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

It is useful to note that most operations in the application/scene stage and the early geometry stage of the pipeline are done per vertex, whereas culling and clipping is done per triangle, and rendering operations are done per pixel.

1. Application/Scene

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

2. Geometry

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

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

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

4. Rendering

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

In some sense, 3D chips have become physical incarnations of the pipeline, where data flows “downstream” from stage to stage. It is useful to note that most operations in the application/scene stage and the early geometry stage of the pipeline are done per vertex, whereas culling and clipping is done per triangle, and rendering operations are done per pixel. Computations in various stages of the pipeline can be overlapped, for improved performance. For example, because vertices and pixels are mutually independent of one another in both Direct3D and OpenGL, one triangle can be in the geometry stage while another is in the Rasterization stage. Furthermore, computations on two or more vertices in the Geometry stage and two or more pixels (from the same triangle) in the Rasterzation phase can be performed at the same time.

Another advantage of pipelining is that because no data is passed from one vertex to another in the geometry stage or from one pixel to another in the rendering stage, chipmakers have been able to implement multiple pixel pipes and gain considerable performance boosts using parallel processing of these independent entities. It’s also useful to note that the use of pipelining for real-time rendering, though it has many advantages, is not without downsides. For instance, once a triangle is sent down the pipeline, the programmer has pretty much waved goodbye to it. To get status or color/alpha information about that vertex once it’s in the pipe is very expensive in terms of performance, and can cause pipeline stalls, a definite no-no.

# Geometry section

## Transforms

Objects get moved from frame to frame to create the illusion of movement, and in a 3D world, objects can be moved or manipulated using four operations broadly referred to as transforms. The transforms are actually performed on object vertices using different types of “transform matrices” via matrix mathematics. Translation, Rotation, Scaling, Skewing.

## View Projection

## Trivial Accept/Reject Culling

Culling (view-frustum) / trivial rejection

If at least one (or two) of a triangle’s vertices has all three of its coordinates (x, y, or z) inside the view volume, that triangle intersects the view volume boundaries somewhere and the portions falling outside the frustum will need to be clipped off later in the pipeline. The remaining portion of the triangle, now forming a non-triangular polygon will need to be subdivided into triangles (called retesselation) within the frustum. These resulting triangles will also need to be clip-tested.

## Back-Face Culling

The next operation is called back-face culling (BFC), which as the name suggests, is an operation that discards triangles that have surfaces that are facing away from the view camera.

Back-face culling is one of those operations that can be done at different points in the pipeline. Some 3D texts, such as Foley-Van Dam, describe BFC being done in view space before lighting operations are done, for the obvious reason that by discarding these triangles, the hardware won’t waste time lighting back-facing triangles that the viewer can’t see. However, other texts, such as Moller-Haines’ *Real-Time Rendering* shows that BFC can be done in either view or screen space. (Moller, T., Haines, E., *Real-Time Rendering* (RTR), (A.K. Peters, Natick, MA, 1999)

Determining whether triangles are back facing depends on the space where the tests are done. In view space, the API looks at each triangle’s normal. This is a vector that is pre-defined when the model is created in a program like 3D Studio Max or Maya, and is perpendicular to the surface of the triangle.

Looking from a straight-line viewing vector projected from the camera (in view space) to the center of a triangle where the normal vector originates, a measure of the angle between the normal vector and the viewing vector can be calculated. The angle is tested to see if it is greater than 90º. If so, the triangle is facing away from the camera, and can be discarded. If it’s less than or equal to 90º, then it’s visible to the view camera and cannot be thrown out. In other words, a check is made to see if the camera is on the same side of the plane of the triangle as the normal vector.

To determine which triangles are back facing in 2D screen space, a similar test can be done, wherein a face normal (a vector that is perpendicular to the surface of the triangle) is calculated for a projected triangle, and the test seeks to determine if this normal points at the camera or away from the camera.

## Lighting

The phrase “Transform and Lighting” has become standard parlance in 3D speak, and we don’t often hear one without the other. But lighting doesn’t happen after all transforms have occurred. It usually happens once the 3D scene has been transformed into view space.

But to better understand their relationship consider this explanation by nVidia’s Chief Scientist Dave Kirk: “lighting is the luminance value, whereas shading is about reflectance and/or transmittance.” These are related to lighting, but shading calculations occur later in the pipeline after rasterization, and we’ll cover the topic later.

## Perspective Divide – Transform to Clip Space

## Clipping

At this point in Geometry processing, models have been animated, and put in their proper places in the scene, triangles outside of the view volume have been discarded through culling, and lighting calculations have been done. Now we move on to “clipping,” which is the operation to discard only the parts of triangles that in some way partially or fully fall outside the view volume.

For clipping to occur with mathematical efficiency, the scene must be transformed from view space into clip space as we described earlier. With the transformation of the frustum into a unit cube, the clipping planes are now orthogonal (perpendicular) to the axes of the space. Recall the perspective divide occurs here, where *x*, *y*, and *z* are divided by *w*, a scale factor that represents distance of the vertex from the view point, so that objects further from the view camera are smaller.

There are multiple approaches to doing clipping, and these step-by-step procedures are called algorithms, and are usually named for their inventors–Liang-Barsky, Cyrus-Beck-Liang-Barsky, Nichol-Lee-Nichol and Sutherland-Hodgeman to name several.

Taking Sutherland-Hodgeman as an example, this algorithm can work in either 2D (four clipping boundaries) or in 3D (six clipping boundaries–left, right, top, bottom, near, far). It works by examining a triangle one boundary at a time, and in some sense is a “multi-pass” algorithm, in that it initially clips against the first clip boundary. It then takes the newly clipped polygon and compares it to the next clip boundary, re-clips if necessary, and ultimately does six “passes” in order to clip a triangle against the six sides of the 3D unit cube.

Once a triangle has been clipped, it must be retesselated, or made into a new set of triangles. Take for example the diagram where you see a single triangle being clipped whose remaining visible portion now forms a quadrilateral. The clipper must now determine the intersection points of each side of the triangle with that clipping boundary, and then draws new triangles that will be part of the final scene.

Another clipping method uses a “guard band” to further reduce the number of triangles that need to be clipped. To achieve this, two frusta are created–a guard band frustum and a screen frustum. A triangle that goes out of the screen frustum but stays within the larger guard band frustum doesn’t need to be clipped. Triangles are trivially rejected from the screen frustum, trivially accepted within the guard band frustum, and failing both of those conditions, the triangle is clipped against the screen frustum.

## Transform to Screen Space

Clip coordinates, are also referred to as *normalized device coordinates* (NDC), and after clipping operations have concluded, these coordinates are now mapped to the screen by transforming into screen space. Although *z* and *w* values are retained for depth buffering tests, screen space is essentially a 2D coordinate system, so only *x* and *y* coordinates need to be mapped to screen resolution. To map NDCs with a range of -1 to 1 to screen space, the following formula is generally used:

# Triangle Setup

Here’s another place where the semantics sometimes get a bit murky. Some define the rasterization process as including triangle setup, whereas others view triangle setup as a separate step that precedes the rasterization stage of the pipeline. We’re going to treat it in the latter manner. Notice that once again you can have overlapping operations here (as we’ll see in a second) but for the sake of clarity, we’re going to handle triangle setup separately.

We should also clarify two terms that sometimes get used interchangeably, and probably shouldn’t: rasterization and rendering. Rasterization is a generic term that describes the conversion from a two-dimensional or three-dimensional vector representation to an x-y coordinate representation.

In the case of graphics cards (or printers), this process turns an image into points of color. A rasterizer can be a 3D card, or it can be a printer for that matter. But the process of rasterization is to assign color values to a given number of points to render the image, be it on a screen or a piece of paper. Here’s where it gets murky. Rendering is the series of operations that determine the final pixel color displayed for a frame of animation, and while this can be thought of broadly as rasterization, the actual conversion of the 3D scene into screen-addressed pixels, or “real” rasterization, happens during triangle setup, also called scan-line conversion.

Think of triangle setup as the prelude to the rendering stage of the pipeline, because it “sets the table” for the rendering operations that will follow.

First off, the triangle setup operation computes the slope (or steepness) of a triangle edge using vertex information at each of edge’s two endpoints. You may recall the equation of a straight line being y=mx+b, where y is the y-axis value, x is the x-axis value, b is the value of y when x=0 (the “y intercept”), and m is the slope (or the ratio of the rate of change between x and y values).

The slope is often called delta x/delta y, dx/dy, Dx/Dy, or literally change in x/change in y). Using the slope information, an algorithm called a digital differential analyzer (DDA) can calculate x,y values to see which pixels each triangle side (line segment) touches. The process operates horizontal scan line by horizontal scan line. The DDA figures out the x-value of the pixels touched by a given triangle side in each successive scan-line. (Watt, p. 143)

What it really does is determine how much the *x* value of the pixel touched by a given triangle side changes per scan line, and increments it by that value on each subsequent scan-line.

To actually calculate the y value of the triangle edge for a given integer value of x, as we move incrementally along the x axis one pixel at a time, we use the slope value. For every single pixel increment along the x-axis, we must increment the y-axis value of the triangle edge by Dy, which is equal to the slope m when x is incremented by one pixel.

Note that each scan line is the next incremental y coordinate in screen space. The y values of non-vertex points on the triangle edge are approximated by the DDA algorithm, and are non-integer floating-point values that typically fall between two integer y values (scan lines). The algorithm finds the nearest y value (scan line number) to assign to y.

This can be seen in the stair-step “jaggie” effect along edges that 3D systems try to reduce using higher resolution display or anti-aliasing techniques that we’ll describe soon. Ultimately, the result of the DDA operation is that we now have x,y values for all scan line crossing points of each line segment in a triangle.

Taking this logic a step further, since nearly all 3D scenes are triangle-based, the rasterization operations are triangle-based as well. In carrying out its processing, the DDA generated the left- and right-hand edges of a triangle’s intersection with a given scan-line. The portion of a scan line that bridges the two triangle edges is called a span (Watt, p. 143). Looking at the scan-line diagram will help this become clearer. We’ll see how the span is used in the rendering section below.

A few other things happen here during the triangle setup phase worth noting. Specifically, color and depth values are interpolated for each pixel. Up until this point, only vertices have had color and depth information, but now that the triangle edge pixels are being created, interpolated color and depth values must also be calculated for those pixels.

These values are interpolated using a weighted average of the color and depth values of the edge’s vertex values, where the color and depth data of edge pixels closer to a given vertex more closely approximate values for that vertex. In addition, the texture coordinates are calculated for use during texture mapping. Recall that the texture addresses define which part of the texture is needed on a specific part of a model–like a wood texture for a tree trunk–and are “pre-baked” into the model’s data at each vertex. Similar to color and depth values, the texture coordinates are interpolated as well.

Now here’s where we can get into some potential overlapping operations. In addition to interpolating the color and depth values, shading operations (which we cover next) can also be done during this operation as well.

# Rendering / Rasterization

## Shading

The rasterizer receives the two pixel endpoints per each scan line that a triangle covers from the setup stage, and calculates the shading values for each of the end pixels. Recall the span between these two pixel endpoints per scan-line described earlier. The rasterizer will shade the span based on various shading algorithms. These shading calculations can range in their demand from fairly modest (Flat and Gouraud), to much more demanding (Phong).

‘Shading’ is one of those terms that sometimes seems like a semantic football, as noted earlier, Dave Kirk, Chief Scientist at nVidia describes it this way: “Lighting is the luminance value, whereas shading is about reflectance or transmittance.” The three most common shading methods, flat, Gouraud, and Phong operate per triangle, per vertex, and per pixel, respectively.

* **Flat Shading**: The simplest of the three models, here the renderer takes the color values from a triangle’s three vertices (assuming triangles as primitive), and averages those values (or in the case of Direct3D, picks an arbitrary one of the three). The average value is then used to shade the entire triangle. This method is very inexpensive in terms of computations, but this method’s visual cost is that individual triangles are clearly visible, and it disrupts the illusion of creating a single surface out of multiple triangles. (Lathrop, O., *The Way Computer Graphics Works*, Wiley Computer Publishing, New York, 1997)
* **Gouraud Shading**: Named after its inventor, Henri Gouraud who developed this technique in 1971 (yes, 1971). It is by far the most common type of shading used in consumer 3D graphics hardware, primarily because of its higher visual quality versus its still-modest computational demands. This technique takes the lighting values at each of a triangle’s three vertices, then interpolates those values across the surface of the triangle (RTR, p. 68). Gouraud shading actually first interpolates between vertices and assigns values along triangle edges, then it interpolates across the scan line based on the interpolated edge crossing values. One of the main advantages to Gouraud is that it smoothes out triangle edges on mesh surfaces, giving objects a more realistic appearance. The disadvantage to Gouraud is that its overall effect suffers on lower triangle-count models, because with fewer vertices, shading detail (specifically peaks and valleys in the intensity) is lost. Additionally, Gouraud shading sometimes loses highlight detail, and fails to capture spotlight effects, and sometimes produces what’s called Mach banding (that looks like stripes at the edges of the triangles)(RTR, p. 69).
* **Phong Shading**: Also named after its inventor, Phong Biu-Tuong, who published a paper on this technique in 1975. This technique uses shading normals, which are different from geometric normals (see the diagram). Phong shading uses these shading normals, which are stored at each vertex, to interpolate the shading normal at each pixel in the triangle (RTR, p. 68). Recall that a normal defines a vector (which has direction and magnitude (length), but not location). But unlike a surface normal that is perpendicular to a triangle’s surface, a shading normal (also called a vertex normal) actually is an average of the surface normals of its surrounding triangles. Phong shading essentially performs Gouraud lighting at each pixel (instead of at just the three vertices).  
  And similar to the Gouraud shading method of interpolating, Phong shading first interpolates normals along triangle edges, and then interpolates normals across all pixels in a scan line based on the interpolated edge values.

More recently, another per-pixel lighting model has come onto the scene using a technique called dot product texture blending, or DOT3, which debuted in the DirectX 6 version of Direct3D. A prelude to programmable shaders, this technique gains the benefit of higher resolution per-pixel lighting without introducing the overhead of interpolating across an entire triangle. This approach is somewhat similar to Phong shading, but rather than calculating interpolated shading normals for every pixel on the fly, DOT3 instead uses a normal map that contains “canned” per-pixel normal information. Think of a normal map as a kind of texture map. Using this normal map, the renderer can do a lookup of the normals to then calculate the lighting value per pixel.

Once the lighting value has been calculated, it is recombined with the original texel color value using a modulate (multiply) operation to produce the final lit, colored, textured pixel. Essentially, DOT3 combines the efficiencies of light maps, wherein you gain an advantage having expensive-to-calculate information (in the case of DOT3 per-pixel normals) “pre-baked” into a normal map rather than having to calculate them on the fly, with the more realistic lighting effect of Phong shading. the per pixel interpolators are used to interpolate the Phong normals across the triangle and DOT3 operations and texture lookups are used to compute the Phong lighting equation at each pixel.

## Texturing

## Fog

## Alpha- Test and Blending

## Shadows

## AntiAliasing

## Depth Buffering

Z-buffer, W-buffer

## Dithering

## Blending

## Display