LSU EE 7700-2 **Set 1: Elements of Real-Time 3D Graphics** Spring 2011

David Koppelman

1.1 Introduction

This set of lecture notes covers elements of real-time 3D graphics, including the mathematics coordinate transformation and projection and the typical organization of such software.

These notes will serve primarily to define terms and provide a basic description of what's covered. If additional explanation is needed see the references.

This set of notes covers elements of 3D graphics and the OpenGL specification. One omission from this set is the topic of texturing, that will either be covered in another set or added to this one later.

1.1.1 Demo Program

Some parts of these notes refer to a sample program, named demo-4-lighting and available for viewing via `http://www.ece.lsu.edu/gp/code/cpu-only/demo-4-lighting.html`or from the course svn repository `https://svn.ece.lsu.edu/svn/gp/cpu-only/`.

This program implements all of the rendering pipeline in software and writes, not the real frame buffer (defined below), but a simulated one.

1.2 Overview

3D graphics is the display of representations of 3D objects on a monitor or some other display, as part of a game, training simulation, animation, etc. For **Real-time** 3D graphics the amount of time it takes to prepare a **frame** for display must be less than some threshold, perhaps 15 ms. The time restriction imposes limitations on what can be displayed and motivates the design of hardware (GPUs) that can prepare frames quickly enough.

1.2.1 Program Parts: Application, Rendering Pipeline, and Glue

In these notes the term **program** will be used to refer to an entire piece of code, for example, an entire flight simulator program. The term **application** will refer only to those parts of a program not specifically concerned with visual display. The broader term **application domain** will refer to the application and the non-graphics-related expertise needed to develop it. For a flight simulator program that would include the expertise and code for determining aircraft engine speed, as well as things like aircraft position and speed. The term **rendering pipeline** will refer to the part of the program which converts some generic visual description (the positions of zillions of triangles, etc) into an image. The rendering pipeline might just be software running on the CPU, or it could be partly implemented as hardware.

The application might be written by aerospace engineers, the rendering pipeline would be written and designed by programmers and computer engineers. Between the application and rendering pipeline might be some glue software that takes representations of, say, airplane position and scenery provided by the application and converts it into the generic form used by the rendering pipeline.

The term **pipeline** is used loosely here, it conveys the fact that the same sequence of operations are applied to a large number of objects. It is not to be confused with the much stricter definition used in EE 4720 (computer architecture), which referred to a particular style of hardware implementation. Though the rendering pipeline in part might be implemented in hardware it is unlikely to realize a one-vertex-per-cycle throughput, nor would it necessarily stall one vertex if the processing of a predecessor vertex took more than the expected amount of time.

For many years special hardware in one form or another has been used to improve the performance of the rendering pipeline. In modern systems a **graphics processor unit** (GPU) is used to run much of the rendering pipeline. This course will cover GPU design, starting with a software-only rendering pipeline, and proceeding to special hardware that can outperform the software-only implementation.

As used here the rendering pipeline can refer to something which is purely software, purely hardware, or a mixture of the two. In modern systems the rendering pipeline would start with software on the CPU, perhaps a part of the program. The pipeline extends into driver software (written by the GPU vendor) also running on the CPU and then crosses over to the GPU where it might include a mixture of vendor software, user (application developer) software, and fixed-functionality sequential machines.

Example—Application / Rendering Pipeline Boundary

In demo-4-lighting the boundary between application code and rendering pipeline code is marked with a comment "Rendering Pipeline Starts Here," above the comment is the code determining the location of the triangles forming the tube in object space. (Not as elaborate as a flight simulator, but easier to understand.) Below the comment is rendering pipeline code.

1.2.2 Frame Buffer

The ultimate goal of the rendering pipeline is to display an image, it does so by writing a **frame buffer**, an area of memory which is **scanned out** (read) by hardware that packages the data and sends it off to a **display** (such as an LCD monitor).

A display consists of an array of addressable locations called **pixels**; the color of each pixel can be independently set. For most current displays the color of each pixel is determined by three components, **red**, **green**, and **blue**, with the intensity of each component set by an integer in the range $[0, 2^8)$ for ordinary displays and $[0, 2^{12})$ for cinematic displays. A zero is used for the dimmest (black) and the maximum number is the brightest.

A frame buffer can be thought of as an array, with each array element corresponding to a pixel. The frame buffer used in course code samples will have 32 bits per pixel with the first byte ignored, the second byte red, the third byte green, and the fourth byte blue. The first element will correspond to the leftmost pixel of the bottom row of pixels on the display, the second element of the frame buffer will correspond the second pixel on the bottommost row, etc.

In reality the frame buffer layout will vary with display and **display mode**. On some systems the frame buffer might be in the CPU's memory, in others it is part of the GPU.

Example—Frame Buffer

The frame buffer in demo-4-lighting is pointed to by variable `f buffer`. Near the end of routine `render light` it is written. (The object `frame buffer` holds the pointer to the frame buffer itself but encapsulates other information, such as the frame buffer size.)

1.2.3 Object Representation

There are many ways to represent 3D objects for display, most systems do so in terms of a small number of items, called **rendering primitives**, or **primitives** for brevity. In fact most systems use essentially three primitives for 3D objects: triangles, lines, and points. Though such systems may accept other primitives, such as rectangles, they likely decompose them into triangles.

Curved surfaces are approximated by triangles and solid objects are represented by their surface. The approximation of curves by triangles is quite effective. The approximation of solids by surfaces is sufficient for current generations of GPUs which cannot realistically model the passage of light through translucent solids. (That is, they don't use **ray tracing** techniques.)

The description of primitives in these notes are of the kind used by OpenGL, and reflect the compromises needed to ensure high performance. These issues will be discussed later in the course.

A primitive has one or more **vertices** (a triangle has three, a line has two, etc.). A vertex in the sense used here includes a **coordinate** corresponding to the location of the vertex, plus **attributes** including a **material color**. (The ultimate appearance of the vertex depends on its material color and lighting conditions.)

Example—Object Representation

In demo-4-lighting there is only one primitive: a triangle. The object `vtx list` is a list of vertices, the first three vertices make up the first primitive, the second three vertices are the second primitive, etc.

1.3 Coordinates, Vectors, and Transformations

The material in this section is based on Foley, van Dam, Feiner, and Hughs Chapter 5, Chapter 6, and the appendix.

1.3.1 Coordinates, Vectors

The term **point** will refer to a position in space and **coordinate** will refer to the representation of that position, the two terms may be used interchangeably when there is no chance of confusion. In class a right-handed coordinate system will be used.

In class two representations of a coordinate will be used, ordinary Cartesian coordinates (positions along an x, y, and z axis) and what are called **homogeneous coordinates**. Ordinary coordinates will be expressed as three element column vectors, $\begin{bmatrix} x \\ y \\ z \end{bmatrix}$, though for brevity they may be written as 3-tuples, (x, y, z). Uppercase letters, often P and Q, will be used to denote points or their coordinates (homogeneous or ordinary).

A homogeneous coordinate is a four-element column vector, $\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$, that might be thought of as a redundant (though useful) way of representing a point. Homogeneous coordinate $\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$, $w \neq 0$ represents the same point as ordinary coordinate $\begin{bmatrix} x/w \\ y/w \\ z/w \end{bmatrix}$. A homogeneous coordinate in which the w component is 1 is said to be **homogenized**. Any homogeneous coordinate with $w \neq 0$ can be homogenized by dividing each component by w.

A **vector** is a difference between two points and is represented by a 3- or 4-element column vector; in the 4-element representation the last element must be zero.

The letters U and V will usually be used to denote vectors. If P and Q are points then PQ indicates the vector $Q - P$ (from P to Q). The length of vector $V = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$, denoted $\|V\|$, is given by $\sqrt{x^2 + y^2 + z^2}$.

A vector V is called a unit vector if $\|V\| = 1$. A vector is **normalized** by dividing each of its components by its length. Let V be any vector, define $\hat{V}$ to be $V/\|V\|$, the normalized version of V.

Let $V_1 = \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix}$ and $V_2 = \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix}$. Their **dot product**, denoted $V_1 \cdot V_2$, is the scalar $x_1x_2 + y_1y_2 + z_1z_2$. If the dot product of two vectors is zero they are orthogonal (perpendicular). Let V be any vector and N be any unit vector; $V \cdot N$ gives the projection of V in the direction N. (Victor and Nancy start running from the same position, Victor with velocity V and Nancy in direction N. If Nancy ran at speed $V \cdot N$ in direction N she would be able see Victor exactly to her left or right.)

Let $V_1 = \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix}$ and $V_2 = \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix}$. Their **cross product**, denoted $V_1 \times V_2$, is given by $\begin{bmatrix} y_1z_2 - z_1y_2 \\ z_1x_2 - x_1z_2 \\ x_1y_2 - y_1x_2 \end{bmatrix}$. If $V_3 = V_1 \times V_2$ then both V_1 and V_2 are orthogonal to V_3.

A vector orthogonal to a plane is called the plane's **normal**. The term normal will also be applied to surfaces.

1.3.2 Basic Transformations

A **transformation** is a mapping from one coordinate set to another (*e.g.*, from feet to meters) or to a new location in an existing coordinate set.

The transformations to be considered here are **translation** (movement), **scaling** (change in size), **rotation**, and **projection** (mapping 3D coordinates to 2D).

The transformation of a coordinate will be realized by multiplying a **transformation matrix** by the coordinate. That is, $P\prime = MP$ gives point P mapped using transformation matrix M. Coordinates are homogeneous and the matrices are 4×4.

The **scale transformation** $S_{s,t,u}$ stretches an object by s along the x-axis, t along the y-axis, and u along the z-axis; it is given by:

$$S_{s,t,u} = \begin{pmatrix} s & 0 & 0 & 0 \\ 0 & t & 0 & 0 \\ 0 & 0 & u & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Scaling is centered on the origin. Note that $S_{1,1,1}$ is the identity matrix (and so of course leaves coordinates unchanged).

The **rotation transformation** $R_{\mathrm{Z}(\theta)}$ rotates points counterclockwise around the z-axis; it is given by:

$$R_{\mathrm{Z}(\theta)} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Similar matrices rotate about the other axes. The **translation transformation** $T_{s,t,u}$ moves points in the indicated direction, it is given by

$$T_{s,t,u} = \begin{pmatrix} 1 & 0 & 0 & s \\ 0 & 1 & 0 & t \\ 0 & 0 & 1 & u \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Translation is the first transformation here for which homogeneous coordinates are necessary. It is easy enough to translate an ordinary coordinate by adding a vector, but it is impossible to achieve translation by multiplying by a matrix. With homogeneous coordinates one *can* use matrix multiplication to achieve translation. That would be wasteful if all one wanted to do was translation, but typically a number of transformations are applied to coordinates. If all of these transformations can be achieved by matrix multiplication the matrices can be pre-multiplied, say $M = M_1 M_2 \cdots M_n$, and then by multiplying each coordinate by just M all of the transformations are performed.

Example—Transformation Matrices

In demo-4-lighting matrices initialized to the scale and translation transformations described above can be created by instantiating `pMatrix Scale` and `pMatrix Translate`. Search for `center_eye` so see an example.

1.3.3 Perspective Projection Transformation

As typically defined a projection transformation maps coordinates from a 3D space to a 2D space, for our purposes the 2D space will be window coordinates. The **projection transformation** defined here will map from 3D space to 3D space, but in such a way that the x and y coordinates (after further transformation) can be used as window coordinates; the transformed z coordinate will be used to determine object visibility.

Following OpenGL notation the coordinates to be transformed will be said to be in **eye space** and the transformed coordinates will be said to be in **clip space**.

A projection can be defined by a viewer location and a **projection window** location; the projection window location should be chosen to correspond to the expected position of the user's computer monitor; both the viewer location and projection window location would be in eye coordinates. The **projection plane** is the plane in which the projection window lies.

Let E be the coordinate of the viewer. A point P is projected by determining a point Q that is on the projection plane and is on the line formed by E and P; point Q may or may not lie within the projection window itself. (If it doesn't lie within the window it shouldn't be displayed.)

Though it would be possible to derive a perspective transformation defined in terms of an arbitrary user and window location the **frustum projection transformation** shown below is for a viewer located at the origin and a projection window parallel to the xy plane and centered somewhere on the negative z axis.

No generality is lost by placing the viewer at the origin because the transformation can be multiplied by translations and rotations to place the viewer in the correct location.

With eye space defined this way it is easy to show that the projected x and y coordinates are inversely proportional to z: x/z, y/z.

The definition of the projection given above maps all points. For our purposes we are interested only on projected points that fall in the projection window (not just the projection plane). Further, we are usually not interested in points behind the viewer and may not be interested in points in front of, but very close to, the viewer. For efficiency's sake we might not be interested in points further than a certain distance from the viewer (they would take time to compute but would contribute little or nothing to the display).

The **view volume** is the part of the eye space which is to be visible. The shape of the view volume is a **frustum** (a pyramid with the top cut off). The apex of the frustum (if it had one) would be at the viewer location (the origin), the top of the frustum (closest to the viewer) is the projection window and is in what is called the **near plane**, the bottom of the frustum is in what is called the **far plane**.

The frustum transformation given below transforms coordinates from eye space to clip space. As eye space puts the viewer and projection window in convenient places, clip space puts the view volume in a fixed, convenient space: a cube centered on the origin with edges of length 2. That is, in clip space coordinate $\begin{bmatrix} x \\ y \\ z \end{bmatrix}$ is within the view volume iff $x, y, z \in [-1, 1]$. The alert reader will have guessed that clip space is called clip space because it is a convenient place to clip. (Additional transformations are needed to generate window coordinates, but these only need operate on x and y (after homogenization).)

The **frustum projection transformation**, $F_{n,f,w,h}$, transforms coordinates for a viewer at the origin facing in the $-z$ direction with a near plane at distance n from the viewer, a far plane at distance f from the viewer, a window width of w (centered on the z axis), and a window height of h (also centered). It is given by:

$$F_{n,f,w,h} = \begin{pmatrix} n\frac{2}{w} & 0 & 0 & 0 \\ 0 & n\frac{2}{h} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -2\frac{fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}.$$

The **field of view** is the angle between the viewer and the sides of the projection window on the xz plane. For a given value of $n > 0$ one can choose a w to achieve a desired field of view, for smaller n the perspective foreshortening will be more apparent. The transformation $\lim_{n\to\infty} F_{n,n+f,w,h}$ realizes an orthographic projection.

In this class only the frustum transformation above will be considered.

1.4 Operations on Primitives and Vertices

1.4.1 Clipping and Scissoring

Since only vertices inside the view volume are to be visible those outside the view volume ought to be discarded at some point, the earlier the better. **Clipping** is the process of deciding which parts of a primitive are in the view volume and either discarding the primitive (if the answer is none), passing the primitive (if the answer is all), or generating new primitives corresponding to the visible portions (if the answer is some).

To clip, one has to determine where (if at all) a primitive's edges intercept the view volume boundaries, called **clip planes**. Note that the frustum transformation above conveniently places the clip planes. If none of the edges intercept the clip planes then either the entire primitive is inside or outside the view volume, these are the easy cases since in the former case the primitive is unchanged and in the later case the entire primitive is discarded. If some edges intercept the clip planes clipping will entail finding new primitives.

Scissoring is like clipping, except it is applied to individual fragments (see below). Since a fragment is like a point, the test is easy, discard the fragment if its window coordinates are outside the window or its z value is out of range (see the section on z buffering below).

If clipping is done perfectly then scissoring is unnecessary (though it still might be desired for special purposes), and if scissoring is done then clipping is not necessary for correctness. Some systems clip the easy primitives and leave the rest for scissoring.

Example—Scissoring

The demo-4-lighting program does not perform clipping. Scissoring is performed using the `pInterpolate` class.

1.4.2 Lighting

The techniques described above can be used to determine exactly where on the window a primitive will lie. In contrast, the lighting techniques used by current real-time 3D graphics systems produce only a gross approximation of the color of the object.

Ultimately a pixel will be written with the color of the object at that position. The color to write is determined by the material properties of the object (something like color under ideal conditions) and lighting conditions.

To correctly compute this color one would have to follow the trajectory of light rays emanating from lights, following them until they reached the viewers eye. Following the trajectory of a ray means testing whether a ray intercepts *every primitive in the view volume*, and choosing the closest one to the light. From that primitive determine how the color will change and the directions the reflected light ray will take. This will be repeated until the ray reaches the viewer's eye. This technique, called **ray tracing**, is extremely time consuming, even after optimizations are applied, such as starting from the viewer and tracing rays until they reach a light source. Current systems (and the author hopes this becomes dated soon) cannot ray trace fast enough for real-time graphics. Ray tracing is routinely used for animation, for which real time rendering is not needed. Later in the course more may be said about future GPU designs that might support it.

The following describes the simple lighting model used in current 3D graphics systems and which can easily realize sufficient speed for real-time purposes. The lighting model is only a rough approximation of how real objects are lighted and a lot of programmer energy is spent finding ways of making things halfway realistic within these restrictions.

In the OpenGL model vertices are assigned not colors, but **material properties** and **normals**. The material properties are essentially colors, they can be though of as colors under ideal lighting. The normal is a vector orthogonal to the vertex and indicates the direction it is facing. One might expect it to be based on the vertex's primitive, for example, if the primitive were a triangle the normal would point away from (be orthogonal to) the plane in which the triangle was embedded. If the primitives were being used to approximate a smooth surface then the normal might be based on the surface, rather than the primitive.

In addition to vertex properties lighting sources are defined. A light source has a location and an intensity (actually there are separate intensities for each color component and each type of lighting).

In the model every vertex has an unobstructed view of every light (it doesn't matter that some primitives are in the way). That means there are no shadows and no reflections, though both effects can be approximated by other means.

The following can contribute to the color of a vertex: the distance from the vertex to the light source, the angle between the vertex normal and the light (used for **diffuse** lighting), and the combined position of the user, light, and vertex normal (used for **specular** lighting).

Each vertex has separate material properties (colors) for diffuse lighting, specular lighting, ambient lighting, and emissive lighting. Each light source has separate ambient, diffuse, and specular intensities.

For a complete description of how these properties and intensities are used to compute a color see OpenGL 2.1 Specification Section 2.14.1.

Under OpenGL this light computation is applied to each vertex.

Example—Lighting

Program demo-4-lighting uses diffuse lighting, search for "Apply Lighting" Pressing 'd' toggles attenuation (change in intensity with distance), pressing 'a' toggles the use of normals, and pressing 'n' switches between using the triangle's surface normals and the tube's surface normal. The arrow keys, Page Up, and Page Down can be used to move the light.

Note that there are no shadows and that when the light is near, the large triangle is not realistically lighted (because lighting is based on the vertices).

1.5 Rasterization, a.k.a. Scan Conversion

The rendering pipeline up until this point has operated on vertices. In the **rasterization** step vertices, now in window space, are grouped into primitives and converted into **fragments**, each fragment is processed by subsequent steps in the rendering pipeline. Rasterization is also called **scan conversion**.

A fragment is the portion of a primitive that covers one pixel. If a triangle covers, say, 100 pixels then it will be rasterized into 100 fragments. Because there are many more fragments then vertices, the time and hardware needed to process them can be much larger than that needed to process vertices.

Rasterization proceeds by **interpolating** two vertices, that is, finding points on the line connecting them. When interpolating in the x direction each successive point will differ by 1 in x, likewise for interpolating in the y direction. For example, let $P = \begin{bmatrix} 5 \\ 10 \end{bmatrix}$ and $Q = \begin{bmatrix} 15 \\ 40 \end{bmatrix}$. Interpolating in the x direction yields $\begin{bmatrix} 5 \\ 10 \end{bmatrix}, \begin{bmatrix} 6 \\ 13 \end{bmatrix}, \begin{bmatrix} 7 \\ 16 \end{bmatrix}, \ldots, \begin{bmatrix} 15 \\ 40 \end{bmatrix}$. Let $PQ_{i,\star} = \begin{bmatrix} x_P + i \\ y_P + i\frac{y_Q - y_P}{\texttt{x}_\texttt{Q} - \texttt{x}_\texttt{P}} \end{bmatrix}$, so that $PQ_{i,\star}$ is the ith interpolated point in the x direction. Interpolating in the y direction yields $\begin{bmatrix} 5 \\ 10 \end{bmatrix}, \begin{bmatrix} 5\frac{1}{3} \\ 11 \end{bmatrix}, \begin{bmatrix} 5\frac{2}{3} \\ 12 \end{bmatrix}, \ldots, \begin{bmatrix} 15 \\ 40 \end{bmatrix}$; similarly let $PQ_{\star,i}$ denote the ith interpolated point in the y direction.

Consider the triangle formed by points PQR. One might rasterize by interpolating QP and QR in the y direction and then for each pair of points $QP_{\star,i}$ and $QR_{\star,i}$, interpolating in the x direction (performing what might be called nested interpolation).

The rasterization described above will find all pixels completely covered by the triangle and is sufficient for our purposes. For others' purposes one would need to determine which pixels are *partly* covered by the triangle (those along the perimeter).

When interpolating two vertices it is useful to interpolate other quantities too, including color components and the z coordinate. For example, let vertex P have red component r_P and vertex Q have red component r_Q. Then the red component for the ith interpolated point from P to Q in the x direction is $r_P + i\frac{r_Q - r_P}{\texttt{x\_Q-x\_P}}$;, note that the expression is exactly analogous to the one giving the y value.

In summary, rasterization converts a primitive (say, a triangle), into fragments. Each fragment consists of a coordinate in window space and an interpolated z value and color components. A primitive consisting of three vertices can be converted into hundreds of fragments and so any further steps are designed to be as computationally efficient as possible.

Example—Rasterization and Interpolation

In demo-4-lighting rasterization is performed under the comment "Rasterize Primitives". A class, `pInterpolate`, is used to compute the x or y values and also to scissor and interpolate z values and color components.

1.6 Frame Buffer Update, Z-Buffering, Blending

One might think of frame buffer locations as pixels, because the value at a location corresponds to a pixel on the display. One might be tempted to think of the fragments produced by rasterization as pixels too, but that would be wrong because fragments may never be written to the frame buffer, or they might be modified.

Before a fragment gets written it must pass a series of tests, the one described here is a z-buffer test, which ensures that fragments closer to the viewer are visible regardless of the order in which the corresponding primitives entered the rendering pipeline.

The **z-buffer** is an array with the same number of elements as the frame buffer (and may be thought of as part of the frame buffer, and may indeed share storage).

Each element of the frame buffer holds the z coordinate of the fragment in the corresponding pixel in the frame buffer, or if unoccupied, some maximum value. A new fragment is written to the frame buffer and z buffer if its z coordinate is smaller (closer to the user) than the value in the z-buffer.

The z-buffer test is one of many that control whether a fragment is written, others won't be considered in the course but curious readers can look up stencils in the OpenGL documentation.

It is also possible to **blend** a new fragment with the current occupant of a frame buffer location, simulating partial transparency among other things. Blending consists of some combination of the corresponding red, green, and blue components of the arriving and incumbent fragments, such as making the new red the average of the arrival and incumbent values.

Example—Z-Buffer

In demo-4-lighting the z-buffer (pointed to by variable `z_buffer`) is allocated each time the frame is rendered, real systems would only allocate when the window size changed. (Search for `malloc`.) It is updated in the innermost loop of the rasterization nest (search for `linez`).

1.7 GPU Application Programmer Interfaces (APIs) - OpenGL and Direct3D

Consider a purely software 3D graphics library. Users could specify the primitives described above (plus others), coordinate space transformations, user position, lighting, etc. The designer of that library might provide functions that would provide the user with great flexibility achieved with clear and uncluttered user code. Internally, the library too would be clear and uncluttered, and would make use of fast, memory-lean algorithms.

Now suppose one wanted to move the library to a GPU. The user code would make the library calls on the CPU, these calls might do some computation on the CPU and see that the rest be done on the GPU. The resulting image should be the same as the one that would be produced by the earlier software-only implementation of the library.

Suppose the library produced some effect, say the blending of colors of nearby objects, that the GPU couldn't do. Then either all rendering would have to be done on the CPU and so there would be no speedup, or else the blending effect would not be shown. Either way the user would not be happy.

If the library designer knew from the outset that such blending was infeasible then either the feature would not be included, or would be optional and documented as potentially slow. But for the library to be long-lived and usable on many GPUs, it would have to disallow things that make GPUs slow without disallowing things GPUs can do fast.

OpenGL and Direct3d are two major examples of such libraries. OpenGL originated as Mesa, a library developed by Silicon Graphics. OpenGL is defined as a specification, not as a specific piece of software. A library meeting this specification (which *is* a piece of software) is called an **implementation**. (Many standards for computer languages work like that.) There are OpenGL implementations for many systems, including the many flavors of Unix and MS Windows. Direct3D, in contrast, runs only on Windows systems.

Both OpenGL and Direct3D organize the rendering pipeline into a sequence of steps, the OpenGL steps are described below. They impose restrictions on things that GPUs cannot do well. An important restriction is limiting the degree to which vertices can affect each other. This restriction provides the hardware with flexibility in where and when vertex operations are performed (since there is no need to pass a result from one vertex to another). The OpenGL steps are outlined in the next section. More on the API itself and the reasons for the restrictions will be covered later in the course.

1.8 OpenGL Coordinate Spaces and Rendering Pipeline

The OpenGL rendering pipeline starts with primitives (triangles, and other convex polygons) specified in a user-convenient **object space**. Attributes can be attached to the vertices making up the primitives, including material properties such as color (which is not the final color of the vertex) and a **normal** which indicates the direction a vertex is facing. (It is useful to make the vertex normal different than the primitive's mathematically correct normal when the primitive is approximating a curved surface. See demo-4-lighting.)

Using a user-provided **model view transformation** vertices in object space coordinates are transformed to **eye space**. As described earlier, in eye space the viewer is assumed to be at the origin facing the $-z$ direction. Lights are specified by the user in eye space coordinates. In eye space a color is computed for each vertex based on its material properties, normal, and the location of the lights (with the viewer at the origin). This color joins the vertex on its passage through the pipeline.

Using a user-provided **projection matrix** coordinates are transformed to **clip space**. In clip space the clip volume is a cube centered on the origin with a length of 2, so any point with a coordinate outside $[-1, 1]$ is clipped. But that applies to homogenized coordinates. Recall that a coordinate is homogenized by dividing each component by its fourth (w). So to determine if a clip space coordinate is in the volume one might test whether $-1 \leq x/w \leq 1$, $-1 \leq y/w \leq 1$, and $-1 \leq z/w \leq 1$. To avoid division clipping actually does the equivalent test $-w \leq x \leq w$, $-w \leq y \leq w$, and $-w \leq z \leq w$.

Though a vertex either is or isn't in the clip volume, a primitive can be part in and part out. An OpenGL implementation can do all, some, or no clipping in clip space, leaving the clipping for later (where it will be called scissoring).

After clipping coordinates are homogenized (called a **perspective divide** in OpenGL parlance), the resulting coordinates are said to be in **normalized device coordinates**. A final **viewport transformation** converts the coordinates to **window space**.

Rasterization is performed in window space. Each fragment faces a series of tests it must pass to be written to the frame buffer, one of which is the z-buffer test. (The others aren't covered here.)