PURE

PR00F’s Ultimate Rendering Engine

1. Application/Scene

* Scene/Geometry database traversal
* Movement and animation of objects
* Movement and aiming of view camera (eye)
* Object Visibility Check including possible Occlusion Culling
* Select LOD (Level of Detail)

1. Geometry

* Vertex Specification
* Vertex Processing (receives vertices, sends transformed vertices)
  + Transforms (rotation, translation, scaling)
  + Transform from Model Space to World Space (Direct3D)
  + Transform from World Space to View Space
  + Normals also transformed during above steps to View Space but in different way
  + View Projection (from View Space to Clip Space)
* Primitive (Triangle) Assembly (receives transformed vertices, sends screen-space triangles)
  + Trivial Accept/Reject Culling (or can be done later in Screen Space)
  + Back-Face Culling (can also be done later in Screen Space)
  + Clipping
  + Lighting
  + Perspective Divide (from Clip Coordinates to Normalized Device Coordinates)
  + Transform from NDC to Screen/Window Coordinates

1. Rasterization / Triangle Setup (receives screen-space triangles, sends fragments)

* Back-face Culling (or can be done in view space before lighting)
* Slope/Delta Calculations
* Scan-Line Conversion

1. Rendering

* Fragment Processing / Shading (receives fragments, sends pixels)
  + Shading
  + Texturing
  + Fog
* Per-Sample Processing (receives pixels)
  + Alpha Translucency Tests
  + Depth Buffering
  + Antialiasing (optional)
  + Display

# PURE Rendering Pipeline Explained

## Short Description

asd

## Long Description

asd

Application/Scene Stage 2

Geometry Stage 2

Vertex Specification 2

Vertex Processing 3

Primitive Assembly 5

Rasterization / Triangle Setup Stage 6

Back-face Culling 6

Slope/Delta Calculations 6

Scan-Line Conversion 6

Rendering Stage 6

Fragment Processing / Shading 6

Per-Sample Processing 7

### Application/Scene Stage

asd

### 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 in the stream is important. Vertices 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 indices (e.g. element array). The latter has an advantage on memory consumption and performance, since same (repeating) 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.

The result of calculations done in this stage can be checked in PR00FPSvsPRRE Transformations.xlsx.

Model 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. It turns out that many 3D programs skip past world space and instead go directly to clip or view space. The OpenGL API doesn’t really have a world space.

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 (using matrix math that we’ll explore later), 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.

Modeling Transformation

Transforming the vertices from object/model-space to world-space. Simple matrix multiplication.

View Transformation

Transforming the vertices 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 from eye-space to clip-space. Simple matrix multiplication. The projection matrix defines the viewing frustum and the projection mode (perspective or orthogonal).

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.

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.

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.

#### Primitive Assembly

Primitives are assembled from the vertices coming from the previous stage. Vertices 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.

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 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:

• discarded (culled), when entirely outside of the viewing volume/frustum;

• clipped (calculating new 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 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. The “perspective divide” performs the normalization feat, by dividing all x, y, and z vertex coordinates by a special “w” value, which is a scaling factor that we’ll soon discuss in more detail. The perspective divide makes nearer objects larger, and farther objects smaller as you would expect when viewing a scene in reality.

<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.

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.

See transformation calculations in PR00FPSvsPRRE Transformations.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 / 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.

#### Back-face Culling

#### Slope/Delta Calculations

#### Scan-Line Conversion

### Rendering Stage

#### Fragment Processing / Shading

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

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

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 are discarded if the pixels covered by the fragments are covered by another window thus OpenGL doesn’t own these covered pixels.

Related OpenGL API: TODO.

Related PRRE API: TODO.

Scissor Test

Fails if the fragments 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.

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 then the Occlusion Query gets updated if there is an active query. Related: HyperZ. 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 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, 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.