**PURE**

**PR00F’s Ultimate Rendering Engine**

# Introduction

Attention! You are reading the internal documentation which contains more information that you might possibly interested in. If you are only interested in the API usage and want to avoid reading about internals, generate an external documentation using PURE\_Doxyfile\_external.

## Motivation

Since my mid-childhood (~13 years old), I have always wanted to create not only PC games but also a 3D graphics engine. Inspired by the success of id Software’s Quake III engine (id Tech 3), I am making my own now. The aim is to create & continuously develop a cross-platform codebase giving nice image quality with strong performance on newer hardware while maintaining compatibility with older hardware as well. The latter is important for me since I am interested in learning old-school techniques from the age of the first 3D-accelerators. ☺ Some parts of the engine was written from ground zero, some other parts are open-source libraries utilized by the engine.

## Copy & Paste

This documentation describes how my engine works and explains relevant technologies. It heavily quotes from ExtremeTech’s 3D Pipeline Tutorial article and other sources. I am not marking such copy-pasted sentences or paragraphs one-by-one with citation marks, but at the same time I am NOT declaring the content of this documentation to be 100% my intellectual product. The text was formed by putting together parts of different articles and my additions into logical order so that it gives a good structured knowledge base in this area, especially for me if I happen to forget some of it a few months later. ☺

## Sources

This documentation contains intellectual properties of various sources, listed below:

* <https://www.opengl.org/wiki>
* <https://www.extremetech.com/computing/49076-extremetech-3d-pipeline-tutorial>
* <http://www.techpowerup.com/gpudb/>
* <https://fgiesen.wordpress.com/2011/07/09/a-trip-through-the-graphics-pipeline-2011-index/>
* <http://developer.amd.com/wordpress/media/2012/10/Depth_in-depth.pdf>
* <https://en.wikipedia.org/wiki/HyperZ>

**Table of Contents**

[Introduction 1](#_Toc55947668)

[Motivation 1](#_Toc55947669)

[Copy & Paste 1](#_Toc55947670)

[Sources 1](#_Toc55947671)

[Engine Usage 3](#_Toc55947672)

[Initialization 3](#_Toc55947673)

[Loading Resources 3](#_Toc55947674)

[Rendering 3](#_Toc55947675)

[Shutdown 3](#_Toc55947676)

[Samples 3](#_Toc55947677)

[Rendering Architecture & Pipeline 4](#_Toc55947678)

[Geometry Stage 4](#_Toc55947679)

[Rasterization / Triangle Setup Stage 9](#_Toc55947680)

[Rasterization Stage 9](#_Toc55947681)

[Glossary 14](#_Toc55947682)

[Do’s and don’t’s 23](#_Toc55947683)

[Sources 24](#_Toc55947684)

# Engine Usage

## Initialization

TODO: add info about how initialization occurs, and how the engine can be initialized (API).

Related PURE API: TODO.

Related OpenGL API: TODO

## Loading Resources

add info about loading or making 3d objects, textures, etc.

## Rendering

how to render

## Shutdown

add info how to shut down.

## Samples

TODO

# Rendering Architecture & Pipeline

In this section, I go through the well-known 3D-rendering pipeline in general while providing PURE-specific information as well.

“3D graphics is the art of cheating without getting caught.” Translated, this means that one of the art-forms in 3D graphics is to elegantly reduce visual detail in a scene so as to gain better performance, but do it in such a way that the viewer doesn’t notice the loss of quality. One quick example of this is culling, which tells the renderer, “If the view camera (the viewer’s eye) can’t see it, don’t bother processing it and only worry about what the view camera can see.”

PURE currently supports fixed function pipeline (i.e. neither vertex- nor fragment shaders) only.

Now we go through the main stages of the 3D pipeline in order.

It is useful to note that most operations in the application/scene stage and the early geometry stage of the pipeline are done per vertex, whereas culling and clipping is done per triangle, and rendering operations are done per pixel.

## Geometry Stage

### Vertex Specification

In this early stage, we define the vertex stream by specifying the vertex attributes (eg. position), the storage of this stream (eg. host memory), and how to interpret the stream (primitive type eg. triangles). The order of the [vertices](#_Vertex) in the stream is important. [Vertices](#_Vertex) can be streamed in the same order as they are actually placed in memory (e.g. vertex array), or in different order specified by [vertex](#_Vertex) indices (e.g. element array). The latter has an advantage on memory consumption and performance, since same (repeating) [vertex](#_Vertex) data can be stored only once while being referred multiple times by the same index.

TODO: add PPP info on this.

### Vertex Processing

Vertices are transformed from object-space to clip-space. Modeling-, view-, and projection transformations on the vertices including optional normals are done. These are calculated on the GPU nowadays thanks to HW T&L.

OpenGL transformations in general:

* 4x4 float matrices are used as transformation matrices;
* transformation matrices are stored as 1D arrays in column-major order (!);
* matrix multiplications happen in reverse order, e.g. setting a perspective projection matrix then translating it results in translating first, then multiplying by the projection matrix;
* <http://www.songho.ca/opengl/gl_transform.html> ;
* <https://www.opengl.org/archives/resources/faq/technical/transformations.htm> .

The result of calculations done in this stage can be checked in [PR00FPSvsPURE Transformations.xlsx.](PR00FPSvsPRRE%20Transformations.xlsx)

#### Modeling Transformation

Transforming the [vertices](#_Vertex) from **object/model-space to world-space**. Simple matrix multiplication.

Model / Object Space: where each model is in its own coordinate system, whose origin is some point on the model, such as the right foot of a soccer player model. Also, the model will typically have a control point or “handle”. To move the model, the 3D renderer only has to move the control point, because model space coordinates of the object remain constant relative to its control point. Additionally, by using that same “handle”, the object can be rotated.

World Space: where models are placed in the actual 3D world, in a unified world coordinate system. The OpenGL API doesn’t really have a world space.

#### View Transformation

Transforming the [vertices](#_Vertex) from **world-space to eye-space/view-space** (simulating a viewer/camera). Simple matrix multiplication.

View Space (also called Camera Space): in this space, the view camera is positioned by the application (through the graphics API) at some point in the 3D world coordinate system, if it is being used. The world space coordinate system is then transformed, such that the camera (your eye point) is now at the origin of the coordinate system, looking straight down the z-axis into the scene. If world space is bypassed, then the scene is transformed directly into view space, with the camera similarly placed at the origin and looking straight down the z-axis. Whether z values are increasing or decreasing as you move forward away from the camera into the scene is up to the programmer, but for now assume that z values are increasing as you look into the scene down the z-axis. Note that culling, back-face culling, and lighting operations can be done in view space.

Note: in OpenGL, we have a combined **ModelView matrix** by a **Model- and View Matrix**. See more at <http://www.songho.ca/opengl/gl_transform.html#modelview> .

Normals are also transformed from **object-space to eye-space/view-space** but in a little different way. See more at <http://www.songho.ca/opengl/gl_normaltransform.html> .

Vertex normals are consumed by the pipeline in this space by the lighting equation.

nVidia’s Chief Scientist Dave Kirk: “lighting is the luminance value, whereas shading is about reflectance and/or transmittance.” These are related to lighting, but shading calculations occur later in the pipeline after rasterization, and we’ll cover the topic later.

Generate (if necessary) and transform texture coordinates.

#### Projection Transformation

Transforming the [vertices](#_Vertex) from **eye-space to clip-space**. Simple matrix multiplication. The projection matrix defines the **viewing frustum** and the **projection mode (perspective or orthogonal)**. See more at:

* <http://www.songho.ca/opengl/gl_transform.html#projection>
* <http://www.songho.ca/opengl/gl_projectionmatrix.html>
* <https://www.opengl.org/wiki/GluPerspective_code>
* <https://www.opengl.org/sdk/docs/man2/xhtml/gluPerspective.xml>

Clip Space: Similar to View Space, but the frustum is now “squished” into a unit cube, with the x and y coordinates normalized to a range between –1 and 1, and z is between 0 and 1, which simplifies clipping calculations. The clipping planes are now orthogonal (perpendicular) to the axes of the space.

The view volume is actually created by a projection, which as the name suggests, “projects the scene” in front of the camera. In this sense, it’s a kind of role reversal in that the camera now becomes a projector, and the scene’s view volume is defined in relation to the camera. Think of the camera as a kind of holographic projector, but instead of projecting a 3D image into air, it instead projects the 3D scene “into” your monitor. The shape of this view volume is either rectangular (called a parallel projection), or pyramidal (called a perspective projection), and this latter volume is called a view frustum (also commonly called frustrum, though frustum is the more current designation).

The view volume defines what the camera will see, but just as importantly, it defines what the camera won’t see, and in so doing, many objects models and parts of the world can be discarded, sparing both 3D chip cycles and memory bandwidth.

The frustum actually looks like an pyramid with its top cut off. The top of the inverted pyramid projection is closest to the camera’s viewpoint and radiates outward. The top of the frustum is called the near (or front) clipping plane and the back is called the far (or back) clipping plane. The entire rendered 3D scene must fit between the near and far clipping planes, and also be bounded by the sides and top of the frustum. If triangles of the model (or parts of the world space) falls outside the frustum, they won’t be processed. Similarly, if a triangle is partly inside and partly outside the frustrum the external portion will be clipped off at the frustum boundary, and thus the term clipping. Though the view space frustum has clipping planes, clipping is actually performed when the frustum is transformed to clip space.

Note: using OpenGL either right- or left-handed viewing system can be used. PURE uses left-handed coordinate system by avoiding gluPerspective().

See more at <https://anteru.net/2011/12/27/1830/> .  
Projection matrix tricks: <http://www.terathon.com/gdc07_lengyel.pdf> .

Related OpenGL API: gluPerspective(), gluLookAt(), glFrustum().

Related PURE API: TODO.

### Primitive (Triangle) Assembly

Primitives are assembled from the vertices coming from the previous stage. Vertices are transformed from clip-space to screen/window-space.

Some say the Clipping, Perspective Divide and Viewport Transformation are not in this stage but in a separate stage called “Vertex Post-processing”.

#### Clipping

[Primitives](#_Primitive) are clipped to the clipping volume (viewing volume/frustum with user-defined clip planes).

In this stage, actually 3 things can happen to a [primitive](#_Primitive):

* discarded (culled), when entirely outside of the viewing volume/frustum;
* clipped (calculating new [vertex](#_Vertex) coordinates as appropriate) when partially outside of the viewing volume. This can generate more than 1 triangle from 1 triangle if required;
* leave unchanged, when entirely inside the clipping volume.

Actually not all triangles that are partially outside of the viewing volume may be clipped, check about **guard-band clipping**: <https://fgiesen.wordpress.com/2011/07/05/a-trip-through-the-graphics-pipeline-2011-part-5/> .

The clipping behavior against the Z-coordinate of the [vertices](#_Vertex) can be modified by enabling **depth clamping**. If enabled, clip-space Z-coordinates are not clipped by the near and far planes.

#### Perspective Divide

Transforming **clip coordinates to normalized device coordinates**, into [-1; 1] range.

<http://stackoverflow.com/questions/3255837/z-value-after-perspective-divide-is-always-less-than-1>

#### Viewport Transformation

Transforming **normalized device coordinates to window (screen) coordinates**. Depth values are transformed into [0; 1] range.

See transformation calculations in [PR00FPSvsPURE Transformations.xlsx.](PR00FPSvsPRRE%20Transformations.xlsx)

Screen Space: where the 3D image is converted into x and y 2D screen coordinates for 2D display. Note that z and w coordinates are still retained by the graphics systems for depth/Z-buffering (see Z-buffering section below) and back-face culling before the final render. Note that the conversion of the scene to pixels, called rasterization, has not yet occurred.

Related OpenGL API: glViewPort(), glDepthRange().

Related PURE API: TODO.

#### Face Culling

Applies to triangles only. A triangle can be discarded (culled) based on its facing. This is done by the winding order of the triangle. It can be CW (clockwise) or CCW (counter-clockwise) depending how the triangle’s 3 vertices rotate in order around the center of the triangle.

Note: face culling can be done in either view space (after view transform, checking the angle between the viewing vector and the triangle’s normal vector) or screen space (testing if triangle’s projected normal vector points away or towards the camera).

Related OpenGL API: glFrontFace(), glEnable(GL\_CULL\_FACE), glCullFace().

Related PURE API: TODO.

## Rasterization / Triangle Setup Stage

Fragments are generated in this stage. Triangle setup aka scan-line conversion: finding out which pixels are covered by the incoming triangle, interpolating vertex attributes across the triangle.

Some define the rasterization process as including triangle setup, whereas others view triangle setup as a separate step that precedes the rasterization/rendering stage of the pipeline. Think of triangle setup as the prelude to the rasterization/rendering stage of the pipeline, because it “sets the table” for the rendering operations that will follow.

### Slope/Delta Calculations

First off, the triangle setup operation computes the slope (or steepness) of a triangle edge using vertex information at each of edge’s two endpoints. Using the slope information, an algorithm can calculate x,y values to see which pixels each triangle side (line segment) touches. The process operates horizontal scan line by horizontal scan line. It determines how much the x value of the pixel touched by a given triangle side changes per scan line, and increments it by that value on each subsequent scan-line.

Note that each scan line is the next incremental y coordinate in screen space. The y values of non-vertex points on the triangle edge are approximated by the algorithm, and are floating-point values that typically fall between two integer y values (scan lines). The algorithm finds the nearest y value (scan line number) to assign to y.

Result of the DDA operation is that we now have x,y values for all scan line crossing points of each line segment in a triangle.

Rasterization operations are triangle-based as well. DDA generated the left- and right-hand edges of a triangle’s intersection with a given scan-line. The portion of a scan line that bridges the two triangle edges is called a span.

### Scan-Line Conversion

## Rasterization Stage

TODO.

Related OpenGL API: TODO.

Related PURE API: TODO.

### Fragment Processing

If **early depth-testing** is enabled, depth test can occur before this stage. **Early stencil-testing** also exists. So it may happen that fragment shading won’t be even done.

Color, depth and stencil values are generated from each fragment. Texturing also happens here.

Specifically, color and depth values are interpolated for each pixel. Up until this point, only vertices have had color and depth information, but now that the triangle edge pixels are being created, interpolated color and depth values must also be calculated for those pixels.

These values are interpolated using a weighted average of the color and depth values of the edge’s vertex values, where the color and depth data of edge pixels closer to a given vertex more closely approximate values for that vertex. In addition, the texture coordinates are calculated for use during texture mapping. Similar to color and depth values, the texture coordinates are interpolated as well.

The rasterizer will shade the span based on various shading algorithms.

‘Shading’ is one of those terms that sometimes seems like a semantic football, as noted earlier, Dave Kirk, Chief Scientist at nVidia describes it this way: “Lighting is the luminance value, whereas shading is about reflectance or transmittance.” The three most common shading methods, flat, Gouraud, and Phong operate per triangle, per vertex, and per pixel, respectively.

Flat Shading: The simplest of the three models, here the renderer takes the color values from a triangle’s three vertices (assuming triangles as primitive), and averages those values (or in the case of Direct3D, picks an arbitrary one of the three). The average value is then used to shade the entire triangle. This method is very inexpensive in terms of computations, but this method’s visual cost is that individual triangles are clearly visible, and it disrupts the illusion of creating a single surface out of multiple triangles. (Lathrop, O., The Way Computer Graphics Works, Wiley Computer Publishing, New York, 1997)

• Gouraud Shading: Named after its inventor, Henri Gouraud who developed this technique in 1971 (yes, 1971). It is by far the most common type of shading used in consumer 3D graphics hardware, primarily because of its higher visual quality versus its still-modest computational demands. This technique takes the lighting values at each of a triangle’s three vertices, then interpolates those values across the surface of the triangle (RTR, p. 68). Gouraud shading actually first interpolates between vertices and assigns values along triangle edges, then it interpolates across the scan line based on the interpolated edge crossing values. One of the main advantages to Gouraud is that it smoothes out triangle edges on mesh surfaces, giving objects a more realistic appearance. The disadvantage to Gouraud is that its overall effect suffers on lower triangle-count models, because with fewer vertices, shading detail (specifically peaks and valleys in the intensity) is lost. Additionally, Gouraud shading sometimes loses highlight detail, and fails to capture spotlight effects, and sometimes produces what’s called Mach banding (that looks like stripes at the edges of the triangles)(RTR, p. 69).

• Phong Shading: Also named after its inventor, Phong Biu-Tuong, who published a paper on this technique in 1975. This technique uses shading normals, which are different from geometric normals (see the diagram). Phong shading uses these shading normals, which are stored at each vertex, to interpolate the shading normal at each pixel in the triangle (RTR, p. 68). Recall that a normal defines a vector (which has direction and magnitude (length), but not location). But unlike a surface normal that is perpendicular to a triangle’s surface, a shading normal (also called a vertex normal) actually is an average of the surface normals of its surrounding triangles. Phong shading essentially performs Gouraud lighting at each pixel (instead of at just the three vertices). And similar to the Gouraud shading method of interpolating, Phong shading first interpolates normals along triangle edges, and then interpolates normals across all pixels in a scan line based on the interpolated edge values.

More recently, another per-pixel lighting model has come onto the scene using a technique called dot product texture blending, or DOT3, which debuted in the DirectX 6 version of Direct3D. A prelude to programmable shaders, this technique gains the benefit of higher resolution per-pixel lighting without introducing the overhead of interpolating across an entire triangle. This approach is somewhat similar to Phong shading, but rather than calculating interpolated shading normals for every pixel on the fly, DOT3 instead uses a normal map that contains “canned” per-pixel normal information. Think of a normal map as a kind of texture map. Using this normal map, the renderer can do a lookup of the normals to then calculate the lighting value per pixel. Once the lighting value has been calculated, it is recombined with the original texel color value using a modulate (multiply) operation to produce the final lit, colored, textured pixel. Essentially, DOT3 combines the efficiencies of light maps, wherein you gain an advantage having expensive-to-calculate information (in the case of DOT3 per-pixel normals) “pre-baked” into a normal map rather than having to calculate them on the fly, with the more realistic lighting effect of Phong shading. the per pixel interpolators are used to interpolate the Phong normals across the triangle and DOT3 operations and texture lookups are used to compute the Phong lighting equation at each pixel.

Fog.

Related OpenGL API: TODO.

Related PURE API: TODO.

### Per-Sample Processing

Usual operations of this final stage are depth testing, blending, etc.

Details at: <https://www.opengl.org/wiki/Per-Sample_Processing> .

#### Pixel Ownership Test

This fails and [fragments](#_Fragment) are discarded if the [pixels](#_Pixel) covered by the [fragments](#_Fragment) are covered by another window thus OpenGL doesn’t own these covered [pixels](#_Pixel).

Related OpenGL API: TODO.

Related PURE API: TODO.

#### Scissor Test

Fails if the [fragments](#_Fragment) fall outside of the scissor rectangle.

Related OpenGL API: TODO.

Related PURE API: TODO.

#### Alpha Test

Related OpenGL API: TODO.

Related PURE API: TODO.

#### MSAA (MultiSample AntiAliasing)

This is a method to achieve FSAA (fullscreen antialiasing). More at: <https://www.opengl.org/wiki/Multisampling> .

Related OpenGL API: TODO.

Related PURE API: TODO.

#### Stencil Test

Fails if the specified stencil function fails between the source and destination stencil values. This feature is unsupported by PURE. Related: [HyperZ](#_HyperZ_(ATi)).

Related OpenGL API: TODO.

Related PURE API: TODO.

#### Depth Test

Fails if the specified depth function between the source and destination depth values fails. If depth test passes for a [fragment](#_Fragment) then the Occlusion Query gets updated if there is an active query. Related: [HyperZ](#_HyperZ_(ATi)). More on depth testing and precision:

* <http://learnopengl.com/#!Advanced-OpenGL/Depth-testing>
* <https://developer.nvidia.com/content/depth-precision-visualized>

Related OpenGL API: TODO.

Related PURE API: TODO.

The **Depth Buffer** stores floating point depth values in the [0;1] range. Precision can be 16-, 24- or 32-bit. Description and tricks for depth buffer and testing can be found on the following pages:   
<https://developer.nvidia.com/content/depth-precision-visualized>  
<http://learnopengl.com/#!Advanced-OpenGL/Depth-testing>

#### Blending

Related OpenGL API: TODO.

Related PURE API: TODO.

#### Dithering

When the incoming [fragment](#_Fragment) color can’t be stored exactly due to less precision of the output image, 2 representable colors can be used instead of the incoming color: the one from rounding up and the other from rounding down. It depends on the implementation which will be used. If dithering is enabled, the output color will be selected based on the position of the [fragment](#_Fragment), by varying between the 2 selectable colors. GL\_DITHER

Related OpenGL API: TODO.

Related PURE API: TODO.

#### Logic Operations

Unsupported by PURE.

Related OpenGL API: TODO.

#### Write Mask

Masking off writing to particular buffers. Unsupported by PURE.

Related OpenGL API: TODO.

# Glossary

Intentionally not in alphabetical order.

### Scene

A virtual 2- or 3-dimensional place or space where things happen, e.g. [CJ is chasing some vagos on a train in Los Santos](https://www.youtube.com/watch?v=6y7o3RNgWR8).

### Rendering

Generating an image of a 2- or 3-dimensional scene.

### Pipeline

Sequence of processing tasks arranged so that the output of each task is the input of the next task.

### Graphics / Rendering Pipeline

Software and/or hardware implementation of a [pipeline](#_Pipeline) where the input is a bunch of scene data and the output is an image of the [scene](#_Scene) as a result of the consecutive [rendering](#_Rendering) tasks. More details in [Rendering Pipeline Architecture](#_Rendering_Pipeline_Architecture).

### Pixel

TODO.

### Vertex

TODO.

### Vertex Shader

TODO.

### Pixel / Fragment Shader

TODO.

### Vertex Pipeline / Processor

Execution unit. Executes [vertex shader](#_Vertex_Shader) instructions. 1 pipeline processes 1 vertex.

### Vertex Processing / Shader Unit

SIMD principle. Contains multiple [vertex processors](#_Vertex_Pipeline_/).

### Pixel / Fragment Processor / Pipeline

Execution unit. Executes [pixel / fragment shader](#_Pixel_/_Fragment_1) instructions. 1 pipeline processes 1 [fragment](#_Fragment).

### Fragment Processing / Shader Unit

SIMD principle. Contains multiple [fragment processors](#_Pixel_/_Fragment).

### Unified Shader

TODO. Since Geforce 8xxx / Radeon HD2xxx series.

### Fragment

A screen-space position and some other data like output of a vertex shader.

### Raster Pipeline / ROP / Z-pipe

Execution unit. Executes scissor test, alpha test, stencil test, depth test and blending.

### Raster Operation Unit

Contains multiple [raster pipelines](#_Raster_Pipeline_/).

### Fragment Crossbar

Routes the [fragments](#_Fragment) coming from the [fragment pipelines](#_Pixel_/_Fragment) to the [ROPs](#_Raster_Pipeline_/) (this is needed as the number of [fragment pipelines](#_Pixel_/_Fragment) is not equal to the number of [ROPs](#_Raster_Pipeline_/)).

### Primitive

TODO.

### Texel

TODO.

### Texture

TODO.

### Texture Unit

TODO. The number of texture units defines the maximum number of [textures](#_Texture) accessed at the same time by the same [fragment shader](#_Pixel_/_Fragment_1). **Texture access** is aka **texture lookup** aka **texture fetching**.

### Texture Filtering

TODO.

### Isotropic Filtering

TODO.

### Anisotropic Filtering

TODO.

### Aliasing

TODO.

### Antialiasing, AA

TODO.

### Multisample Antialiasing, MSAA

TODO. The render output units super-sample only the Z buffers and stencil buffers, and using that information get greater geometry detail needed to determine if a pixel covers more than one polygonal object. This saves the pixel/fragment shader from having to render multiple fragments for pixels where the same object covers all of the same sub-pixels in a pixel. This method fails with texture maps which have varying transparency (e.g. a texture map that represents a chain link fence).

#### Quincunx Antialiasing (nVidia)

TODO. A blur filter that shifts the rendered image a half-pixel up and a half-pixel left in order to create sub-pixels which are then averaged together in a diagonal cross pattern, destroying both jagged edges but also some overall image detail.

Since Geforce 3.

#### AccuView Antialiasing (nVidia)

TODO. 4XS. Since Geforce 4.

#### Transparency Antialiasing (nVidia)

TODO. Since Geforce 7.

#### Coverage Sampling Antialiasing, CSAA (nVidia)

TODO. Since Geforce 8.

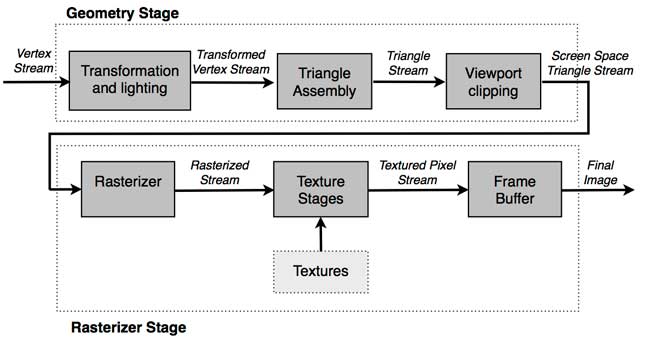
<http://www.nvidia.com/object/coverage-sampled-aa.html>

### Supersampled Antialiasing, SSAA

Render the scene large size internally then scale the result down to the output resolution. Slower than [MSAA](#_MultiSample_Antialiasing,_MSAA).

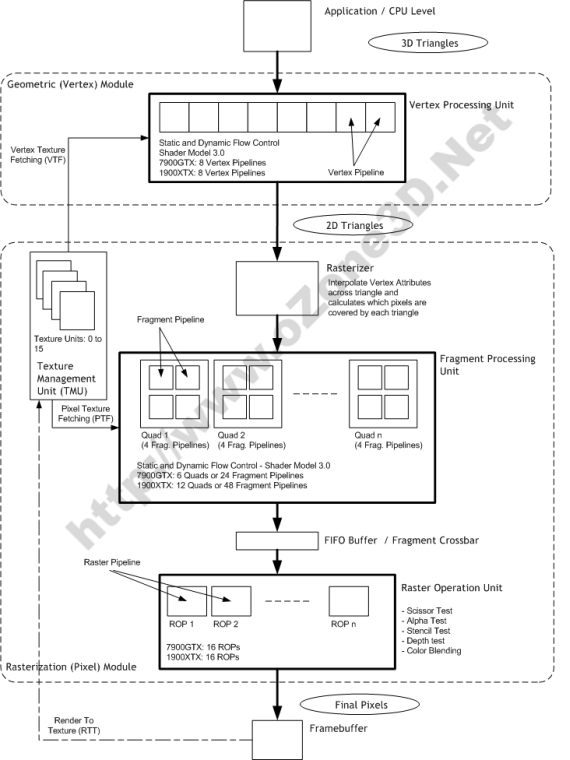
### Rendering Pipeline Architecture

Next I describe the usual steps in a rendering pipeline that produces a rendered image of a 3D object, with the help of the picture below.

source: <http://www.adobe.com/devnet/flashplayer/articles/how-stage3d-works.html>

First we define the [vertex](#_Vertex) stream by specifying the [vertex](#_Vertex) attributes (eg. position), the storage location of this stream (eg. host memory), and how to interpret the stream ([primitive](#_Primitive) type eg. triangles). Modeling-, view-, and projection transformations on the vertices including optional normals are done, so the [vertices](#_Vertex) are transformed from **object-space to clip-space**. [Primitives](#_Primitive) (e.g. triangles) are assembled from the transformed [vertices](#_Vertex), clipping is applied and then they are further transformed from **clip-space to screen/window-space**. Then we find out which pixels are covered by the incoming triangles, and interpolate [vertex](#_Vertex) attributes across the triangle. These generated values are fragments. Color, depth and stencil values are generated from each [fragment](#_Fragment) with the use of optional textures. The final steps include depth testing, blending. etc.

Following picture is an example of a rendering pipeline architecture implemented in HW. A bit more specific than the previous picture.

  
source: <http://www.ozone3d.net/tutorials/gpu_sm3_dx9_3d_pipeline_p02.php>

Note: as seen in the picture, [Rasterizer](#_Rasterization_1) is not equal to [Raster Operation Unit](#_Raster_Operation_Unit).

In some sense, 3D chips have become physical incarnations of the pipeline, where data flows “downstream” from stage to stage. Computations in various stages of the pipeline can be overlapped, for improved performance. For example, because vertices and pixels are mutually independent of one another in both Direct3D and OpenGL, one triangle can be in the geometry stage while another is in the Rasterization stage. Furthermore, computations on two or more vertices in the Geometry stage and two or more pixels (from the same triangle) in the Rasterzation phase can be performed at the same time.

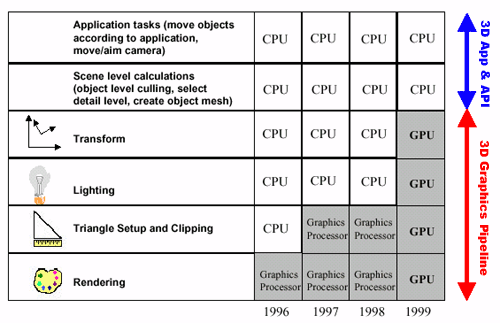
Another advantage of pipelining is that because no data is passed from one vertex to another in the geometry stage or from one pixel to another in the rendering stage, chipmakers have been able to implement multiple pixel pipes and gain considerable performance boosts using parallel processing of these independent entities. It’s also useful to note that the use of pipelining for real-time rendering, though it has many advantages, is not without downsides. For instance, once a triangle is sent down the pipeline, the programmer has pretty much waved goodbye to it. To get status or color/alpha information about that vertex once it’s in the pipe is very expensive in terms of performance, and can cause pipeline stalls, a definite no-no.

### HW T&L - Hardware Transformation and Lighting

Moving the [vertex](#_Vertex) transformation and lighting calculations from SW (CPU) to HW (GPU).

Since Geforce 256.

PURE implicitly benefits of this thanks to the vendors’ OpenGL implementation.

  
source: <http://www.anandtech.com/show/391/5>

#### Pre-Transform (pre-T&L) Cache

Stores the untransformed [vertices](#_Vertex). Optimizations regarding this part of the cache are simply sorting the [vertices](#_Vertex) in order of appearance. Typically extremely large, being able to hold ~64k [vertices](#_Vertex) on a Geforce 3 and up.

#### Post-Transform (post-T&L) Cache

This is a GPU FIFO buffer containing data of [vertices](#_Vertex) that have passed through this stage but not yet converted into [primitive](#_Primitive). Can be used with indexed rendering only (element arrays). 2 [vertices](#_Vertex) are considered equal if their index is the same within the same drawing command. If so, the processing of the current [vertex](#_Vertex) is skipped in this stage and the output of the appropriate previously-processed [vertex](#_Vertex) data is added to the output stream. Varies in size from effectively 10 (actual 16) [vertices](#_Vertex) on GeForce 256, GeForce 2, and GeForce 4 MX chipsets to effectively 18 (actual 24) on GeForce 3 and GeForce 4 Ti chipsets.

AMD Tootle is recommended for optimizing 3D-models: <http://developer.amd.com/tools-and-sdks/archive/legacy-cpu-gpu-tools/amd-tootle/>

There is some other vertex cache optimization: <http://home.comcast.net/~tom_forsyth/papers/fast_vert_cache_opt.html>

<https://www.opengl.org/wiki/Post_Transform_Cache>

<http://www.opentk.com/doc/advanced/vertex-cache-optimization>

### AGP Fast Writes

A method of allowing the CPU to send data directly to the AGP bus without having to use main system memory. Speeds up AGP reads. Doesn’t have effect on AGP writes.

Since Geforce 256.

PURE implicitly benefits if current HW and SW supports Fast Writes.

### HyperZ (ATi)

Z- and Stencil Buffer optimization techniques. See the 3 techniques below.

Since Radeon R100 (7xxx series).

TODO: add how PURE benefits.

#### Hierarchical Z, HiZ

We manage 8x8 blocks of [pixels](#_Pixel) (tiles) and store the maximum (LESS, LEQUAL) or minimum (GREATER, GEQUAL) Z-value for each tile. During [triangle rasterization](#_Rasterization_1), we calculate the minimum/maximum Z-value for each triangle. First we compare this value to the tile’s maximum/minimum Z-value: if the triangle’s minimum/maximum Z-value is greater/less than the tile’s maximum/minimum Z-value, it means the triangle is not visible at all and can be early rejected (**Early Z Reject**). This is much faster than comparing the triangle’s Z-values to the stored values in the depth buffer.

Before the HD2xxx series, this information was stored in on-chip memory. Since the HD2xxx series, it is stored in off-chip memory.

Since the HD2xxx series, HiZ is applied on the stencil buffer as well.

#### Z Compression

If a tile contains very few number (1-2) of triangles, instead of storing all Z-values in that tile, store the plane equation for that triangle(s). So this is a lossless compression. Does not reduce the amount of memory that is required to store the depth buffer. It only saves bandwidth. It will still need to allocate the full buffer to handle all potential uncompressed states.

#### Fast Z Clear

Benefit of Z Compression. Instead of writing the depth clear value across the entire depth buffer, we just reset the state of all tiles to “cleared” (by storing the plane equation for a constant Z=1 triangle).

### Lightspeed Memory Architecture, LMA (nVidia)

Memory bandwidth optimizations including similar features as [HyperZ](#_HyperZ_(ATi)). **Z-Occlusion Culling** ~ [HiZ](#_Hierarchical_Z,_HiZ). **Occlusion query**: determining if the geometry to be rendered will be visible by using a bounding box occlusion test first.

Since Geforce 3.

### Nvidia Shading Rasterizer, NSR (nVidia)

TODO. Programmable pixel pipeline. Since Geforce 2.

### Render to Vertex Buffer, R2VB

TODO. Since Radeon X1xxx series.

### High Dynamic Range Rendering, HDR

TODO. Since Geforce 6xxx series.

### Shadow Buffers (nVidia)

TODO. Since Geforce 3.

### UltraShadow (nVidia)

TODO. Since Geforce FX57xx series.

<http://www.tomshardware.com/reviews/nvidia-geforcefx-5900-ultra,630-4.html>

### UltraShadow II (nVidia)

TODO. Since Geforce 6xxx series.

# Do’s and don’t’s

*“Style guides and usage books don't agree. The Chicago Manual of Style and others recommend dos and don'ts. The Associated Press and others recommend do's and don'ts. Eats, Shoots & Leaves recommends do's and don't's”* - <http://www.quickanddirtytips.com/education/grammar/dos-and-donts>

Don’t change from one of the “less”-based depth-tests to a “greater”-based test or vice versa in the middle of the frame because GPUs turn [HiZ](#_Hierarchical_Z,_HiZ) off until the next clear.

# Sources