1. Pipeline Overview

The fundamental problem of rendering is to take as input a series of coordinates and map that to pixels on the screen, modulated by the lighting equations. In addition to the coordinates of the primitives, the input also includes associated lighting data such as colors, textures, material properties etc.

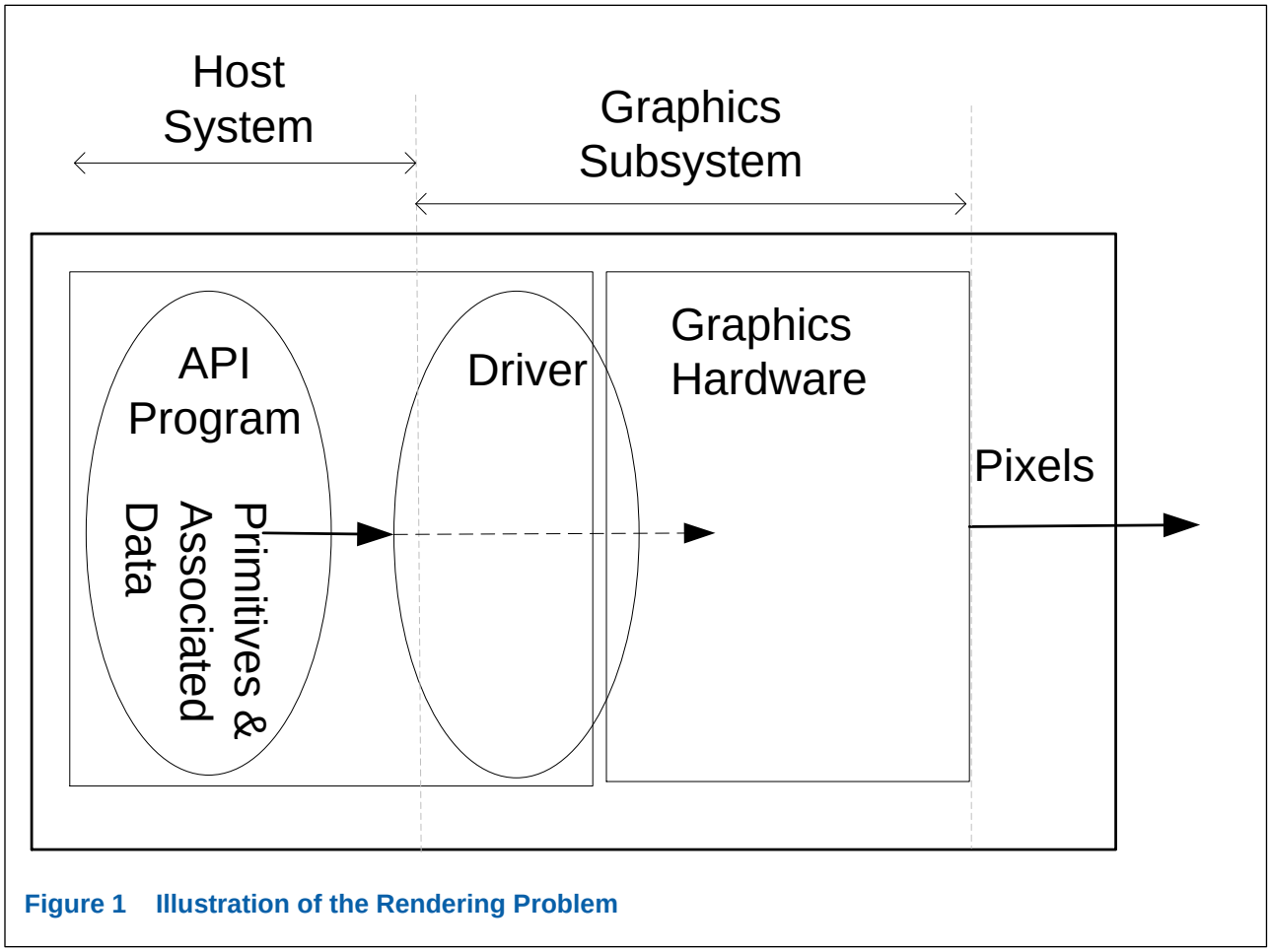

Figure 1 Illustration of the Rendering Problem

Rendering Sequence: The sequence of instructions generated by the API level program is what we refer to as the "rendering sequence". From the API level program, the rendering sequence is generated by the run-time, undergoes a series of transformation and is eventually handed off to the graphics hardware.

The exact transformations are specific to the API's and the ultimate rendering sequence generated is specific to the graphics hardware.

Graphics Stack: The system – hardware and software – that is responsible for producing the first iteration of the rendering sequence and eventually producing the pixels seen on the screen is what we call the *Graphics Stack.*

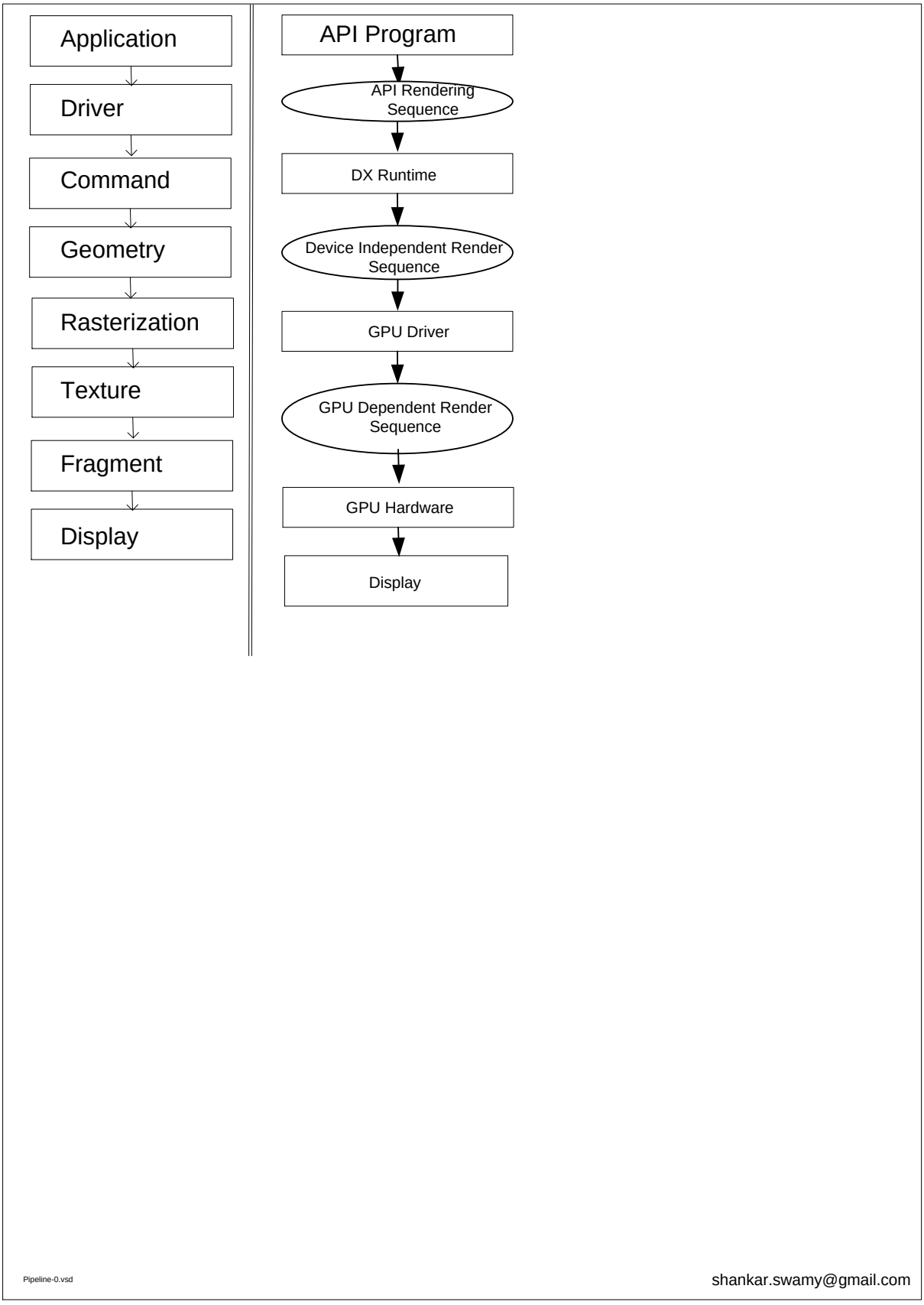

Figure 2 Some graphics stack representations

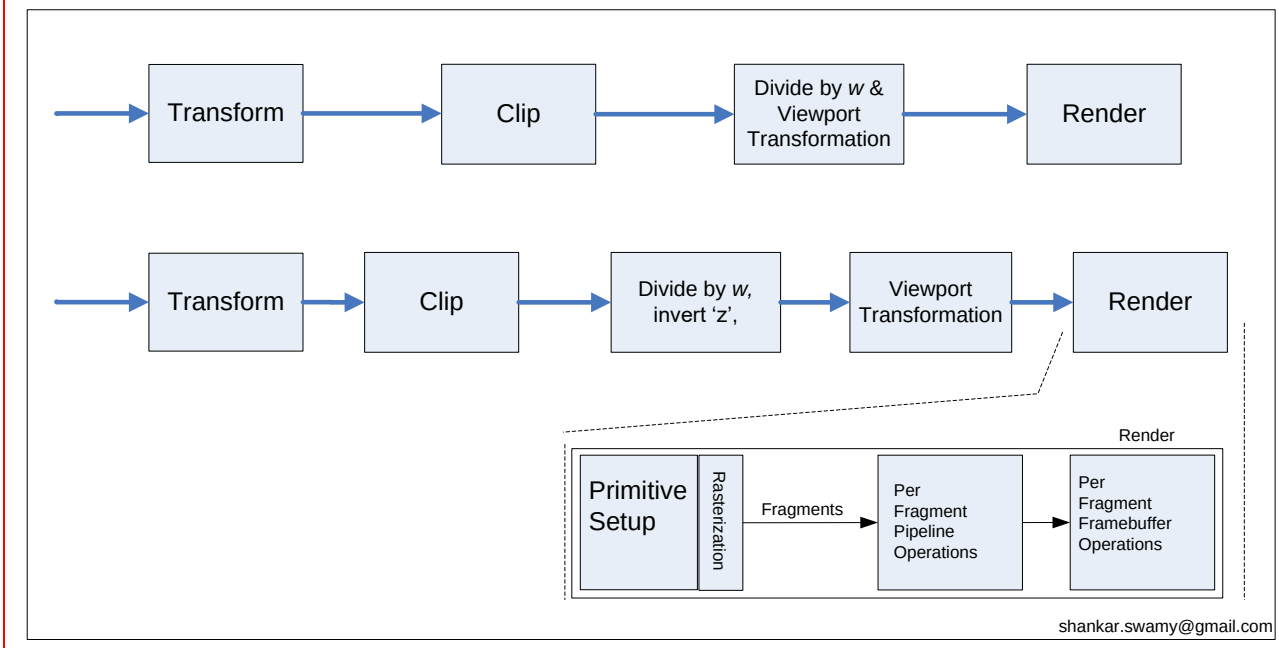

Figure 3 Sequence of transformations & operations for 3D rendering of primitives

1.1 Classic Rendering Pipeline

"Classic" in this context means "OpenGL" because at first there was only OpenGL *fixed function* rendering pipeline (i.e. no programmable shaders).

Figure 4 shows a block diagram of an early version of OpenGL. A lot of the fundamental features of this continue to be true today:

1. Operations on vertices, pixels, fragments and textures are orthogonal.
2. Symmetry in imaging and geometry paths.
3. Circular paths within the pipeline.

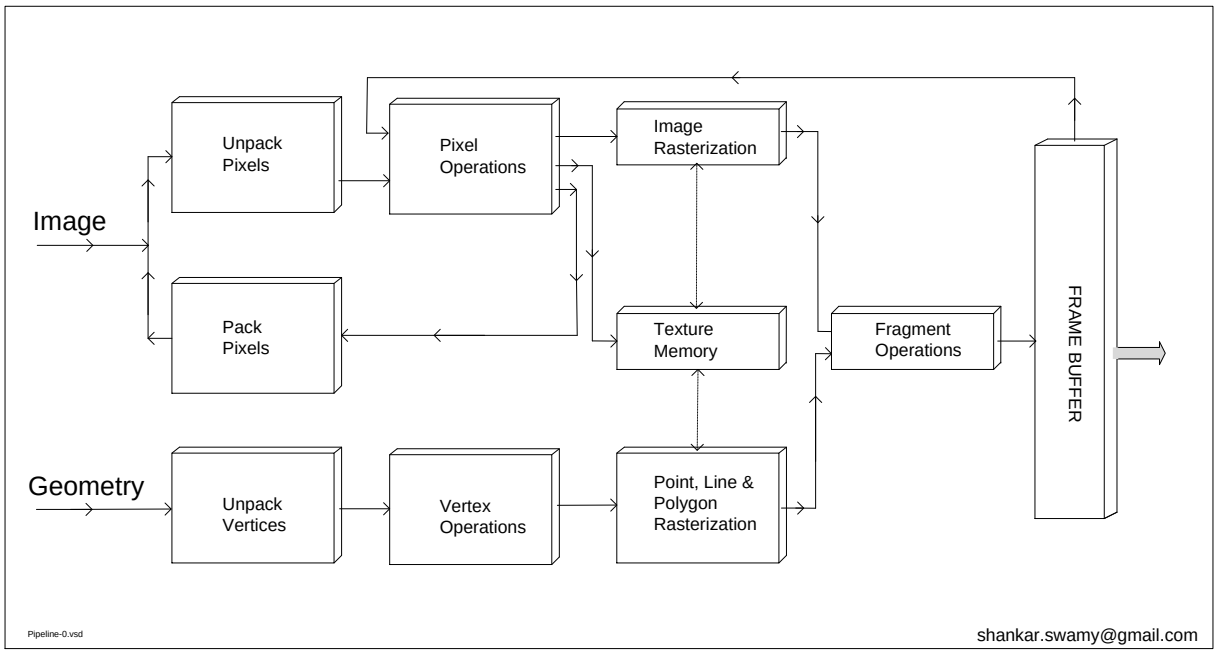

Figure 4 A block diagram of an early version of OpenGL

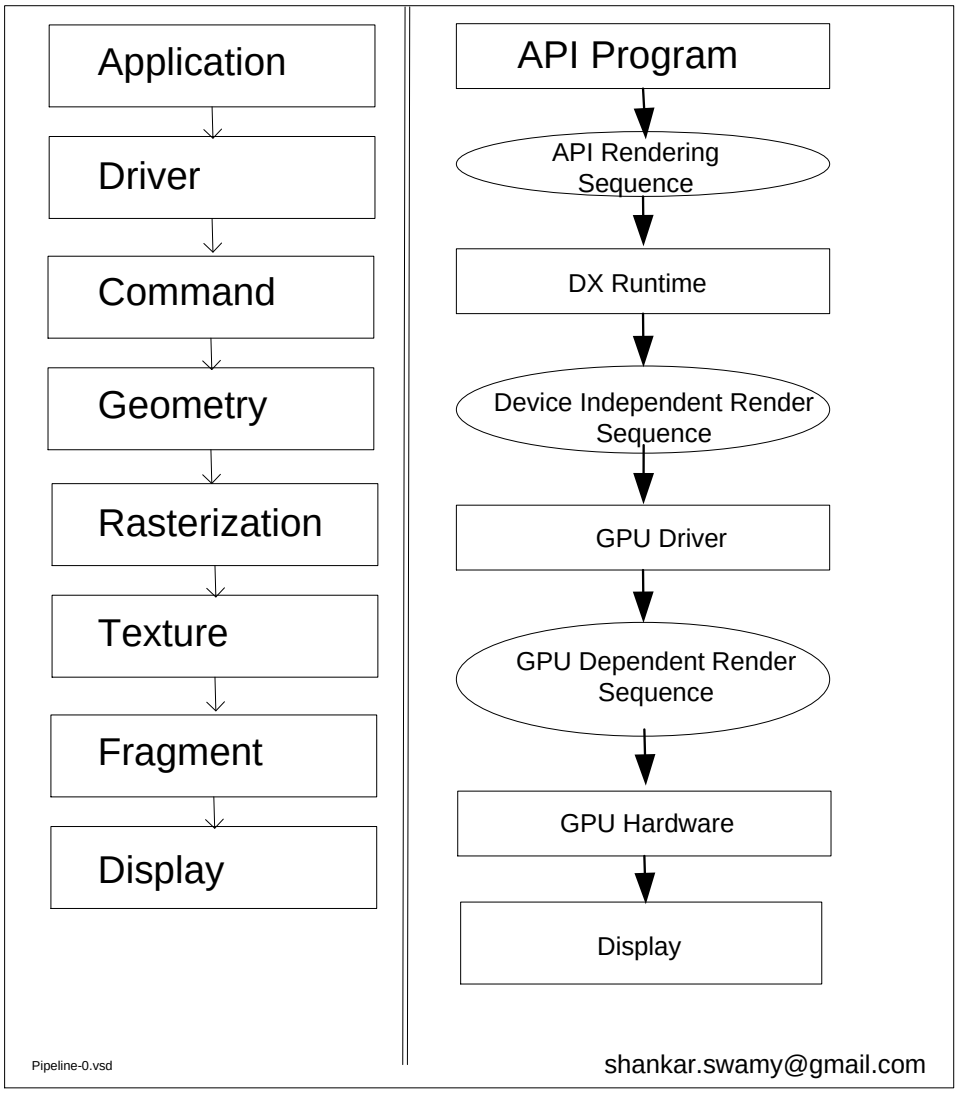

Figure 5 A more advanced, early OpenGL (fixed function) pipeline

OpenGL specification did not (does not) provide a model implementation; rather specifies what results of rendering should be and not how they should be computed. So any implementation would *a possible compliant* implementation.

Notice two distinct stages of implementation in the following:

1. the geometry processing pipeline: up to (but excluding) the Triangle Setup,
2. the rasterization pipeline: from Triangle Setup till the end of the pipeline.

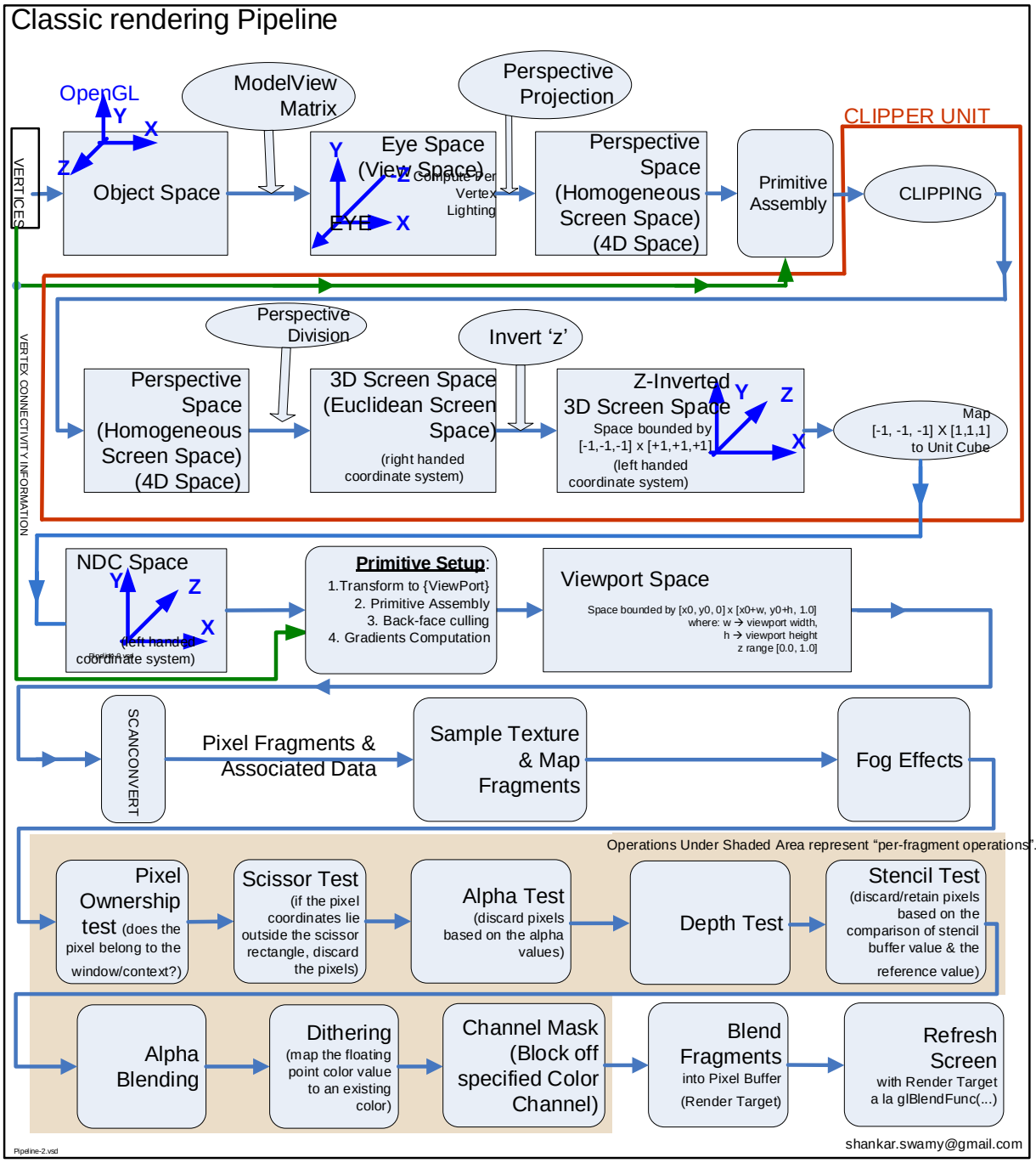

Figure 6 A generic, non-programmable pipeline implementing an early OpenGL compliant rendering

1.1.1 Notes on Figure 6

1. The perspective transformation is split into two parts: 1. the perspective projection and2. the perspective division. The perspective projection transforms the vertices from {Camera} to {Perspective} or the homogeneous screen space. Then the perspective division transforms it to Euclidian {Screen}.

   The {Object} → {Homogeneous Screen} is an affine transformation. But the {Homogeneous Screen} → {Euclidian} back is <u>not</u> an affine transformation. This is the root of the "texture interpoliatoin problem". To convert the triangle into pixel coordinates, we *project* the coordinates fo the triangle to the {Screen}. It is a projective transfomration. While in {Homogeneous Screen} coordinates, we need to interpolate the texture coordinates between the two vertices. We cannot start with the texture coordinates in {Texture} of the two vertices and linearly interpolate to get the texel for each pisel between the two vertices. That is not going to work as {Homogeneous Screen} → {Euclidean i.e. Texture} is not affine. The texture needs to be interpolated using "Rational Linear Interpolation".

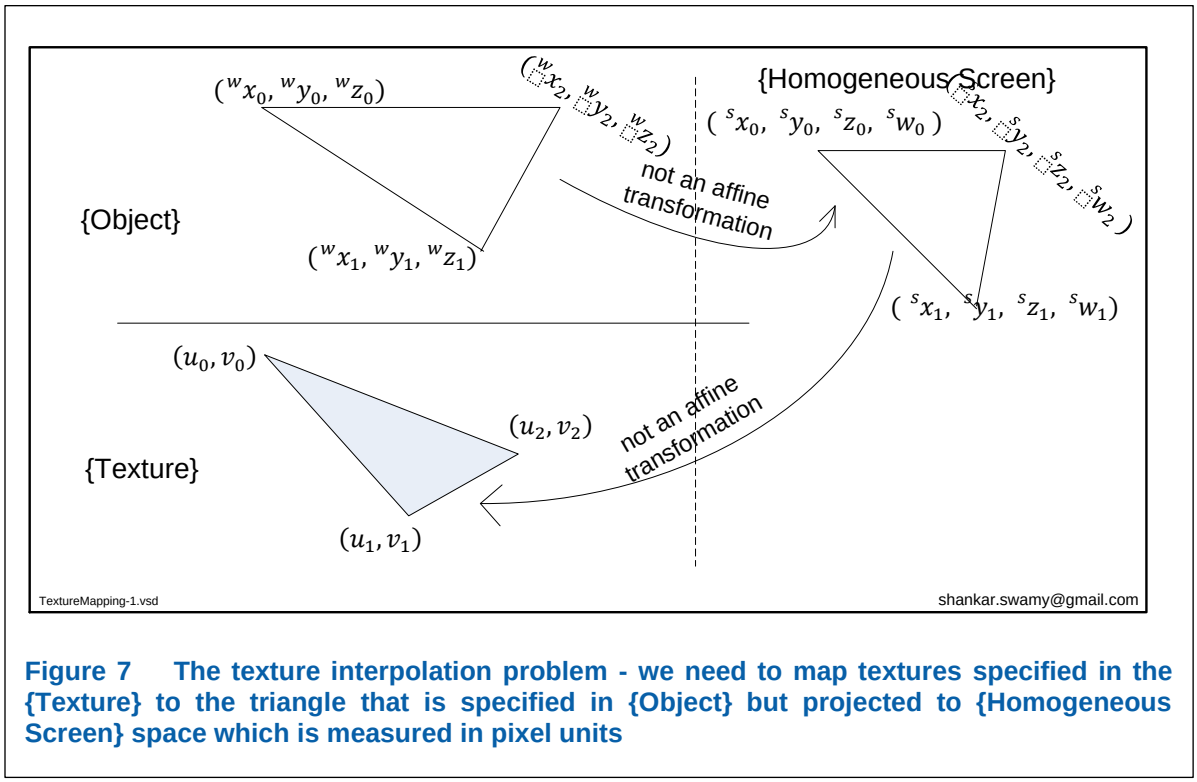

Figure 7 The texture interpolation problem - we need to map textures specified in the {Texture} to the triangle that is specified in {Object} but projected to {Homogeneous Screen} space which is measured in pixel units

In Figure 7, we need to assign texls to each pixel based on $({}^{s}x, {}^{s}y)$ of the pixel. We know $({}^{s}x_0, {}^{s}y_0) \rightarrow (u_0, v_0)$ and $({}^{s}x_1, {}^{s}y_1) \rightarrow (u_1, v_1)$. Ignoring the bilinear interpolation for the moment, if we have to linearly interpolate the texture coordiantes (u_0, v_0) and (u_1, v_1) to determine the texel associated with certain $({}^{s}x, {}^{s}y)$, we cannot map linearly $({}^{s}x, {}^{s}y) \rightarrow (\frac{(u_1 - u\_0)}{({}^{s}x_1 - {}^{s}x_0)}({}^{s}x - {}^{s}x_0), \frac{(v_1 - v\_0)}{({}^{s}y_1 - {}^{s}y_0)}({}^{s}y - {}^{s}y_0))$. This will not work. The pixel coordinates from the {Homogeneous Screen} do not map linearly to {Texture} space, as the transformation to {Homogeneous Screen} from {Object} is a projective transformation. However we do know that $({}^{s}x/w, {}^{s}y/w)$ map linearly to (u_0, v_0) and (u_1, v_1). So we need to interpolate as:

$$\left(\frac{^{s}x}{w},\ \frac{^{s}y}{w}\right) \rightarrow \left(\frac{(u_1 - u\_0)}{\left(\frac{^{s}x_1}{w_1} - \frac{^{s}x_0}{w_0}\right)}\left(\frac{^{s}x}{w} - \frac{^{s}x_0}{w_0}\right),\quad \frac{(v_1 - v_0)}{\left(\frac{^{s}y_1}{w_1} - \frac{^{s}y_0}{w_0}\right)}\left(\frac{^{s}y}{w} - \frac{^{s}y_0}{w_0}\right) \right)$$

We will run into a similar issue when a primitive is clipped. The texture coordinates are specified for the entire primtive, and if part of it is clipped, the texture coordinates need to be interpolated to assign new texture coordintates to the clipped vertex. This is illustrated in Figure 8.

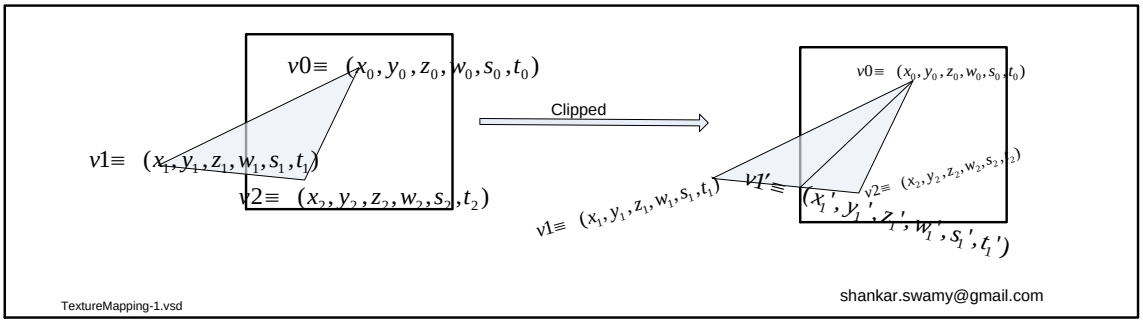

Figure 8 When a primitive is clipped, we need to interpolate the texture coordinates using "Rational Interpolation" and determine the new texture coordinates for the new vertex introduced. Though the above situation will result in two new vertices in place of one that is outside, we have marked up only one of them in the Figure.

2. The OpenGL perspective projection transformation combines a few more operations beyond the perspective transformation and those operations are shown in shaded color in Figure 11.
3.

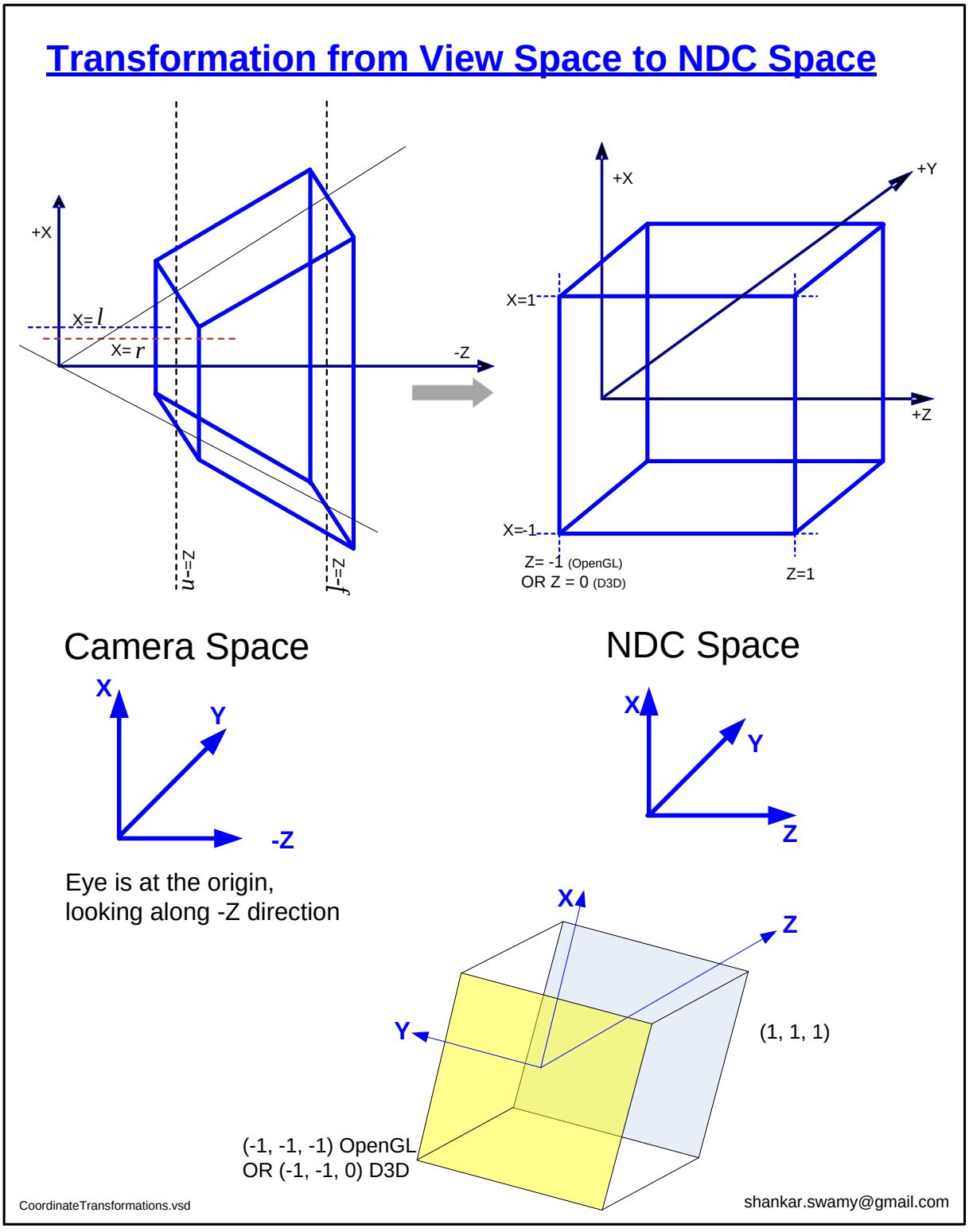

Figure 9 {View} and {NDC} Systems – the redundant flipping of z-axis between {View} and {NDC} is for historic reasons – has no real value it and in implementations does not add extra overhead.

Figure 9 {View} and {NDC} Systems shows the various coordinate spaces that the geometric primitives are transformed through as they traverse through the pipeline. In this document we indicate the coordinate spaces either explicitly as in "Perspective Space" or by just using the name of the coordinate space surrounded by curly braces as in "{Perspective}". To refer to a coordinate in a particular space we use the super-prefix notation such as $({}^{vp}x,\ {}^{vp}y,\ {}^{vp}z)$ for coordinates (x, y, z)in the {Viewport}.

The coordinate spaces which are more frequently used in this document along with the notations used to represent them are given in Table 1

Table 1 Coordinate systems frequently used document

Coordinate Space with equivalent names	Representation for coordinates	Immediate previous coordinate space in the pipeline	Transformation path from the immediate previous coordinate space
World space or {World}	(x, y, z)	none	none
Texture Space or {Texture}	(u, v) or (u, v, s, t)	none	none
Homogeneous Texture space or {HTexture}	$({}^{h}u,\ {}^{h}v)$ or $({}^{h}u,\ {}^{h}v,\ {}^{h}s,\ {}^{h}t)$	{Texture}	(Projective transformation) Not needed.
View space or {View}, also {Camera}, {Eye},	$({}^{v}x,\ {}^{v}y,\ {}^{v}z)$ for vertex; $({}^{v}u,\ {}^{v}v)$ for texture	{Object}	Modeling and viewing matrices
Perspective space or {Perspective}, also {Clip}, {Homogeneous Screen}, {Projective}	$({}^{p}x,\ {}^{p}y,\ {}^{p}y)$ for vertex; $({}^{p}u,\ {}^{p}v)$ for texture	{View}	Perspective projection matrix
3D Screen space {3D Screen}	$({}^{s}x,\ {}^{s}y,\ {}^{s}z)$	{Perspective}	Perspective division *i.e.* divide by w
NDC space or {NDC}	$({}^{ndc}x,\ {}^{ndc}y,\ {}^{ndc}z)$	{3D Screen}	Invert z- coordinate and scale the view volume to $[-1,-1,-1] \times [1,1,1]$
Viewport space or {Viewport}	$({}^{vp}x,\ {}^{vp}y,\ {}^{vp}z)$	{NDC}	Map $({}^{vp}x,\ {}^{vp}y)$ to $[{}^{vp}x_{org},\ {}^{vp}y_{org}] \times [{}^{vp}x_{org} + VP_{width},\ {}^{vp}y_{org} + VP_{height}]$, with the z-value that is closest to origin of all possible pixels along the z-axis

For a coordinate system, using a suitable name that is indicative of its nature or function makes it easier to analyze the given problem. And for some of the systems, the suitable name depends on the context. Thus we end up with multiple names for some of these systems and we use different names for the same system, depending on the context. Some transformations such as inverting the Z-axis to go from {3D Screen} to {NDC} are there because of legacy reasons. They have not disappeared because they do not add additional

computational overhead to the pipeline as such transformations are integrated into a matrix multiplication that anyway has to be carried out.

In the notation outlined above the space that a coordinate belongs is unambiguously identified with the prefixed superfix to the coordinate.

1.2 Relationships and tranformations between different coordinate systems

We also deal with the coordinate space associated with the textures, in addition to the ones mentioned earlier.

Texture coordinates, as they are specified are fractional – that is, they are within the range $[0, 1.0]$. So within two coordinate systems related by affine transformations, the texture coordinates do not change – that is, they are identify transformations.

In the graphics pipeline the primitives start from either the object space or the view space. They undergo a projective transformation. The projected vertices are transformed to the screen space and rasterized in the screen space.

1. It is not simple due to projective transformation which is different from a rigid transformation such as modeling and viewing transformations.

2. We have two groups of coordinate spaces separated by the projective transformation. Since the projective transformation is not an affine transformation, it is not possible to move from a coordinate system in one group to any one in the other by exclusively using affine transformation. But within a group it is possible to transform the vertex element to different systems by using affine transformations exclusively. Two coordinate systems between which we can transition with exclusively using affine transformations are said to be *affine with each other* (2).
3. Explain the diagram below.
4. Refer to the Appendix on Olano paper.

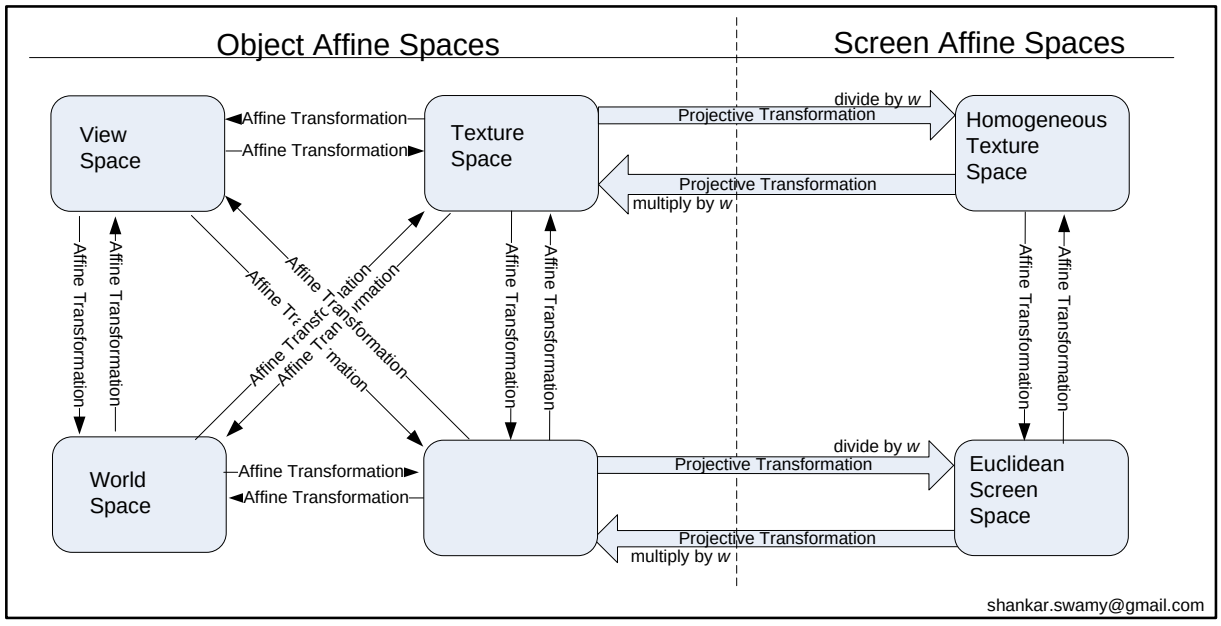

Figure 10 Six different vector spaces used in 3D pipeline

The vector spaces mentioned in Figure are explained in the following table.

Table 2 Transformation modes for transforming between different vector spaces used in the 3D pipeline

Source vector space	Destination vector space	Transformation mode
Object Space	Texture Space	Affine: Linear/Bilinear
Object Space	World Space	Affine: Model Matrix (rotation, translation, scale are all affine matrices)
Object Space	Homogeneous Screen Space	Affine: Multiply by the *View Matrix* that often includes the perspective projection
Texture Space	Homogeneous Screen Space	Affine: This is just an identity transformation which is an affine matrix*
Texture Space	Homogeneous Texture Space	Projective: Need to *project* to a space that is affine with screen space: needs a divide by $\tilde{w}$.**
Homogenous Screen Space	Screen Space	Projective: the *perspective divide*: divide by ^{p}w**.
Homogeneous Texture Space	Texture Space	Projective: Inverse of the *perspective divide*: this is a multiply by *w*
Screen Space	Homogeneous Space	Projective: Inverse of the *perspective divide*: this is a perspective multiply by *w*

*This is an identity transformation - always: to go from a Euclidean to homogeneous coordinates is an identity transformation. It is when we switch spaces – say from the Euclidean space to a projective space that we have a more complex operation which includes a perspective divide. Homogeneous coordinates are also very useful in Euclidean space also: they make all affine operations to be represented as multiplication by [4x4] matrices.

**For efficiency, it should be “multiply by *1/w*” rather than a divide by *w*.

1.3 Overviews of the rendering Pipeline

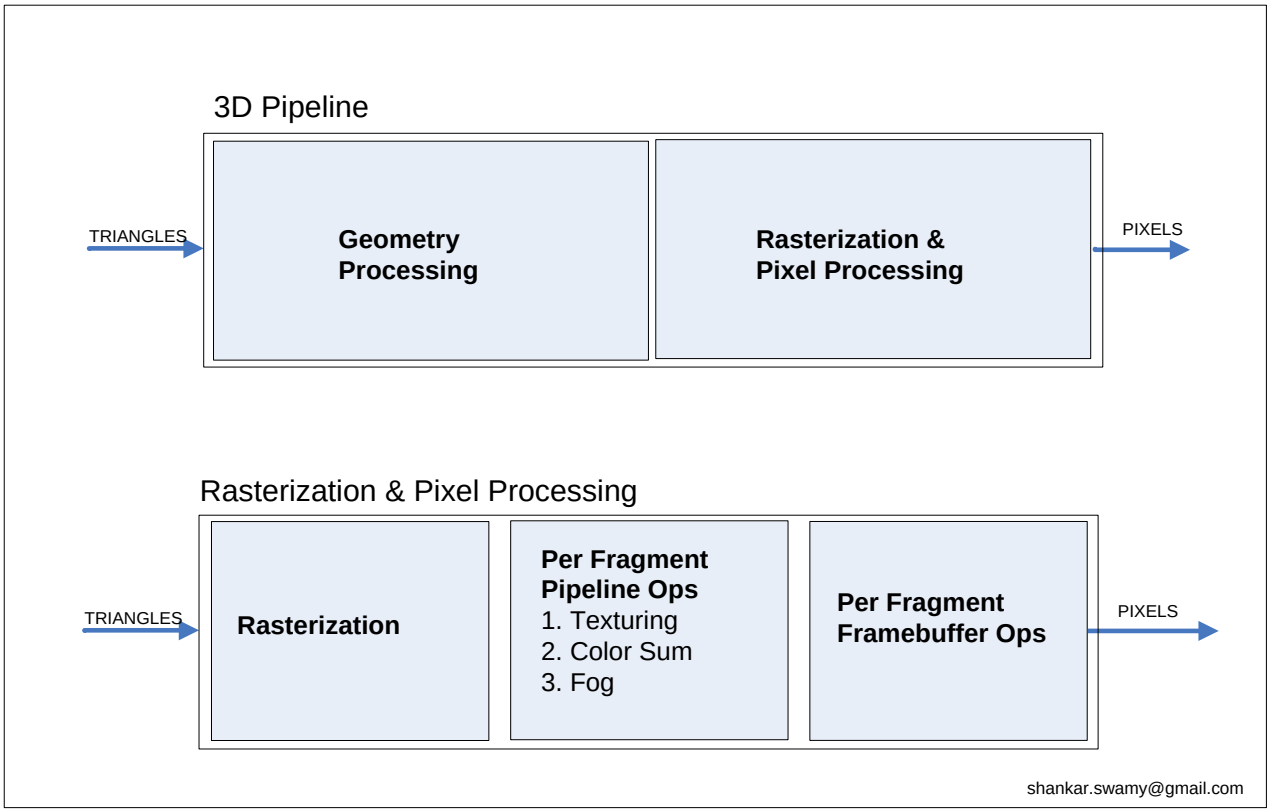

Figure 11 Interactive Rendering System: Overview

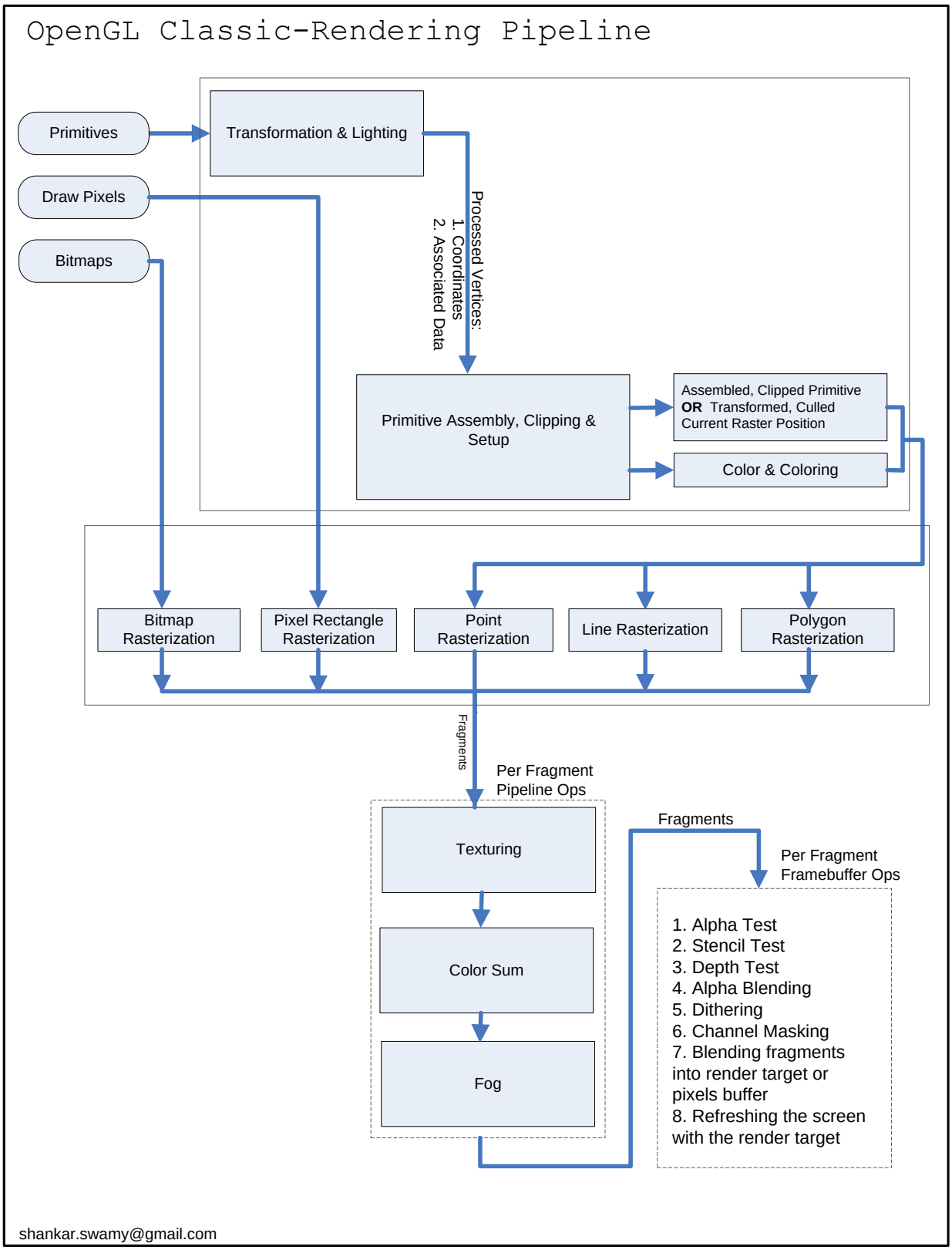

Figure 12 Classic OpenGL rendering pipeline

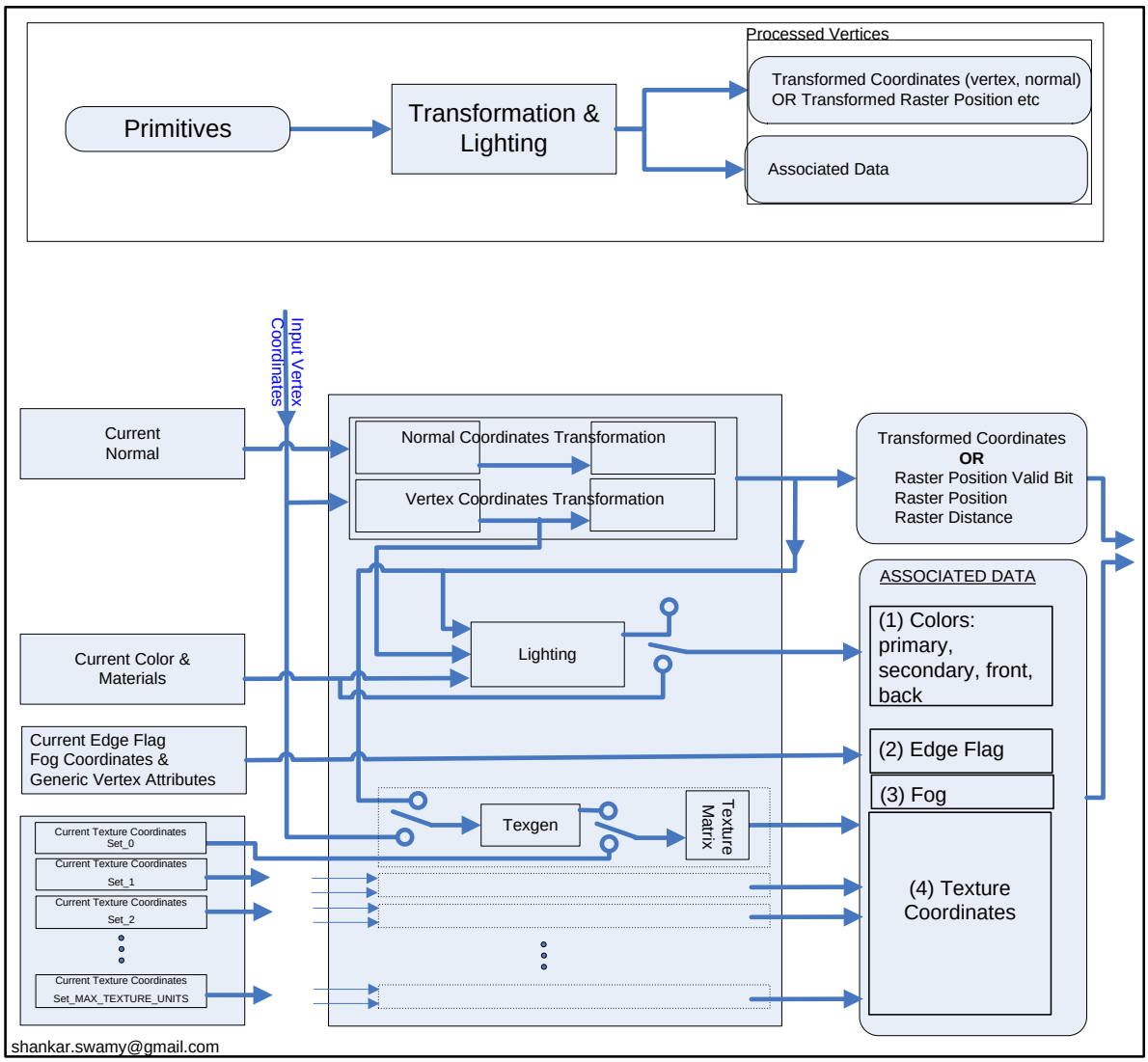

Figure 14 Classic OpenGL rendering pipeline showing the data flow paths

Notice the correlation between the three of the four components of the *Associated Data* above and the three processing stages after the rasterization: Texturing, Color Sum and Fog. The fourth one: *Edge Flag* data is used in the rasterization process.

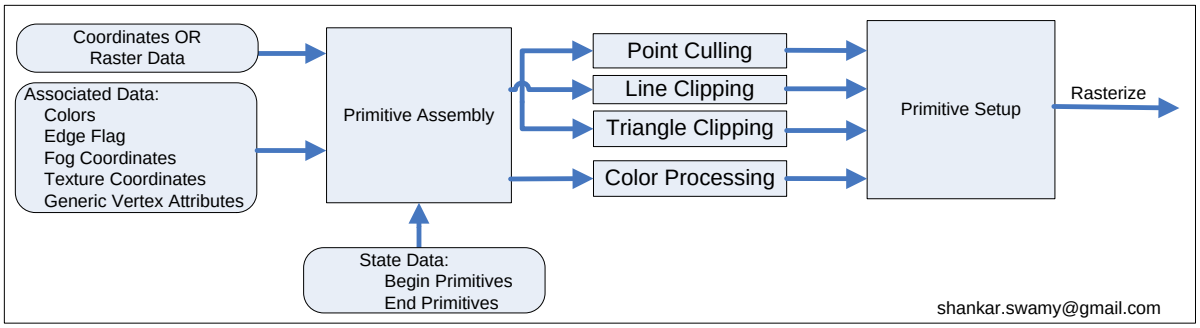

Figure 15 Classic OpenGL rendering pipeline showing pre-rasterization setup of data

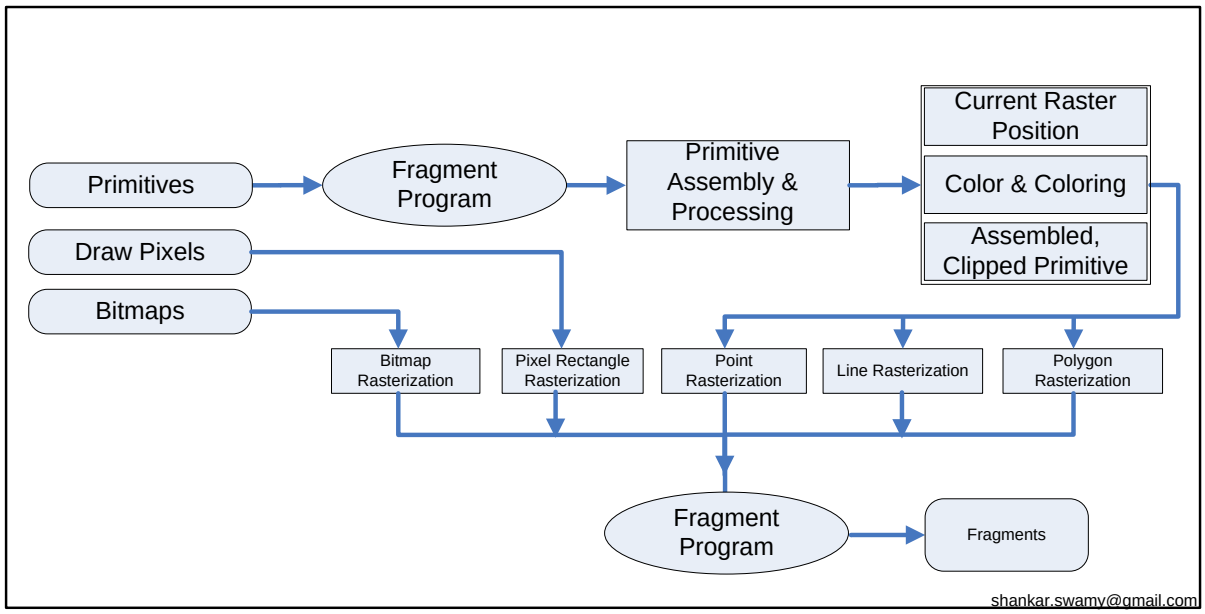

Figure 16 OpenGL programmable rendering pipeline showing the data flow paths

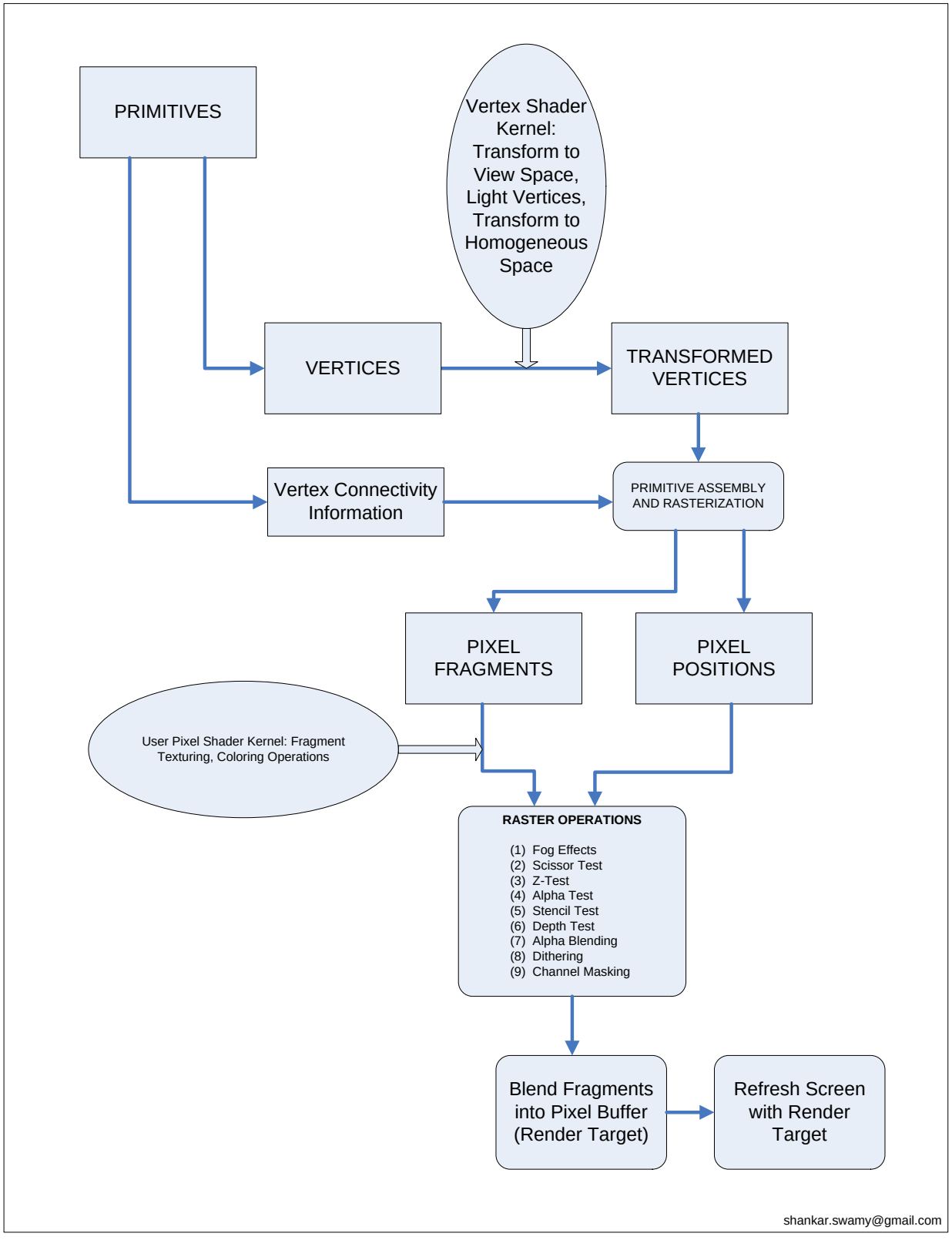

Figure 17 IV Generation programmable pipeline sequence of operations

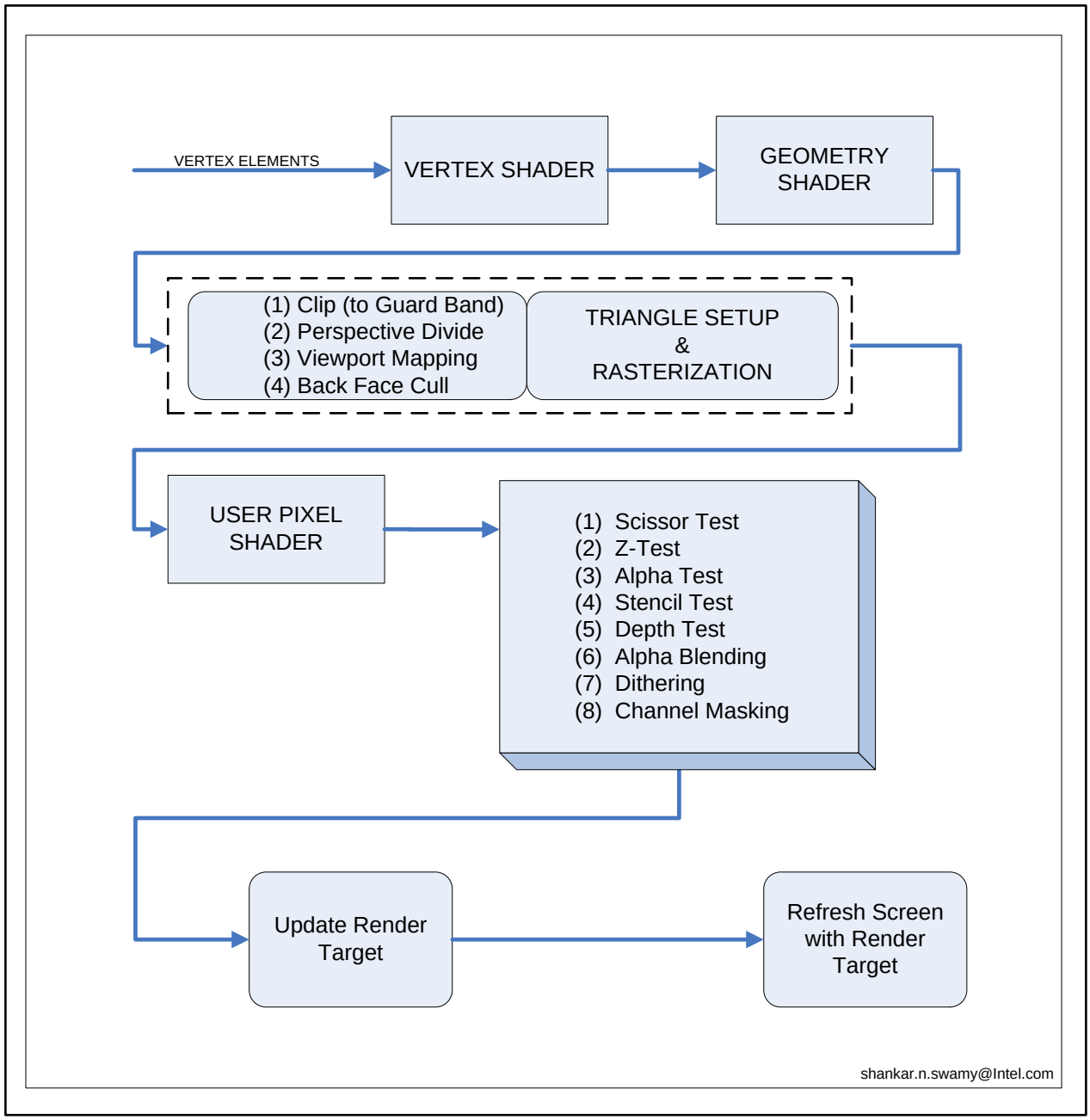

Figure 18 An overview of DX10 pipeline

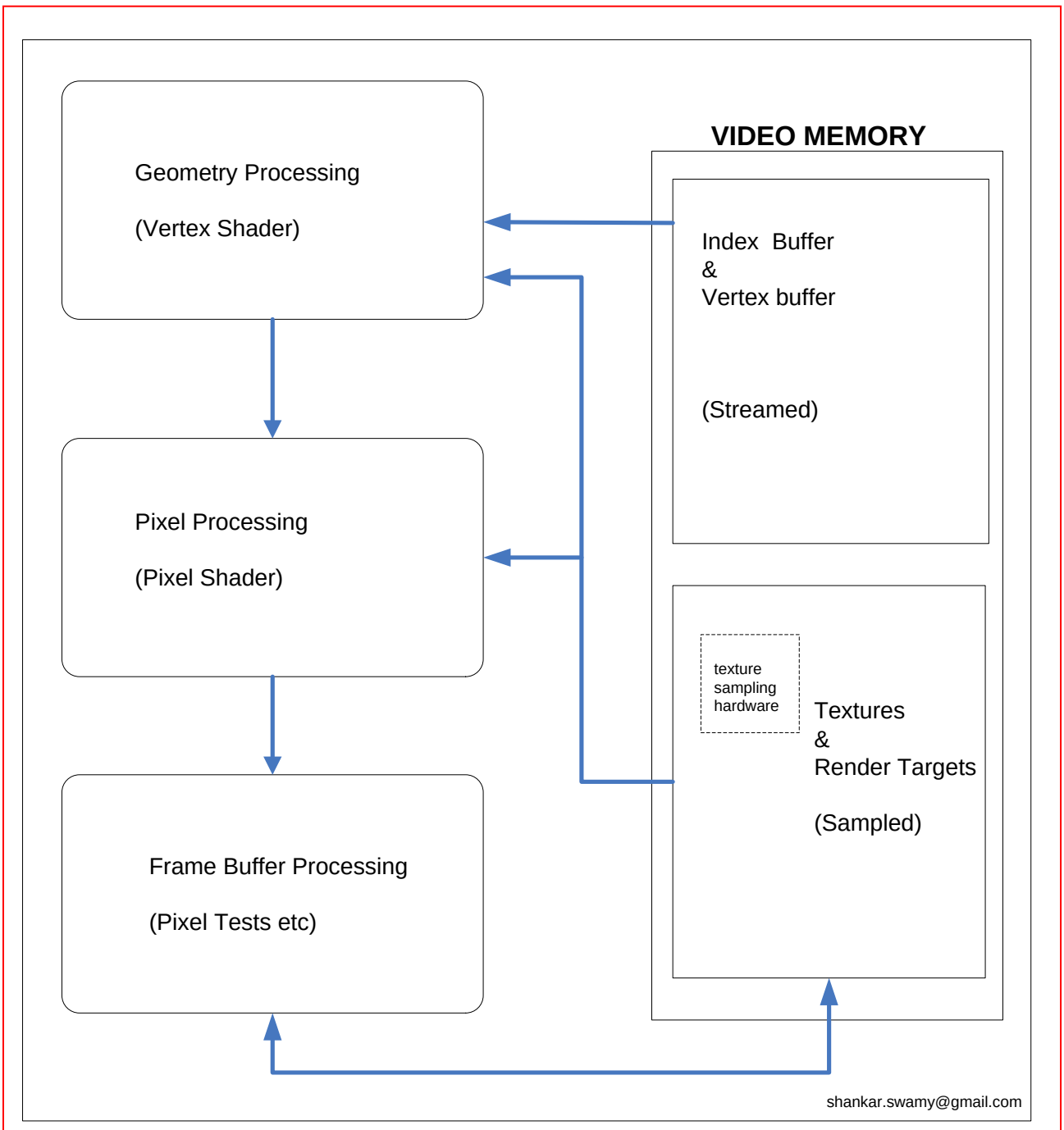

Figure 19 DX9 programmable pipeline - API view

Notice that the Vertex and Index buffers on one hand, and the Textures and Render Targets on the other hand are separate. (Former is streamed and the latter is sampled.)

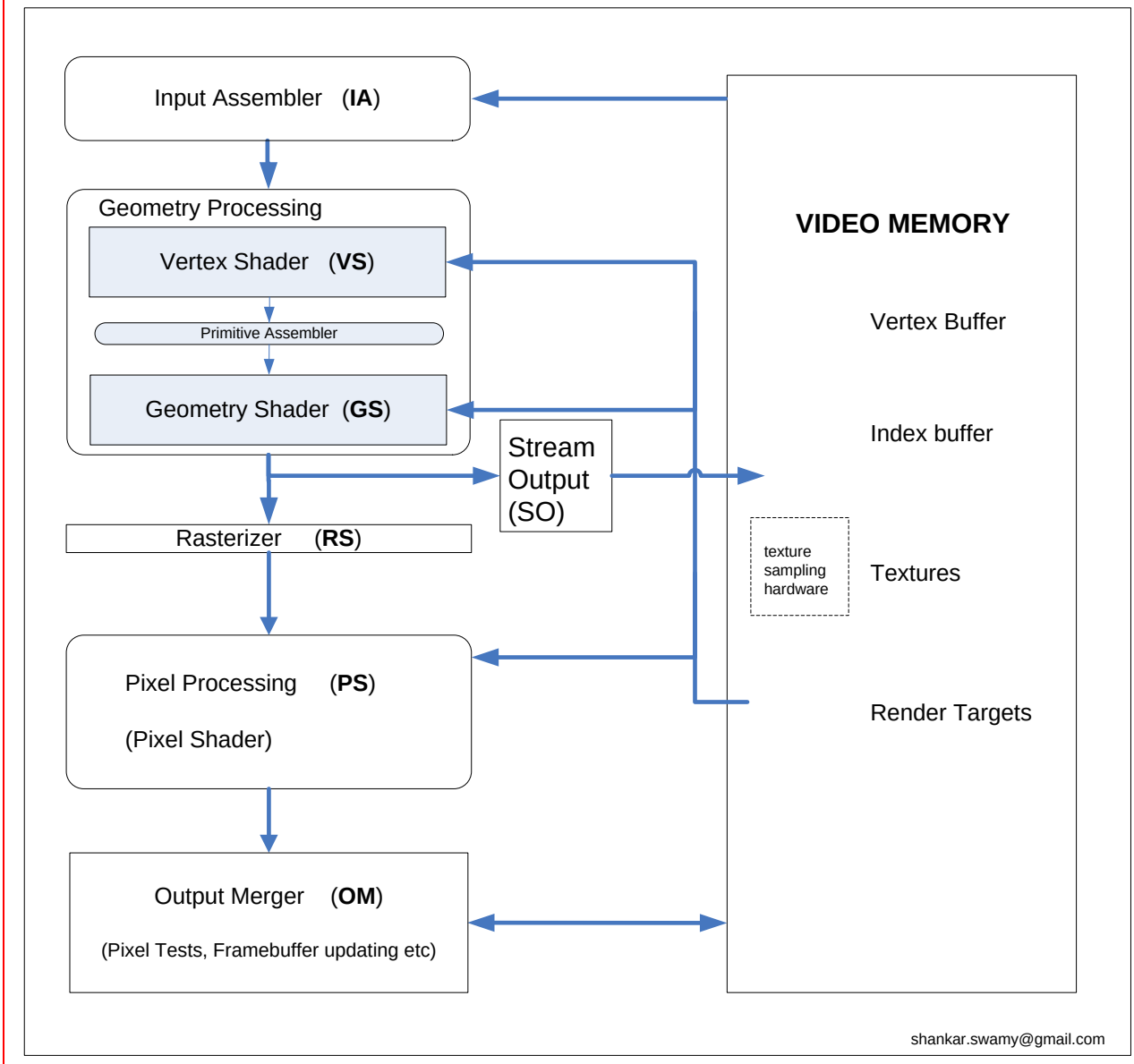

Figure 20 DX10 pipeline - API view

1.4 Rasterization

In DX10 model, there is just video memory. Programmer has the flexibility to reinterpret the video memory as needed. However, the distinctive streaming and sampling of buffers is still retained.

The Stream Output Stage allows for the primitives from the Geometry Shader to be streamed back into the pipeline through the video memory.

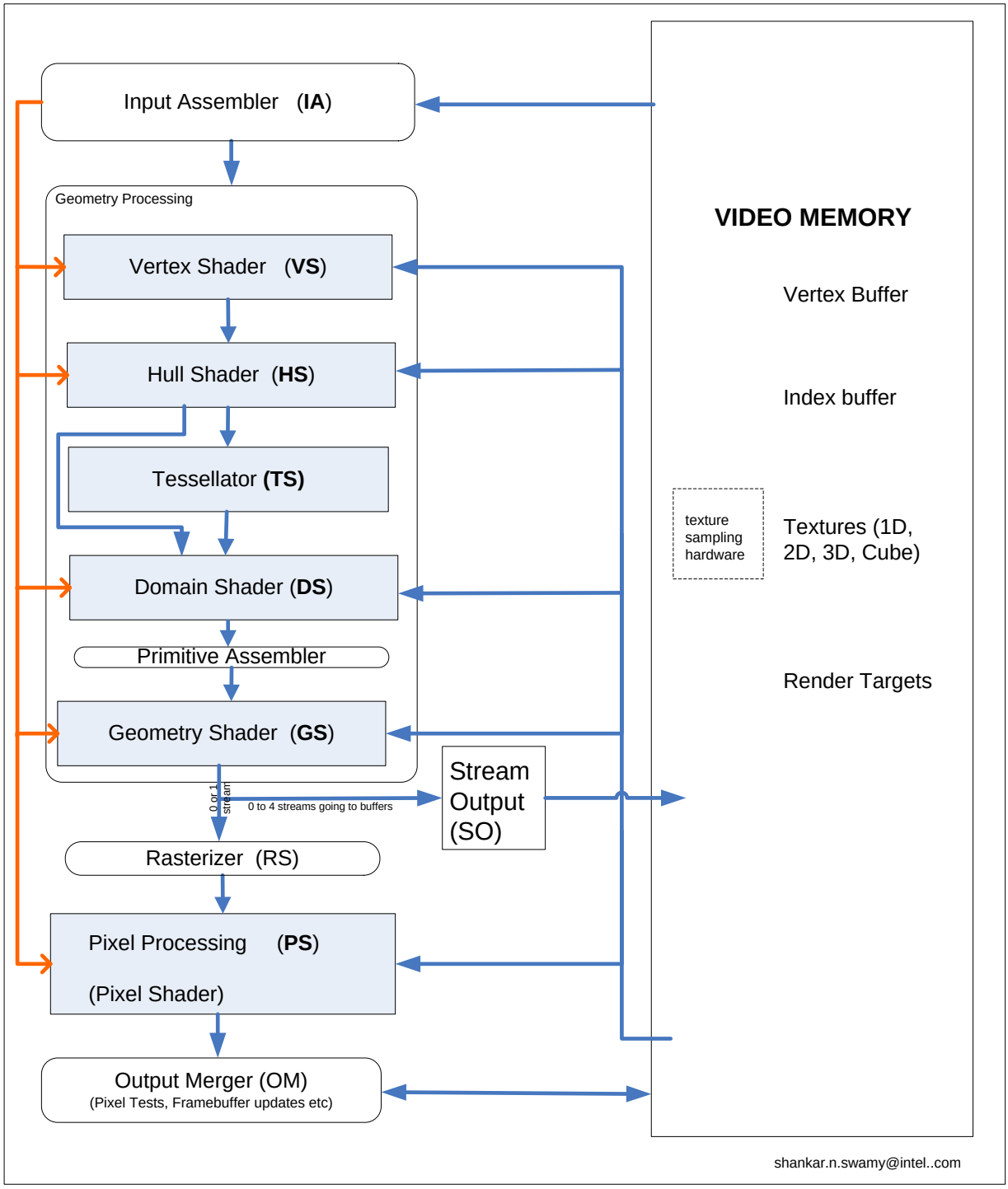

Figure 21 DX11 pipeline - API view

1.5 Geometry Processing Pipeline

This stage the primitive vertices are transformed from the object space to the screen space. This involves two separate matrix multiplications. The model-view matrix takes the vertices to the view space where they are lit. This is a direct coordinate transformation. The coordinate transformations are discussed in Appendix **Error! Reference source not found.**.

The projection matrix does the perspective projection (focus on the perspective projection and ignore the orthographic projection for now), and delivers the vertices in the screen space.

The OpenGL projection matrix goes somewhat beyond doing mere perspective projection. The OpenGL projection matrix is discussed in greater detail in Appendix **Error! Reference source not found.**.

This stage involves three steps:

a. Per-triangle step: This is called the "triangle setup" and involves calculating a series of parameters such as the slopes the triangle sides etc which are used in the next two steps.
b. Per-span step: This is called the "edge walk". During this step, the triangle is decomposed into several horizontal spans.
c. Per-pixel step: In this step, each span is rasterized into a series of fragments by interpolating the colors and texture coordinates along the spans.

We note that the rasterization stage is the point in the pipeline where the processing shifts from the object space to image space.

1.6 Per Fragment Pipeline Operations

There are three main operations here:

1. Texturing,
2. Color Sum
3. Fog.

Texturing: Involves sampling and filtering the textures.

Color sum: When the fragment enters this stage, it will have two sets of colors: $C_{pri} = (R_p, G_p, B_p, A_p)$ and $C\_\sec = (R_s, G_s, B_s, A_s)$. In the color sum stage, the two sets are combined to produce a single set of colors C, as per the relation:$C = (R_p + R_s, G_p + G_s, B_p + G_s, A_p)$. The A_s component is not used. Following this summing, the components are normalized to $[0, 1.0]$ range.

Color sum is a post-texturization process and the input primary color might have been modified by the texture mapping, if the texturing is enabled in the pipeline.

The algorithm for computing the colors coming in to the rasterization phase is discussed in the section xxxxx

Fog:

Per Fragment Framebuffer Operations

1.7 Guardband Clipping

In geometry processing, there are two distinct types of operations:

1. Vertex Operations: apply to each vertex regardless of which primitive it is part of. Examples: normalizing the length of a vector, transforming the vertex/texture coordinates between different spaces, perspective division (or inverse perspective division) etc.
2. Primitive Operations: Needs to operate on the entire primitive. There are two of these: clipping primitives against the frustum and back-face culling of primitives. Neither of these can be accomplished without full information on primitives. (Back face culling is done by computing the sign of the area of the primitive in the screen space. In any case, vertices do not have a "face". Only a primitive does.)

1.8 Bibliography

1. *Clipping using homogeneous coordinates.* **Blinn, James F. and Newell, Martin E.** 1978. ACM SIGGRAPH Computer Graphics. Vol. 12, pp. 245-251.