**PGE**

**PR00F’s Game Engine**

**Introduction**

Since my mid-childhood (~13 years old), I have always wanted to create not only PC games but also a game engine. Inspired by the success of id Software’s Quake III engine (id Tech 3), I have created my own. My 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’m interested in learning old-school techniques from the age of the first 3D-accelerators. ☺

This document describes how my engine works, and also tells about relevant technologies I know.

No part of this document may be used or reproduced in any form by any means without prior written permission from me, Ádám Szabó.

**Table of Contents**

[Terminology & Technology Knowledge 3](#_Toc474576631)

[Core Knowledge 3](#_Toc474576632)

[Extended Knowledge 4](#_Toc474576633)

[Engine Construct 11](#_Toc474576634)

[PGESysCFG 11](#_Toc474576635)

[PGESysNET 12](#_Toc474576636)

[PGESysGFX 13](#_Toc474576637)

[PGESysSFX 19](#_Toc474576638)

[Do’s and don’t’s 21](#_Toc474576639)

[Sources 22](#_Toc474576640)

# Terminology & Technology Knowledge

Intentionally not in alphabetical order.

## Core Knowledge

### 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.

### Texel

TODO.

### Fragment

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

### Vertex

TODO.

### Primitive

TODO.

### Texture

TODO.

### Texture Filtering

TODO.

### Aliasing

TODO.

### Antialiasing, AA

TODO.

## Extended Knowledge

### Isotropic Filtering

TODO.

### Anisotropic Filtering

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).

### Vertex 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 Shader

TODO.

### 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.

### 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

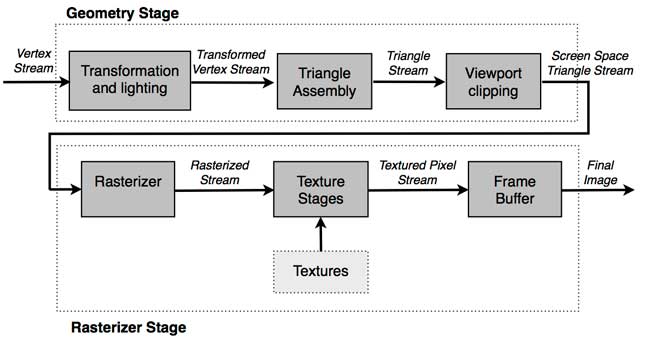
Routes the [fragments](#_Fragment) coming from the [fragment pipelines](#_Pixel_/_Fragment) to the [ROPs](#_Raster_Pipeline_/) (this is needed as the number of [fragment pipelines](#_Pixel_/_Fragment) is not equal to the number of [ROPs](#_Raster_Pipeline_/)).

### 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**.

### Rendering Pipeline Architecture

Next I describe the usual steps in a rendering pipeline.

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

#### Vertex Specification

In this early stage, we define the [vertex](#_Vertex) stream by specifying the [vertex](#_Vertex) attributes (eg. position), the storage of this stream (eg. host memory), and how to interpret the stream ([primitive](#_Primitive) type eg. triangles). [Details described later.](#_Vertex_Specification)

#### Vertex Processing

[Vertices](#_Vertex) 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](#_Hardware_Transformation_and). [Details described later.](#_Vertex_Processing_1)

#### Primitive (Triangle) Assembly

[Primitives](#_Primitive) are assembled from the [vertices](#_Vertex) coming from the previous stage. [Vertices](#_Vertex) are transformed from **clip-space to screen/window-space**. Some say there is a separate step between Vertex Processing and Primitive Assembly where some tasks are executed instead of here. [Details described later.](#_Primitive_(Triangle)_Assembly)

#### Rasterization

[Fragments](#_Fragment) are generated in this stage. **Triangle setup aka scan-line conversion**: finding out which pixels are covered by the incoming triangle, interpolating [vertex](#_Vertex) attributes across the triangle. [Details described later.](#_Rasterization)

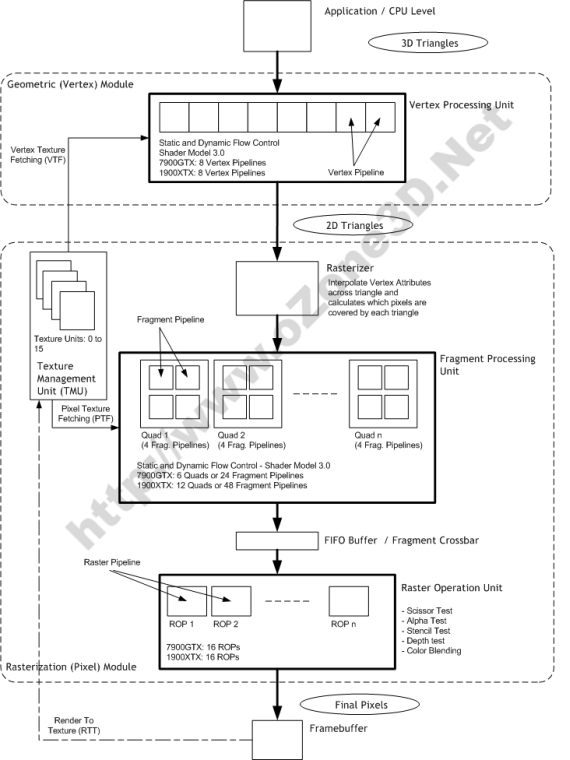
#### Fragment Processing / Shading

Color, depth and stencil values are generated from each [fragment](#_Fragment). Texturing also happens here. [Details described later.](#_Fragment_Processing)

#### Per-Sample Processing

Usual operations of this final stage are depth testing, blending, etc. [Details described later.](#_Per-Sample_Processing)

Following picture is an example of a rendering pipeline architecture implemented in HW. Basically it is the same as the previous picture:

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

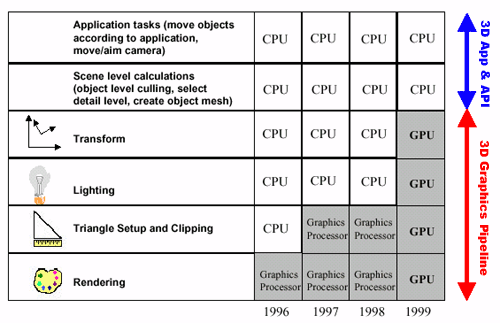
Note: as seen in the picture, [Rasterizer](#_Rasterization_1) is not equal to [Raster Operation Unit](#_Raster_Operation_Unit).

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

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

Since Geforce 256.

PRRE 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.

PRRE 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 PRRE 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.

# Engine Construct

PGE has a few subsystems: configuration (PGESysCFG), networking (PGESysNET), graphics (PGESysGFX), sound (PGESysSFX).

## PGESysCFG

## PGESysNET

## PGESysGFX

It is designed as an independent and reusable graphics engine, integrated into the game engine. PRRE means PR00F’s Reduced Rendering Engine.

### Initialization

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>

### Rendering Architecture & Pipeline

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

PRRE 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.

#### Vertex Specification

[Short description.](#_Vertex_Specification_1)

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

[Short description.](#_Vertex_Processing)

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 [PR00FPSvsPRRE Transformations.xlsx.](PR00FPSvsPRRE%20Transformations.xlsx)

##### Modeling Transformation

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

##### View Transformation

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

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.

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>

Note: using OpenGL either right- or left-handed viewing system can be used. PRRE 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 PRRE API: TODO.

#### Primitive (Triangle) Assembly

[Short description.](#_Primitive_(Triangle)_Assembly_1)

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 [PR00FPSvsPRRE Transformations.xlsx.](PR00FPSvsPRRE%20Transformations.xlsx)

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

Related PRRE 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 PRRE API: TODO.

#### Rasterization

[Short description.](#_Rasterization_1)

TODO.

Related OpenGL API: TODO.

Related PRRE API: TODO.

#### Fragment Processing

[Short description.](#_Fragment_Processing_1)

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.

Shading (Flat / Gouraud, Phong, DOT3), Texturing.

Fog.

Related OpenGL API: TODO.

Related PRRE API: TODO.

#### Per-Sample Processing

[Short description.](#_Per-Sample_Processing_1)

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 PRRE API: TODO.

##### Scissor Test

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

Related OpenGL API: TODO.

Related PRRE API: TODO.

##### Alpha Test

Related OpenGL API: TODO.

Related PRRE 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 PRRE API: TODO.

##### Stencil Test

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

Related OpenGL API: TODO.

Related PRRE 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 PRRE API: TODO.

##### Blending

Related OpenGL API: TODO.

Related PRRE 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 PRRE API: TODO.

##### Logic Operations

Unsupported by PRRE.

Related OpenGL API: TODO.

##### Write Mask

Masking off writing to particular buffers. Unsupported by PRRE.

Related OpenGL API: TODO.

## PGESysSFX

Basically it is just a wrapper to FMOD.

# 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

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

<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>