PURE

Text contains quotes heavily from ExtremeTech’s 3D Pipeline Tutorial article:

<https://www.extremetech.com/computing/49076-extremetech-3d-pipeline-tutorial>

In some sense, 3D chips have become physical incarnations of the pipeline, where data flows “downstream” from stage to stage. 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. 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.

# PURE Rendering Pipeline Explained

## Short Description

asd

## Long Description

asd

Application/Scene Stage 1

Geometry Stage 1

Vertex Specification 1

Vertex Processing 2

Primitive Assembly 4

Rasterization / Triangle Setup Stage 5

Back-face Culling 5

Slope/Delta Calculations 5

Scan-Line Conversion 5

Rendering Stage 5

Fragment Processing / Shading 5

Per-Sample Processing 6

3D graphics is the art of cheating without getting caught.

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.

### 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.

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 from eye/view-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. The clipping planes are now orthogonal (perpendicular) to the axes of the space.

#### 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 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 (trivial accept), 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 by dividing all x, y, and z vertex coordinates by a special “w” value, which is a scaling factor that represents distance of the vertex from the view point, so that objects further from the view camera become smaller.

<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.

Backface 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.

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.

#### Back-face Culling

#### 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

### Rendering Stage

#### Fragment Processing / Shading

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.

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.

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 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.