but they are placed on the walls in **order to illuminate the object from the side in a similar way to the bouncing of lights.**

This means that they are lights which are created for simulate the bouncing of another light source.

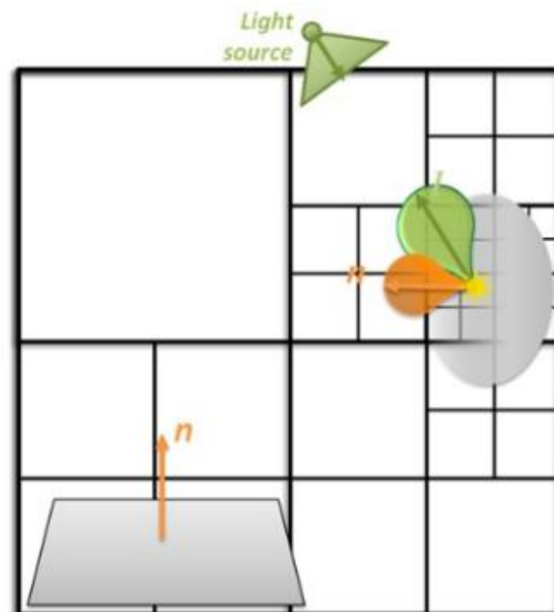
NVIDIA VXGI (Voxel Global Illumination)

Voxel Cone Tracing implementation, a really complicated technique for computing the Global Illumination and still maintain high performance in real-time graphics.



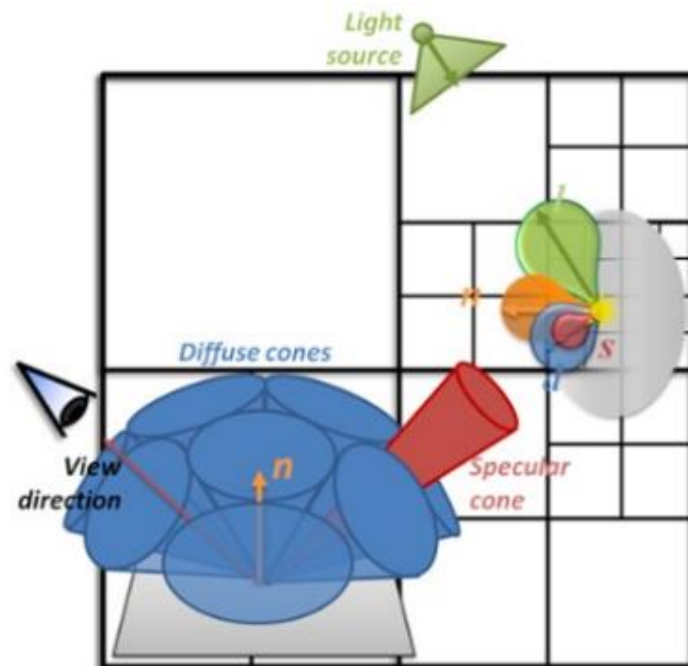
Step 1: Render from light sources. Bake incoming radiance and light direction into the octree

The render starts from the light sources, the scene is subdivided in **Voxels**, which is a **hierarchical structure** similar to **octrees** where is subdivide more where the scene is more computationally dense. **Then the incoming radiance and light direction is baked inside the Voxel**, this is done for every Voxels, this is an approximation that tells how strong the light of a Voxel is.



Step 2: Filter irradiance values and light directions inside the octree

Inside the **Voxel** data structure will occur a filtering of the **irradiance values** and **light directions**.



Step 3: Render from camera. Sample diffuse + specular BRDF components using voxel based cone tracing

Then it is applied a version of ray tracing which is called **cone-tracing**, which is an **approximation of raytracing**, where i consider a **cone of rays oriented in a particular direction**. Along the cones they take the intersection of the cones with Voxels values of the **BRDF** of the object and they place and use together to compute the final illumination. The results are extremely realistic.

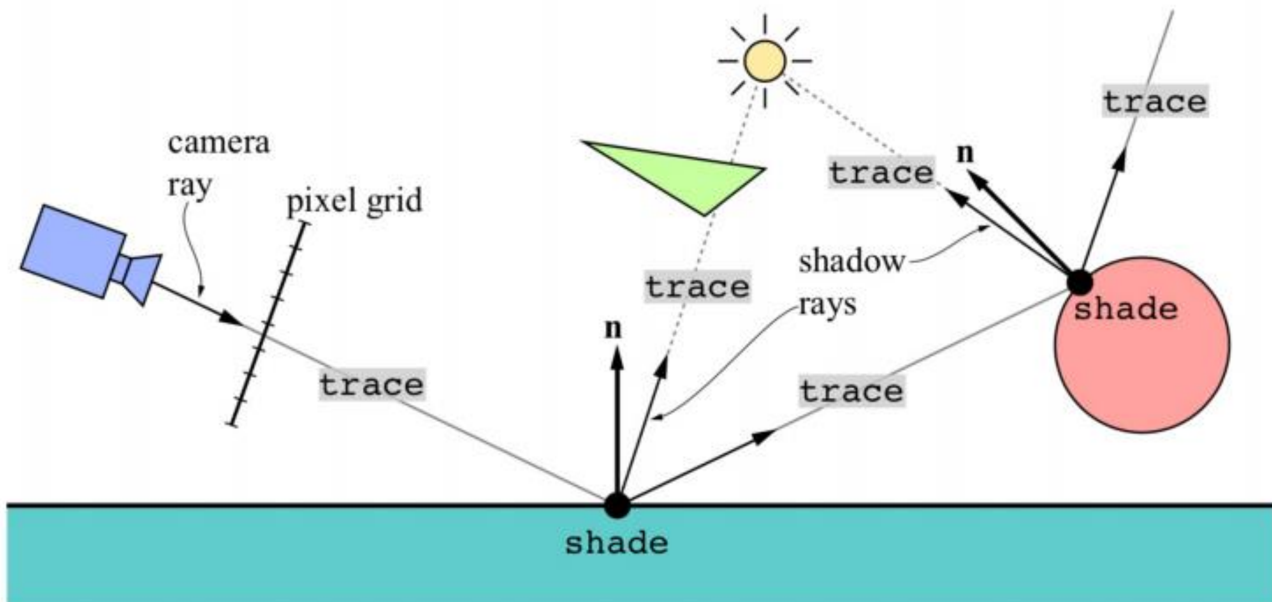
Real-time Raytracing

Every time we talk about real-time graphic and global illumination we talk about of **additional techniques separated from the illumination model**.

In the **Ray Tracing** approach lot of additional techniques that are used in real-time graphics are computed natively. In **Ray Tracing** and **Path Tracing** the reflections, indirect light and shadows are computed natively.

This technique is used obviously based for image-oriented rendering, now the marketing is pushing the "real-time ray tracing" in Videogames. Real time ray tracing in is complete and final form still doesn't exist for videogames.

As soon as GPUs become programmable by Shaders or CUDA, path-tracing and raytracing are perfect for parallel computation, but the same full-raytracing approach is not possible, we have a real-time framerate which is still not achievable with this technique (even with a limited amount of geometry).



A general raytracing presumes that i shoot a **primary ray** from the camera, which will **overpass** the **frame of pixels** (so they are at the beginning of the pipeline and not at the start). Then the **primary ray** will intersect some object, and in consequence there will be a shoot of **shadow rays which will point to the light** and i check if there is one intersection with this one.

If the intersection occurred with an actual light, then the point is illuminated by this one otherwise is in shadow. Other rays that are sent on the previous step is the **reflection ray** which will bounce on another object and so on. **The reflection can be specular or like in the path tracing approach i consider the volume of the BRDF** (**BSDF**) and i search the direction inside the volume. It is possible to consider a refraction ray.

Anyway, the actual **Real-Time Ray Tracing** is not a full implementation of a **path tracer**, with the advancement of computational resources of **GPUs**, was possible to introduce some aspects of **Ray Tracing** inside the **real-time rendering pipeline** (hybrid approach, through ray tracing shaders). The power of these technical solutions introduced a hardware which can perform some part of the operations, the first example comes with the **NVIDIA RTX** platform, following the proposal of the **DirectX Ray Tracing API** (**DXR**) and **Vulkan**.

After the proposal and distribution of first family of **GPUs** supporting ray tracing also the new consoles **PS5** and **Xbox X** they have **graphics hardware** supporting "**Ray Tracing**".

We have not switched together to ray shaders, full **real-time rendering using ray tracing is not possible at the moment**. **There is the need of several rays per-pixel for avoid the strong noise**. **Denoising** techniques for real-time ray tracing are heavily investigated but still not good enough for full real-time rendering.

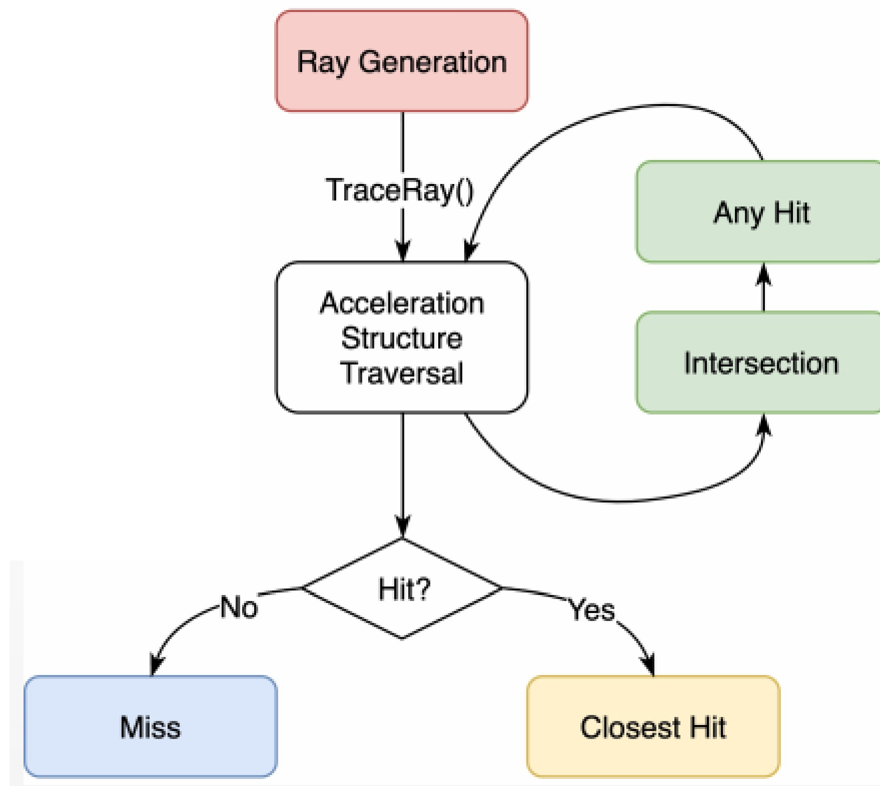
On the contrary, several techniques and stages of rasterization-based GPU pipeline are heavily optimized like **rasterization**, **texture operations**, **deferred rendering** and so on. The **current rasterization-based pipeline is the result of decades of optimization**, lot of work of optimization is already done and working greatly.

Shaders in DXR

The idea was to keep what is performed well by the pipeline and create a **hybrid pipeline** which exploits classic rasterization techniques and also using the new **Ray Shaders** for lot of techniques

(*reflections, refractions, ambient occlusion…*) like the one we have seen, **which are a lot easier to perform with rays.**

The idea of raytracing shaders is to try to discard this **complex additional time-consuming raster techniques** and introduce **partially the use of raytracing** because they are easier to use and simulate these effects with rays.



What happens is this, with the **DXR** we have this set of shaders :

- **Ray Generation Shader** : it is the main function which **shoots the primary rays** from the camera. This process of shooting rays is performed using **separated buffers** in a way **similar to the compute shaders** (which is something that helps to perform **GPGPU** techniques inside the real-time rendering pipeline).

- The **Intersection Shader**, which works on two **acceleration structure** which are created specifically with the use of raytracing, they are called **BLAS** and **TLAS** (the first for the *bottom-level* and the other for the *top-level*).

- We can have inside the intersection shader approach a **Closest Hit Shader**, which performs an intersection test and gives back the closest point.

- **Any Hit Shader**, which considers all the intersections along a ray (e.g.).

- **Miss Shader**, it performs some operation if there is no intersection.

Final considerations

Research on real-time Global Illumination is very active right now, there are two main approach the **rendering on game consoles** and the **rendering on PC**.

Until 2 years ago we had **PS4** and **XBOX ONE** generation and on game console you can't update the hardware obviously, this will make the **user** and **developer stuck** to work on the same hardware for lot of years. In the case they would introduce a new technology again you have to consider the generation of the console.

PS5 and **XBOX X** were not available, and lot of tricks for simulating the global illumination were used. However, on **PC** the possibility of implement new technology is much higher, since the GPU hardware is continuously updated. Anyway, **GPU are really expensive**, and only a small percentage of **enthusiasts** can change the hardware with that fast pace. Recently proposed Global Illumination techniques are often implemented exploring the potentialities of the last GPUs. Some time is needed before they are included in production game engines.