An Efficient and Dynamically Configurable System for the 3D Pipeline

Walt Donovan
Salim Abi-Ezzi
Version 1.01 11/20/91

1.0 Introduction

This paper discusses a system for the efficient implementation of the PHIGS+/XGL 3D graphics pipeline. The system is based on a formal mathematical analysis and is independent of specific hardware architectures, which makes it portable, both in design and code, from one generation of 3D graphics products to another. It is this lasting value and stability that are the key goals of this work.

We give a coherent treatment of the different computations that graphical data are subjected to in the 3D pipeline (before reaching the rendering stage.) These computations include transformation, face determination, model and view clip testing and clipping, lighting computations, and the dynamic tessellation of curved lines and surfaces. We choose the right ordering of these computations to maximize efficiency.

We obtain efficiency by exercising two ideas: first, we identify and formally define "well behaved" graphical primitives and attributes (which we claim characterizes what is commonly used in practice) and then optimize for this class of data. Second, we optimize further by dynamically configuring the 3D pipeline based on the specific type of primitive and the specific values of attributes. For example, a primitive could be a point, line, or surface, and a transformation attribute could be rigid, affine, or projective. An efficient pipeline for wireframe rendering is different from that for lit and shaded surfaces, and an efficient pipeline for flat surfaces is different from that for curved surfaces.

Another goal of the design presented here is to keep the implementation as simple as possible. For example, the different supported pipelines share code, and we suggest that the system not include complex code for the correct processing of some rare cases of ill-behaved data.

We first describe a conceptual 3D pipeline which is consistent with that of XGL and PHIGS+, we then discuss the characteristics of what we consider to be well behaved graphical data and argue that these characteristics apply to commonly used data in practice, we present a system of pipelines for the accurate and efficient implementation of the conceptual pipeline, and finally we discuss the case of ill-behaved data.

1.1 Acknowledgments

Thanks to Tim Van Hook for suggesting the ancestor of pipeline modules, branch-free clip testing, and calculating facet facing in device coordinates. Also, discussions with him helped clarify an earlier version of this work. Thanks to Leon Shirman for a detailed review and treatment of the NURB pipelines. Thanks to Kevin Wu for a detailed review of this paper.

1.2 Notation

This paper uses different typefaces to distinguish coordinate systems and matrices. Coordinate systems use **THIS**. A **3** or **4** is suffixed if it is important to distinguish the dimensionality of the coordinates; otherwise, it is generally homogeneous. By definition, **MC3** (for example) is obtained from **MC4** by dividing by the w component. (to simplify discussion, if w is zero, we consider that the point is on the sphere at infinity.)

Transformation matrices use THIS. Since we represent all matrices with a single letter, that would represent the product of four matrices T, H, I, and S. The inverse is represented by T^{-1}, and the transpose by H^T. Points p_1, p_2, ... are represented by a four tuple column vector $(x\ y\ z\ w)^T$. Vectors v_1, v_2, ... are represented by a four tuple row vector (x y z 0). The dot product is calculated by multiplying a row vector with a column vector: v.p. Thus, the dot product is equal to the matrix product, and we can associate and factor expressions easily.

2.0 The Six 3D Geometry Types In PHIGS/XGL

The six 3D geometry types we concern ourselves with in this paper are

TABLE 1. The Six 3D Geometry Types

GEOMETRY TYPE	ABBREVIATION
Points	P
Points with variable device extents	PE
Lines	L
Curves	C
Facets	F
Curved surfaces	S

These geometry types are defined by a collection of vertices, interpreted as appropriate to that geometry. We assume that it is possible to transform the geometry simply by transforming these vertices.

Points are defined by a single 3D or 4D vertex.

For points with extent, a single 3D or 4D vertex is transformed to display coordinates; this transformed vertex specifies the location of an image to be drawn.

Lines are defined by two 3D or 4D vertices, the endpoints.

Curved lines are defined by many 3D or 4D vertices, the control points. We assume that the curve lies on or within the convex hull of the control points.

Facets are flat polygons. They are defined by three or more coplanar 3D or 4D vertices, the vertices of the polygon. Facets may be as simple as triangles or quadrilaterals, or be as complex as fill area sets (multiple-contour, possibly self-intersecting polygons.)

Curved surfaces are defined by a grid of 3D or 4D vertices, the control points. We assume that the surface lies on or within the convex hull of the control points. Curved surfaces are bounded by curves.

2.1 Geometry Types of XGL Primitives

The table below lists the geometry types of present and possible future XGL primitives.

TABLE 2. XGL primitives and Geometry Types

PRIMITIVE	DIMENSIONS	GEOMETRY TYPE
xgl_annotation_text	2,3	P
*xgl_gen_triangle_strip	3	F

TABLE 2. XGL primitives and Geometry Types

PRIMITIVE	DIMENSIONS	GEOMETRY TYPE
xgl_image	2,3	PE
*xgl_zblit	3	P
xgl_multi_elliptical_arc	3	(C)
xgl_multi_simple_polygon	2,3	F
xgl_multiarc	2,3	(C)
xgl_multicircle	2,3	(S)
xgl_multiellipse	2,3	(S)
xgl_multimarker	2,3	P
xgl_multipolyline	2,3	L
xgl_multirectangle	2,3	F
xgl_nu_bspline_curve	2,3	C
xgl_polygon	2,3	F
xgl_quadrilateral_mesh	3	F
xgl_stroke_text	2,3	L
xgl_triangle_strip	3	F
*xgl_nu_bspline_surface	3	S
*xgl_multisphere	3	(S)
*xgl_multiconic	3	(S)

*possible new XGL primitives

xgl_gen_triangle_strip implements the Leo generalized triangle strip. xgl_zblit merges an image that has an associated z-buffer into the frame buffer xgl_nu_bspline_surface draws a general non-uniform rational B-spline surface, with trimming curves. xgl_multisphere draws spheres. xgl_multiconic draws general 3D conic surfaces (as a special case of NURB surfaces.)

For the primitives marked with (C) or (S) above, the objects are not specified using vertices, and thus do not fall within the six geometry types. A preprocessing step is needed to convert the data supplied to control vertex form. For example, a circle could be represented as a NURB with three rational Bezier curves (5 deBoor control points).

In the 2D pipeline case, it makes sense to have a geometry type “circle” (parameters x, y, and radius), and define a way to transform these parameters directly, since in this case a circle in **MC** typically remains a circle in **DC**. We do not believe that is worthwhile doing in 3D, where a circle typically ends up being an ellipse in **DC**.

3.0 The Conceptual Pipeline

To view objects, we process their geometry through a pipeline. The parameters to this pipeline determine how the objects will appear on the viewing window.

We will outline a conceptual pipeline that defines exactly what happens for all possible input parameters.

The steps of the conceptual pipeline are:

1. *Vertices are defined in model coordinates* **MC4**. 3D vertices just set the w coordinate to 1.
2. *Model transform*. Vertices are transformed by the model transform M from **MC** to world coordinates **WC** ($p_{WC} = Mp_{MC}$). To preserve dot products, unit normals are transformed by M^{-1} and renormalized ($n_{WC} = n_{MC}M^{-1}/|n_{MC}M^{-1}|$). If M is singular, we can not use the supplied normals (the lighting code will calculate the facet normal in **WC** and use it as the vertex normals. If the calculated facet normal is the zero vector, we substitute an arbitrary vector.)
3. *Rational w-clip*. Vertices at infinity (w == 0) in **WC** are removed. The object is clipped to the two planes w == +epsilon and w == -epsilon. Vertex color, position, and normals are interpolated linearly in **WC4** to get the values at the clipped points. This step allows division by w in the next step.
4. *Realize in* **WC3**. A rational w divide is done to get the vertices into **WC3**.
5. *Face determination for facets*. The sign of n.(e-p) is calculated (use n.e instead for parallel projections.) n is a vector perpendicular to the plane of the facet and pointing into the “front” half-space (i.e., a vector parallel to the facet normal), e is the location of the eye in **WC3** (or is a vector to the eye at infinity for parallel projections), and p is any point on the facet. The face is considered “front-facing” if this sign is positive or zero, and “back-facing” if negative. If n cannot be calculated (the facet is degenerate or M is singular), the facet is defined as front-facing.
6. *Light vertices*. The lighting computation is applied to each facet and vertex as appropriate. Generally, the facet normal or the vertex normal, and the vertex (all in **WC3**) are needed.
7. *Model clip*. Model clip planes are specified in **WC3** as the plane equation $n((x\ y\ z\ 1)^T-p) = 0$. n is a vector perpendicular to the model clip plane and pointing to the “positive” half space. p is any point on the model clip plane. This equation divides **WC3** into two parts: a half space for which the sign is positive or zero, and one for which it is negative. Multiple model clip planes may be specified. The acceptance region is defined as the set of points that satisfy some Boolean relation on the half spaces. (For example, we could accept ((plane 1 positive) or (plane 2 positive)).) All parts of the object that do not lie in the acceptance region are removed. Vertex colors and positions are interpolated linearly to get the values at the boundaries of the acceptance region.
8. *View transform*. Vertices are transformed by the viewing transform V from **WC3** to projection coordinates **PC4** ($p_{PC} = Vp_{WC}$).
9. *Device transform*. Vertices are transformed by the device transform D from **PC4** to device coordinates **DC4** ($p_{DC} = Dp_{PC}$). D is a scale and translate matrix only.
10. *View clip*. The object is clipped in **DC4** to the inside of the viewing volume [xmin, xmax] × [ymin, ymax] × [zmin, zmax] in **DC3**. Vertex colors and positions are interpolated linearly to get the values at the clipped points. For example, a vertex is inside the x bounds if xmin*w <= x <= xmax*w for w > 0.
11. *Realize in* **DC3**. A perspective w divide is performed to get the vertices into **DC3**.
12. *Render.* The object in **DC3** is drawn on the display. Additional clipping may be needed, either to the visible window through the window clip list, or of primitives that extend outside the window, such as is required for primitives like xgl_marker, xgl_annotation_text, xgl_image, and xgl_zblit.

3.1 Ordering Requirements

We impose the following ordering requirements on the conceptual pipeline:

- Tessellation must occur before lighting. This ensures that we do not have to light surfaces directly, but only need to light the resulting polygonal facets.
- Vertex lighting must occur before Model or View clipping. This ensures that the object remains consistently lit as the model or view clipping changes.

- Face determination must occur before Lighting. This ensures that the proper selection is made between front and back-facing attributes.

3.2 Planarity Requirement

The conceptual pipeline requires that facets be planar. If a polygon is non-planar, then it needs to be preprocessed into a collection of planar facets.

4.0 The Implementation Pipeline

The conceptual pipeline is not suited for fast implementation. Equivalent pipelines are needed that achieve the semantics of the conceptual pipeline, but are more efficient.

4.1 Assumptions

The main assumption is that the vast majority of applications use only a “well-behaved” subset (to be defined later) of all possible pipeline parameters and data. Thus we develop optimized pipelines for the well-behaved case, and modify these pipelines as needed to deal with “ill-behaved” cases. Given how rare we believe the latter cases are, the performance of the modified pipelines is less of an issue.

Another assumption is that a critical performance issue involves the speed of clip testing, rather than the speed of proper clipping. Objects are either trivially accepted or trivially rejected most of the time, and are occasionally clipped. Thus, the focus is on the performance of the accept and reject cases, and only secondarily on the clipped cases. This applies both to view and model clipping.

A third assumption is that the application does not change the model transform so often that the cost of transforming the lighting data (light vectors and position) to **MC** dominates the pipeline cost.

4.2 Methods

The following methods are used in the implementation pipeline to gain efficiency over the conceptual pipeline.

4.2.1 New Coordinate Systems

Two new coordinate systems are used: clip coordinates and lighting coordinates. Clip coordinates (**CC**) is a 4D space representing a transform of **PC** such that the view volume in **PC** is mapped to a canonical clip volume in **CC3**. This volume is $[-1,1]^3$, which is equivalent to $|x/w| <= 1$, $|y/w| <= 1$, $|z/w| <= 1$. **CC** allows clip testing to be performed more efficiently. The matrix that maps **PC** to **CC** is C. In this case, the entire viewing transform becomes $(DC^{-1})(CV)M$.

Lighting coordinates (**LC**) is a 3D space representing a Euclidean (rotation, translation, and reflection only) transform of **WC** such that the transform from **LC** to **DC** is by a sparse matrix. The viewing matrix V is factored into three matrices GQE. E is an Euclidean matrix, Q is an identity matrix plus one or two nonzero terms (for perspective and parallel projections, respectively), and G is a scale/translate matrix (with the z translate 0.) [Abi-Ezzi90a]

Here, the entire viewing transform is $(DC^{-1})(CGQ)(EM)$. **CC** can be obtained from **LC** by transforming by CGQ; this transform should be done by multiplying by Q first (1 multiply and 1 add in the perspective case), and then multiplying by CG (3 multiplies and 3 adds.) **LC** allows lighting in **LC** (instead of **WC**) because the Euclidean E does not distort any angles or distances. **LC** also allows an inexpensive transformation into **DC3** (DGQ is sparse) and improves performance when doing dynamic tessellation.

4.2.2 Well-Behaved Parameters and Data

One or more of the following restrictions on the pipeline parameters and data characterize our “well-behaved” cases.

1. *Invertibility*: V and D are nonsingular
2. ***MC** Lighting*: M is isotropic (consists of a uniform scale, rotate, translate, and reflect only; so it can’t be singular), and **MC** data is finite (**MC** data supplied is either 3D, or 4D with the w component always greater than zero.)
3. *Lighting Coordinates*: V is factorizable into GQE, M is affine (its last row is 0 0 0 1), M and D are non-singular, and **MC** data is finite.
4. *Single Pyramid*: The viewing volume in **WC3** does not wrap around infinity (i.e., the 8 corner points of the canonical clip volume in **DC4**, when mapped back to **WC4** via V^{-1}, all have w values of the same sign.) or the “clip to plus w only” flag is TRUE. Without loss of generality, the viewing volume in **CC4** or **DC4** is assumed to have only positive w’s. (If the volume is described by negative w’s, the viewing transform can simply be multiplied by -1 to work with positive w’s.)

“Well-behavedness” of the pipeline parameters allows both reordering and simplification of the conceptual pipeline. For example, if Invertibility and Single Pyramid, then it is possible to perform model clipping in **DC3**.

“Well-behavedness” of the vertex data also allows us to simplify the conceptual pipeline. For example, finite **MC** data and affine M allows us to avoid the rational w-clip step of the conceptual pipeline.

4.2.2.1 Invertibility

This restriction along with M being invertible allows us to transform vertices and normals to any coordinate space we wish to work in.

A singular D will cause the display to collapse into a line or a point, or lose all depth; thus we do not consider it very useful to implement. A singular V is equivalent to a singular M with a nonsingular V, so there is no reason to support singular V. We know of no applications that require both singular M and V.

4.2.2.2 MC Lighting

This restriction on the model transform and modeling data means that we can avoid the rational w-clip step, as well as perform lighting calculations cheaply in **MC**. The lighting data (e.g., light vectors and positions) will need to be transformed once to **MC**. The cost of doing that is assumed to be negligible compared with the rest of the pipeline and the rendering or with the alternative of transforming the vertices and normals to **WC**.

MC lighting is cheap whenever the length of nM^{-1} is the same for all unit normal vectors n; this occurs when M is isotropic. If M is Euclidean, the length of the **MC** normal transformed to **WC** is unchanged; if M is isotropic, it is multiplied by a constant independent of the vector. Also, we can get away with calculating the inverse of only the upper left 3 by 3 portion of M; this is easy to see because the bottom row is (0 0 0 1) and it is clear that the right column has no effect on the vector (it is a translate only, and thus cannot rotate the vector.) Thus, only the upper left 3 by 3 of M can affect the vector and thus only that part need be inverted.

We consider it very unlikely that an application would supply infinite **MC** data (even NURBs are usually specified with a positive w component.) The main use of a singular M is to do “fake shadows.” (We support this case.) Typically, M is isotropic; non-isotropic M’s fall into the Lighting Coordinates case.

4.2.2.3 Lighting Coordinates

This restriction on the model transform, modeling data, and the viewing transform means that we can reduce the cost of working with tessellated objects. Instead of tessellating in **MC** or even **WC**, and having to transform the resulting data via a full 4 by 4 viewing transform V (16 multiplies and 12 adds for rational tessellations) plus a scale/translate D (1 divide, 6 multiplies, and 3 adds), the object is tessellated in **LC** and then transformed via a sparse Q (one multiply and add) and a sparse DG (3 multiplies and 3 adds.)

Additionally, lighting coordinates save some effort when **MC** lighting is too expensive because M isn’t isotropic.

V is factorizable if and only if the viewing volume in **WC** (the rectangular prism viewing volume in **DC3** mapped back to **WC3**) has rectangular faces on the front and back planes (i.e. the planes corresponding to zmin and zmax in **DC3**.)

Using PHIGS viewing utility routines always gives a factorizable V. We consider it very unlikely that an application needs a non-factorizable view.

4.2.2.4 Single Pyramid

This restriction means that we don’t need to preserve the sign of w in **CC4** or **DC4** (because clipped objects will have positive w’s.) Clip testing can be done using a six bit clip code and model clipping can be done in **DC3** by transforming the model clip planes to **DC3** (see 4.2.11.)

We consider it unlikely that an application needs double-pyramid viewing (but we do support it.)

4.2.3 Efficient Face Determination and Clip Testing

Face culling and model and view clip testing should be done as early as possible to reduce the work of later pipeline stages. The cheaper tests should be done first; assuming that facet normals are available, the order should be: face, model, view.

Face distinguishing should be done in **DC3** if possible, since that requires just the sign of the z component of the facet cross product.

4.2.4 Clip Code Calculation

A fast routine to calculate clip codes for the canonical clip volume should be used for the view clip testing. There are two clip code functions needed, depending on whether we are doing single or double pyramid viewing. For single-pyramid viewing, the following pseudo-code routine calculates the clip code for a single vertex (x,y,z,w):

```
if (|x| > w)
    if (x < 0)    set x neg
    else          set x pos
```

and similarly for y and z. For double-pyramid viewing, negative w’s are allowed, and the routine needs to change to:

```
if (|x| > |w|)
    if (x < 0)        set x neg
    else              set x pos
```

and similarly for y and z. The routine also needs to preserve the sign of w.

It is possible to write branch-free C code that calculates the clip codes using the above (Appendix B contains a listing of the general case). This is done by working directly on the single-precision floating point values as though they were integers. On the superscalar Viking, that will execute significantly faster than using floating-point compares and branch on condition codes. For double-precision pipelines, it is still possible to work in integers only, but the performance gain is questionable.

If SPARC version 9 implements a “floating compare and set bit in register” instruction, an even faster implementation for the single-precision case may become possible.

In the double pyramid case, one extra bit needs to be added to the clip code (the sign of w), and a slightly-different accumulation routine is used to test for trivial accept and reject (you need to accumulate two “or” flags, one for positive w and one for negative w.)

4.2.5 Specialized Pipelines

It’s important to have pipelines optimized for specific combinations of geometry types and pipeline parameters. An efficient point pipeline is significantly different from an efficient facet pipeline.

To save code space, some pipelines may simplify the pipeline parameters, and then fall into other pipelines.

4.2.6 Dynamic Tessellation

In this paper, all curves and curved surfaces are assumed dynamically tessellated into multiple line segments and multiple triangles, respectively [Abi-Ezzi90b]. This tessellation occurs as a function of the pipeline parameters, and extracts a reasonable number of simple objects to satisfy a specified error criterion. Dynamic tessellation contrasts with static tessellation, which is done once in **MC**, and can have arbitrarily large errors.

Dynamic tessellation is done in one of three coordinate systems. It is done in **LC** if surfaces are being tessellated and lighting needs to be done. Otherwise, it is done in **CC** if the curves or surfaces (with no lighting) need to be clipped. Otherwise, it is done directly in **DC**. The **LC** tessellator also generates vertex normals directly in **LC**.

4.2.6.1 Dynamic tesselation details

Trimmed NURB surfaces are dynamically tessellated in a certain stage of the pipeline into triangles (NURB curves are processed in a similar fashion into lines), in a way dependent on the attributes that are active at that time. The dynamic tessellation approach involves the compilation (preprocessing) of NURB primitives into a form that is amenable to the fast extraction of triangles at traversal time; see [Abi-Ezzi89]. The extracted triangles depend on dynamically changing attributes which include the viewing transformation, the composite modeling transformation, and the approximation criterion. As attributes vary the given surface is approximated by using different sets of triangles, with the goal of maintaining roughly a constant pixel coverage per triangle in **DC** (which is another way of thinking about continuing to meet the approximation criterion).

Compilation is a significant aspect of the dynamic tessellation approach, in that it involves performing some complex operations once and then amortizing over these operations over and over again during subsequent traversals. The main operations during compilation are: conversion to Bezier patches, computing derivative bounds of each patch, determining front and back face cones of normals for face determination at the patch level, and finally for dissecting the trimmed region into a collection of simple regions called “vregions” per patch.

At traversal time the compiled forms of NURB surfaces are tessellated into triangles to within approximation thresholds. These thresholds are typically specified in **DC** (pixel space). In [Abi-Ezzi90a] we describe the lighting coordinate system as being ideal for the tessellation of NURBs. Essentially, we can achieve accurate lighting in **LC**, and we can efficiently map extracted and lit triangles to **DC** due to the proximity of **LC** to **DC**. Also, **LC** has some special features that facilitates the tessellation step size determination.

In [Abi-Ezzi91] we discuss methods for determining tessellation step sizes based on precomputed derivative bounds in order to meet approximation thresholds. At traversal time we map precomputed derivative bounds from **MC** to **LC**, and we map the approximation threshold from **DC** to **LC**. Once we have both the bounds and the thresholds in **LC** then we determine the tessellation step size based on some mathematical relations between the two. This technique avoids the costly computation of derivative bounds at traversal time.

In the ill-behaved cases of M being singular or projective, or V being non-factorizable (this includes being singular) then our scheme of step size determination and tessellation in **LC** is not possible any more. So, we substitute the following technique which is slow but works accurately for all these ill-behaved cases.

At traversal time, we essentially transform the control points of patches to **DC**, where we compute derivative bounds, and perform step size determination. Then we transform control points to **WC** where we perform tessellation and lighting based on the step sizes that we had determined. This approach lacks the two advantages of **LC**: we now need to compute derivative bounds for every traversal, and a full 4 by 4 is applied to extracted triangles to map them from **WC** to **DC**.

4.2.7 Pipeline Modules

The process of face distinguishing, model clip testing, transforming the vertex data to **CC**, view clip testing, and, if inside, transforming from **CC** to **DC**, can be done in a single function. This function should process a block of vertices to amortize the overhead of using the floating point registers. We process vertices through the entire pipeline, keeping as much information as possible in the registers.

There can be several different versions of these functions, call them “pipeline modules.” Each version can be optimized for input vertex data type (e.g., integer, single precision float, double precision float), output vertex data type (e.g., float x,y,z for triangles, and integer x,y for wireframe lines), and pipeline operations needed (e.g., the line module need not perform face determination.)

In some cases, the code in the pipeline module can be made independent of the geometry type. If only model and view clip testing need be done, the module just determines if the block is completely inside, outside, or properly clipped. This works to cull the object represented by the vertices because the object obeys the convex hull property.

4.2.8 Direct Face Distinguishing of Curved Surfaces

For NURB surfaces, some recent work [Shirman91] has shown that it is possible and inexpensive to calculate the “cone of normals” in the compilation step for an individual patch. This cone bounds the range of normals of all vertices on the patch. Thus, face determination can be performed on the entire patch, before tesselation. In order for this technique to work with nonisotropic M, we need to generalize our point cone test to elliptical cones.

4.2.9 Clip Testing By The Caller

One general model of how applications use graphics for display is the “database” model. In that model, the application maintains its information in a large database, in a form that is not usable by a graphics API. The database implements an operation “view” which allows the application user to see the data. This view operation performs hierarchial clip testing (using a bounding box, say) and passes attributes and geometry to the graphics API. The application knows, by virtue of its clip testing, whether or not the geometry passed to the API needs further clip testing or not.

Additionally, the curve and surface pipelines can perform clip testing on the entire object and pass the results down to the pipeline that actually draws the object.

There are four flags that can be passed to the optimized pipeline. Each flag is discussed below.

The graphics API should specify a method by which the application can pass either a cone of normals (both back and front facing) and get back the first two flags, or pass a set of vertices that form a bounding convex hull, and get back the last two flags.

4.2.9.1 do_face_culling

This flag is set (to TRUE) if the object needs face determination performed so that faces can be culled. Otherwise, the flag is cleared (to FALSE.)

4.2.9.2 do_face_disting

This flag is set if the object needs face determination performed so that faces can be distinguished. Otherwise, the flag is cleared (to FALSE.)

4.2.9.3 do_view_clipping

This flag is set if the object needs to be clipped against the view volume. Otherwise, the flag is cleared. Of course, if the object is known to be outside the clip volume, it is assumed that the pipeline is never even called.

4.2.9.4 do_model_clipping

This flag is set if the object needs to be clipped against the model clip volume. Otherwise, the flag is cleared. Of course, if the object is known to be outside the clip volume, it is assumed that the pipeline is never even called

4.2.10 Lazy Evaluation of Matrices

The various implementation pipelines need different matrices. These matrices should be evaluated only when needed and cached until they are invalidated by a change in M, V, or D.

4.2.11 Model Clipping in DC3

The model clip equation n.(q-p) = 0, where $q = (x\ y\ z\ 1)^T$ in **WC3**, can be rewritten as $(n_x\ n_y\ n_z\ n_w)$.q, where n_w is equal to -n.p. The term $(n_x\ n_y\ n_z\ n_w) = m$ contains the coefficients of the plane equation defining the model clip plane. If we multiply m.q by the w component of q in **WC4**, we have that sign (m.r) = sign(w), where r = q.w. Working in **DC4**, we have sign $(m(DV)^{-1}.(DV)r) = \mathrm{sign}(w)$. Dividing by w’, the w component of **DC4** $s = DVr$, we have sign $(m(DV)^{-1}.s)$ = sign(w).sign(w’). Thus, we can model clip in **DC3** as long as we preserve one or both of these signs (and neither D or V are singular.)

We can optimize in the well-behaved cases. For finite MC, sign(w) will always be positive. For single pyramid, sign(w’) will always be positive.

The pipelines following generally do model clip testing in one place and model clipping in another; the clip testing stage should save the sign so it can be used in the model clipping stage.

5.0 The Optimized Point Pipelines

5.1 The Well-behaved CC Point Pipeline

This pipeline is selected if all of the following conditions are met:

1. The primitive is of type P (Points)
2. Invertibility
3. Single Pyramid

The pipeline implementation is:

```
If do_model_clipping {
    Model clip testing in MC
        Case OUTSIDE -----------------------------> done
}
If do_view_clipping {
    Transform via CVM
    Calculate clip codes in CC4
    View clip testing
        Case OUTSIDE -----------------------------> done
        Case INSIDE
            W divide to CC3 and transform via DC^-1
} else
    Transform via DVM and w divide to DC3
Point renderer with window clipping if needed
```

No face determination is needed. Also, there is no clip case; everything is either accepted or rejected.

5.2 The Well-behaved CC Points With Extent Pipeline

For this pipeline, a different clip testing is done; specifically, clip testing against the front and back planes only is performed. The renderer has to do the clipping against the viewport extents.

This pipeline is selected if all of the following conditions are met:

1. The primitive is of type PE (points with extent)
2. Invertibility
3. Single Pyramid

The pipeline implementation is:

```
If do_model_clipping {
    Model clip testing in MC
        Case OUTSIDE -----------------------------> done
}
If do_view_clipping {
    Transform via CVM
    Calculate front and back clip codes (only) in CC4
    View clip testing
        Case OUTSIDE -----------------------------> done
        Case INSIDE
            W divide to CC3 and transform via DC^-1
} else
    Transform via DVM and w divide to DC3
Extent renderer with window & viewport clipping if needed
```

Note that the XGL definition of xgl_image states that the image itself is not model clipped, only the control point, so it is not necessary to model clip the extent.

6.0 The Optimized Line Pipelines

6.1 The CC Line Pipeline

This pipeline is selected if all of the following conditions are met:

1. The primitive is of type L (lines)
2. Invertibility
3. Single Pyramid

The pipeline implementation is:

```
If do_model_clipping {
    Model clip testing in MC
        Case OUTSIDE -----------------------------> done
        Case INSIDE
            Clear do_model_clipping flag
}
If do_view_clipping {
    Transform via CVM
    Calculate clip codes in CC4
    View clip testing
        Case OUTSIDE -----------------------------> done
        Case INSIDE
            W divide to CC3 and transform via DC^-1
        Case CLIPPED
            Homogeneous clip in CC4
            W divide to CC3 and transform via DC^-1
} else
    Transform via DVM and w divide to DC3
If do_model_clipping
    Model clip in DC3
Line renderer with window clipping if needed
```

See section 8.1 for a way to avoid window clipping in some cases.

6.2 The LC Curve Pipeline

This pipeline is selected if all of the following conditions are met:

1. The primitive is of type C (Curve)
2. Lighting Coordinates
3. Single Pyramid

The pipeline implementation is:

```
Compile NURB description into Bezier curve description in MC
If do_model_clipping {
    Model clip testing of control points in MC
        Case OUTSIDE -------------------------> done
        Case INSIDE
            Clear do_model_clipping flag
}
If do_view_clipping {
    Transform MC control points via CGQEM to CC
    View clip testing of control points in CC
        Case OUTSIDE -------------------------> done
        Case INSIDE
            Clear do_view_clipping flag
}
If we have control points in CC
    Transform to LC via $(CGQ)^{-1}$
else
    Transform MC control points via EM to LC
Perform step size determination in LC
If do_view_clipping {
    Use CC control points
    Tessellate in CC4
    Calculate clip codes in CC4
    View clip testing
        Case OUTSIDE ----------------------------> done
        Case INSIDE
            W divide to CC3 and transform via $DC^{-1}$
        Case CLIPPED
            Homogeneous clip in CC4
            W divide to CC3 and transform via $DC^{-1}$
} else {
    If we have control points in CC
        Transform control points via $DC^{-1}$ to DC4
    else
        Transform control points via DGQ from LC to DC4
    Tessellate in DC4 and w divide to DC3
}
If do_model_clipping {
    Model clip testing of control points in DC3
        Case OUTSIDE -------------------------> done
        Case CLIPPED
            Model clip in DC3
}
Line renderer with window clipping if needed
```

Note that we tessellate in **DC4** and output **DC3** data directly in the trivial accept case. For further optimization, the **DC** tessellator can do the float to integer conversion and also output just x and y values if it knew no z values were needed. This reduces redundant memory traffic.

7.0 The Optimized Facet Pipelines

7.1 The CC Facet Pipeline

This pipeline is selected if all of the following conditions are met:

1. The primitive is of type F (facets)
2. MC Lighting
3. Invertibility
4. Single Pyramid

The pipeline implementation is:

```
If do_face_culling {
    If facet normal n not supplied in MC
        Calculate facet cross product n in MC
    Calculate s = n(e-p) or s = ne^T (if parallel). p is any vertex on facet
    If facet is wrong facing -----------------> done
    Save s and/or n if needed later
    Clear do_face_disting
}
If do_model_clipping {
    Model clip testing in MC
        Case OUTSIDE -------------------------> done
        Case INSIDE, clear do_model_clipping flag
}
If do_view_clipping {
    Transform via CVM
    Calculate clip codes in CC4
    View clip testing
        Case OUTSIDE -------------------------> done
        Case INSIDE
            W divide to CC3 and transform via DC^-1 to DC3
            Clear do_view_clipping
        Case CLIPPED
            Save CC4 points
} else
    Transform via DVM and w divide to DC3
If do_face_disting {
    if do_view_clipping {
        If facet normal n not available in MC
            Calculate facet cross product n in MC
        Calculate s = n(e-p) or s = ne^T (if parallel). p is any vertex on facet
    } else
        Calculate s by calculating z component of facet cross product in DC3
    Save s and/or n if needed later
}
Calculate lighting using s, n, vertex, vertex normal in MC3
    If facet normal needed for lighting {
        If facet normal n not available in MC
            Calculate facet cross product n in MC
        Normalize n
    }
    Transform lights to MC3 (once per M)
If do_view_clipping {
    Homogeneous clip in CC4
    W divide to CC3 and transform via DC^-1 to DC3
}
If do_model_clipping
    Model clip in DC3
Facet renderer with window clipping if needed
```

e in the above is the eye location in **MC**. If no lighting is needed by the facet (e.g., vertex colors are already supplied and just need to be interpolated, such as when an application performs radiosity calculations), then we can remove the requirement of MC lighting from this pipeline, and also avoid doing some pipeline steps.

7.2 The LC Facet Pipeline

This pipeline is selected if all of the following conditions are met:

1. The primitive is of type F (facets)
2. Lighting Coordinates
3. Single Pyramid

The difference is the weakening of the restriction on M. The pipeline implementation is:

```
Transform via EM to LC
Transform vertex normals via (EM)^-1 to LC if needed
Perform the CC Facet Pipeline, with the following adjustments:
    M is identity (LC becomes MC, effectively)
    V is GQ (this is sparse)
    D is unchanged
    All the "do_..." flags are unchanged
```

Note that the implementation of this pipeline needs to take advantage of the special form of M and V. Also note that model clip culling needs to be done in **LC**.

7.3 The Curved Surface Pipeline (S)

This pipeline is selected if all of the following conditions are met:

1. The primitive is of type S (Curved surface)
2. Lighting Coordinates
3. Single Pyramid

The pipeline implementation is:

```
Compile NURB description into Bezier surface-patch description in MC
    Calculate cone of normals for patches
Test surface patches for size in DC
    If too small to tessellate, approximate by a few triangles in MC
    and use the CC or LC facet pipeline
If (do_face_culling or do_face_disting) and M is isotropic {
    Compare with eye position in MC
    If surface known to be front or backfacing {
        If facet is wrong facing ----------------------> done
        Clear do_face_culling and do_face_disting
        Save facing if needed later
    }
}
If do_model_clipping {
    Model clip testing in MC
        Case OUTSIDE --------------------------------------> done
        Case INSIDE
            Clear do_model_clipping
}
If do_view_clipping {
    Transform MC controls via CGQEM to CC
    View clip testing of control points
        Case OUTSIDE --------------------------------------> done
        Case INSIDE
            clear do_view_clipping
        Case CLIPPED
            Do guard band clip testing of patch
}
If we have control points in CC
    Transform to LC via (CGQ)^-1
else
    Transform MC control points via EM to LC
Tessellate in LC
    Compute vertex normals also (but not facet normals)
Perform the CC Facet Pipeline, with the following adjustments:
    M is identity (LC becomes MC, effectively)
    V is GQ (this is sparse)
    D is unchanged
    Use current values of do_face_culling, do_face_disting, do_view_clipping,
        and do_model_clipping
```

Note that the **LC** data generated will usually be homogeneous; because of Lighting Coordinates, we know that the w component is positive. Also note that we can use the cone of normals only if M is isotropic. Finally, model clipping is done in **LC** when we perform the **CC** facet pipeline.

We can further optimize this pipeline if we know that no lighting other than ambient is needed for the surface (e.g., we wish to show a wireframe model.) In that case, we should use an equivalent of the LC Curve pipeline (left as an exercise for the reader.)

8.0 Other Optimizations

8.1 Avoiding Unnecessary Window Clipping

If the x and y extents of the **DC** viewport is not partially occluded by other windows, then no window clipping is needed (except for things like markers and annotated text.)

If the intersection of the viewport and the visible window is a single rectangle, you can adjust V and D to make the view clipping also do the work of window clipping. If the object is a patterned line, it is necessary either to somehow adjust the pattern, accept a pattern that is affected by a change to the window size, or require window clipping for patterned lines (i.e., avoid this optimization for patterned lines.)

Window clipping may also be avoided if the display hardware supports a “window ID” attribute for each pixel. Software renderers can simulate window ID’s by using a bit mask.

8.2 Picking

Picking is handled by making the **DC** box the size of the pick aperture and running through the pipeline. For some primitives, there will need to be a “pick renderer”. For example, imagine that one is picking a z-buffered triangle. The triangle intersects the pick aperture, but it might be hidden by the z-buffer. So a pick renderer needs to be invoked for the (view-clipped by pick aperture) triangle which checks if it might write a pixel. If it would have, then it returns PICKED, else NOT PICKED. As soon as it knows it is PICKED, it can stop pick-rendering.

Trivial reject speed is very important for fast picking.

8.3 Speeding up clip testing of Points with variable device extents

If we know the maximum extent “m” that the object can take up in the viewing plane, we can adjust the x and y clip volume bounds to [xmin-m,xmax+m] and [ymin-m,ymax+m] and do the trivial reject against all six planes instead. The renderer will need to do window clipping against the original bounds. This is a technique known as “guard band” clipping.

8.4 Guard band clipping of Curved Lines and Surfaces

An additional performance gain is possible when doing clip testing of curved primitives. A larger clip box that contains the canonical clip volume is used; this is the “guard band”. Guard band clipping algorithms subdivide the object if it intersects the guard band and hope that some of the subdivided pieces will be trivially accepted or rejected. The following algorithm is called when the view clip testing step is handling the CLIPPED case:

```
Guard band clip testing of a curve/patch {
    If the number of lines or triangles in the curve or patch is too small
        It is not worth subdividing ----------> done, process as CLIPPED
    Compare against guard band
        Case GUARD INSIDE --------------------> done, process as CLIPPED patch
        Case GUARD CLIPPED
            Subdivide the patch
            Perform view clip testing on each subdivided patch
                Case INSIDE -------------------> done, process as INSIDE patch
                Case OUTSIDE ------------------> done
                Case CLIPPED
                    Recursively do guard band clip testing
}
```

Note that there is no GUARD OUTSIDE case in guard band clipping because it would have been detected in the view clip test step.

The number of lines or triangles may be too small for two different reasons:

- It will cost more to subdivide further than to tesselate
- The curve or patch is already so small that few lines or triangles will be generated

8.5 Batch Processing

Functions should process multiple vertices wherever possible. For example, apply the **CC** facet pipeline to entire triangle strips. Do not dissect them into triangles and process a triangle at a time. This requires a bit of extra work in face processing, since a strip may now be “twisted”: containing both front and backfacing facets.

8.6 Triangulation of Complex Facets

It may be faster to convert a complex facet (e.g., a fill area set) into triangles once, and then to process only triangles, especially where there is hardware support for rendering triangles but not for general polygons.

8.7 Optimizing Lighting Calculations

There are several ideas that are applicable to minimizing the cost of lighting calculations. For example, cone-of-normals testing can be applied to all of the lights available, and lights that are in the back cone of the patch do not need to be processed for any part of the patch.

8.8 Dynamic Pipeline Cost Analysis

The pipelines describe in sections 5-7 were ordered based on a static cost analysis of the floating point operations performed (adds, multiplies, divides, square roots, compares.) However, a dynamic cost analysis could determine a better ordering of pipeline operations. For example, view clipping occurs after model clip testing in the CC pipeline because (e.g, for triangles) view clipping requires one matrix multiply plus clip code calculation whereas model clip testing requires three (one per vertex.) However, if there were many model clip planes, then it would make sense to do the view clipping first. Another example is when parallel views are being used; view clip testing is 2D only in that case and it may make sense to do it earlier.

Dynamic cost analysis would calculate the cost of doing operations in different coordinate systems, and attempt to minimize the total cost of all operations needed.

9.0 Handling Ill-behaved Cases

What do we do when none of the above pipelines are selected given a set of pipeline parameters and data?

The choice is either to implement the conceptual pipeline itself, thus being correct for all parameter combinations [Abi-Ezzi89], or to take an expedient route, and give an answer that may differ from the conceptual pipeline.

We suggest that both approaches be combined. The rest of this section gives our suggestions on how to handle each possible example of ill-behavedness. Each suggestion is labelled with (correct) if it maintains correctness, or with (expedient) otherwise. Our main goal in making these particular suggestions is to avoid implementing the conceptual pipeline. As an aid to the application, wherever an expedient choice is made, a warning message should be generated.

9.1 Failure of Invertibility

A singular V or D means we don't do model clipping (expedient) or anything requiring lighting coordinates (expedient.)

For curves and surfaces, we treat singular V or D as though V were not factorizable; see 9.5 below.

9.2 Failure of Finite MC data

The data is processed as though the w terms were greater than zero (expedient for facets, surfaces, and model clipping, correct otherwise.) If w is equal to zero, it is set to a small positive value (expedient.)

Lighting may be incorrect for surfaces infinite in extent in **WC**. The wrong side of an object may appear when model clipped.

9.3 M is Projective

A 4 by 4 transform rather than a 4 by 3 is performed (correct). The w component is assumed to be positive in **WC4** (expedient for facets and surfaces, correct otherwise.)

Lighting may be incorrect for surfaces infinite in extent in **WC**.

9.4 M is singular

In this case, we do ambient lighting only (expedient.) We can continue to use the **MC** lighting pipelines.

9.5 V is Not Factorizable

In this case we merely substitute **WC** wherever **LC** appears: the "factor" E is set to identity, and the "product" GQ is set to V (correct.) Additionally, the step size and derivative bounds are calculated in **DC**, rather than in **LC**.

The three pipelines that use **LC** now use **WC**, and the only effect is additional work in the step-size determination, bounds determination, and transformation steps. See 4.2.6.1 for further details.

9.6 Double-pyramid Views

Use double-pyramid 7-bit clip codes in the testing step (correct), and pass the sign of w in **DC4** to the model clipper (correct.)

10.0 Summary

A conceptual 3D pipeline was defined and a set of well-behavedness conditions were specified. We assume that all significant applications are well-behaved. Efficient implementations of the conceptual pipeline for six geometry types were described.

Ill-behaved cases were handled either correctly or expediently. Our suggestions, if accepted, avoids implementing the conceptual pipeline at the expense of inaccurate treatment of some ill-behaved cases.

Hardware implementations of the 3D pipeline may benefit from this work.

We suggest that the system described in this paper be implemented as part of XGL 3.0. In the long term., this code is intended to become the best-performing software implementation of the 3D pipeline in the industry.

11.0 References and Bibliography

Abi-Ezzi89	Salim Abi-Ezzi, “The Graphical Processing of B-Splines in a Highly Dynamic Environment”, RPI Ph.D. dissertation, RDRC-TR-89001, Troy, New York (May 1989)
Abi-Ezzi90a	Salim Abi-Ezzi and Michael Wozny, “Factoring a Homogeneous Transformation for a more Efficient Graphics Pipeline,” Computer Graphics Forum 9, North-Holland (1990)
Abi-Ezzi90b	Salim Abi-Ezzi, “The Dynamic Tesselation of Trimmed NURB Surfaces”, Sun internal report (1990)
Abi-Ezzi91	Salim Abi-Ezzi and Leon Shirman, “Tesselation of Curved Surfaces under Highly Varying Transformations,” Eurographics ‘91 Proceedings, Vienna, Austria (September 1991)
Shirman91	Leon Shirman, personal communication (1991)
Stolfi91	Jorge Stolfi, “Oriented Projective Geometry”, Academic Press (1991)

12.0 Appendix -- Branch-free clip code calculation

```
/*  return 7 bit clip code:
    cc =(z neg)(z pos)(y neg)(y pos)(x neg)(x pos)(w neg)
    for double-pyramid clip testing.
    This evaluates the basic test:
        if (w < 0) set w neg /**/
        if (|x| > |w|)
            if (x < 0) set x neg
            else set x pos
        if (|y| > |w|)
            if (y < 0) set y neg
            else set y pos
        if (|z| > |w|)
            if (z < 0) set z neg
            else set z pos
    using branch-free code. The code takes advantage of the fact
    that IEEE floating point is a sign-magnitude representation
    and that single precision (float) fits in an int. This code
    may need to change for SPARC V9 or non SPARC CPU's.

    To test multiple vertices:
        /* Initialize */
        andflag = -1;
        orflagp = 0;
        orflagm = 0; /**/

        /* For each vertex, calculate cc and then */
        andflag &= cc;
        orflagp |= cc;
        orflagm |= cc ^ 1; /**/

        /* after all vertices processed, perform testing */
        if (andflag != 0) trivial reject
        if (orflagp == 0 || /**/orflagm == 0) trivial accept

    The single pyramid case is simpler; don't do the parts above
    flagged with /**/.

    This code expanded from Tim Van Hook's original implementation
*/

clip_test(v) int *v;/* ptr to float x,y,z,w */
{
int flag;
int x, y, z, w;
int s, t, u;
int sign = 0x80000000;

w = *(v + 3);
s = w & ~sign;                  /* abs(w) */
flag = (unsigned int)w >> 31;   /* extract sign */ /* bit 0: w neg */

x = *(v + 0);
t = x & ~sign;                  /* abs(x) */
u = s - t;                      /* compare to abs(w) */
u = (unsigned int)u >> 31;      /* extract sign, 1 is abs(x) > abs(w) */
t = (unsigned int)x >> 31;      /* x sign */
t += 1; t = u << t; flag |= t;  /* bit 1: x pos bit 2: x neg */

y = *(v + 1);
t = y & ~sign;                  /* abs(y) */
u = s - t;                      /* compare to w */
u = (unsigned int)u >> 31;      /* extract sign, 1 is abs(y) > w */
t = (unsigned int)y >> 31;      /* y sign */
t += 3; t = u << t; flag |= t;  /* bit 3: y pos bit 4: y neg */

z = *(v + 2);
t = z & ~sign;                  /* abs(z) */
u = s - t;                      /* compare to w */
u = (unsigned int)u >> 31;      /* extract sign, 1 is abs(z) > w */
t = (unsigned int)z >> 31;      /* z sign */
t += 5; t = u << t; flag |= t;  /* bit 5: z pos bit 6: z neg */

return flag;
}
```