# Formative Assessment
# Variance in Graphics APIs

Specialized APIs (application programming interfaces) have been created to ease the processes in all stages of computer graphics generation, primarily where 3D graphics are concerned. These APIs have also proved vital to computer graphics hardware manufacturers, as they provide a way for programmers to access the hardware in an abstract way, while still taking advantage of the special hardware of any specific graphics card.

The first 3D graphics framework was possibly Core, published by the ACM in 1977.

The following list details all low-level 3D API's that I could find, with a short description on each. However, there are many other API's used for 3D graphics, including web-based API's and high-level 3D API's. I will be focusing on OpenGL, DirectX and Vulkan, as these are API's I am currently using or am planning to use.

## Low-Level 3D API

### DirectX

Microsoft DirectX is a collection of application programming interfaces (APIs) for handling tasks related to multimedia, especially game programming and video, on Microsoft platforms. Originally, the names of these APIs all began with Direct, such as Direct3D, DirectDraw, DirectMusic, DirectPlay, DirectSound, and so forth. The name DirectX was coined as a shorthand term for all of these APIs (the X standing in for the particular API names) and soon became the name of the collection. When Microsoft later set out to develop a gaming console, the X was used as the basis of the name Xbox to indicate that the console was based on DirectX technology. The Xinitial has been carried forward in the naming of APIs designed for the Xbox such as XInput and the Cross-platform Audio Creation Tool(XACT), while the DirectX pattern has been continued for Windows APIs such as Direct2D and DirectWrite.

### Direct3D (a subset of DirectX)

Direct3D is a graphics application programming interface (API) for Microsoft Windows. Part of DirectX, Direct3D is used to render three-dimensional graphics in applications where performance is important, such as games. Direct3D uses hardware acceleration if it is available on the graphics card, allowing for hardware acceleration of the entire 3D rendering pipeline or even only partial acceleration. Direct3D exposes the advanced graphics capabilities of 3D graphics hardware, including Z-buffering, W-buffering, stencil buffering, spatial anti-aliasing, alpha blending, color blending, mipmapping, texture blending, clipping, culling, atmospheric effects, perspective-correct texture mapping, programmable HLSL shaders and effects. Integration with other DirectX technologies enables Direct3D to deliver such features as video mapping, hardware 3D rendering in 2D overlay planes, and even sprites, providing the use of 2D and 3D graphics in interactive media ties.
The Direct3D API specifically uses Microsoft's own High-level Shader Language (HLSL).

### Glide

Glide is a 3D graphics API developed by 3dfx Interactive for their Voodoo Graphics 3D accelerator cards. Although it originally started as a proprietary API, it was later open sourced by 3dfx. It was dedicated to rendering performance, supporting geometry and texture mapping primarily, in data formats identical to those used internally in their cards. Wide adoption of 3Dfx led to Glide being extensively used in the late 1990s, but further refinement of Microsoft's Direct3D and the appearance of full OpenGL implementations from other graphics card vendors, in addition to growing diversity in 3D hardware, eventually caused it to become superfluous.

## Mantle developed by AMD

In computing, Mantle is a low-overhead rendering API targeted at 3D video games. AMD originally developed Mantle in cooperation with DICE, starting in 2013. Mantle was designed as an alternative to Direct3D and OpenGL, primarily for use on personal computers, although Mantle supports the GPUs present in the PlayStation 4 and in the Xbox One. According to AMD, Mantle will make a shift in focus after March 2015 to other areas since DirectX 12 and the Mantle-derived Vulkan API are largely replacing it in the gaming industry. Therefore, Mantle will be supported by AMD in neither the short term nor long term future, effectively making all future games unable to implement Mantle, and making Mantle unable to be implemented by any older games.

## Metal developed by Apple

Metal is a low-level, low-overhead hardware-accelerated graphics and compute application programming interface (API) that debuted in iOS 8. It combines functionality similar to OpenGL and OpenCL under one API. It is intended to bring to iOS, macOS, and tvOS some of the performance benefits of similar APIs on other platforms, such as Khronos Group's cross-platform Vulkan (which debuted in mid-February 2016) and Microsoft's Direct3D 12 for Windows.

## OpenGL

3D API for embedded devices.

Open Graphics Library (OpenGL) is a cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics. The API is typically used to interact with a graphics processing unit (GPU), to achieve hardware-accelerated rendering.

Silicon Graphics Inc., (SGI) started developing OpenGL in 1991 and released it in January 1992; applications use it extensively in the fields of computer-aided design (CAD), virtual reality, scientific visualization, information visualization, flight simulation, and video games. Since 2006 OpenGL has been managed by the non-profit technology consortium Khronos Group.

## OpenGL ES

OpenGL for Embedded Systems (OpenGL ES or GLES) is a subset of the OpenGL computer graphics rendering application programming interface (API) for rendering 2D and 3D computer graphics such as those used by video games, typically hardware-accelerated using a graphics processing unit (GPU). It is designed for embedded systems like smartphones, tablet computers, video game consoles and PDAs. OpenGL ES is the "most widely deployed 3D graphics API in history".

The API is cross-language and multi-platform. The libraries GLUT and GLU are not available for OpenGL ES. OpenGL ES is managed by the non-profit technology consortium Khronos Group. Vulkan, a next-generation API from Khronos, is made for simpler high performance drivers for mobile and desktop devices.

## QuickDraw 3D

QuickDraw 3D developed by Apple Computer starting in 1995, abandoned in 1998

QD3D was separated into two layers. A lower level system known as RAVE (Rendering Acceleration Virtual Engine) provided a hardware abstraction layer with functionality similar to Direct3D or cut-down versions of OpenGL like MiniGL. On top of this was an object-oriented scene graph system, QD3D proper, which handled model loading and manipulation at a level similar to OpenGL++ (depricated).

Additional functionality included a "plug-in" rendering system, which allowed an application to render a scene in a variety of styles. Without changing the model or their code, developers could render the same scene interactively or (with suitable plug-ins) using methods such as ray-tracing (a rendering technique for generating an image by tracing the path of light as pixels in an image plane and simulating the effects of its encounters with virtual objects) or phong shading(Phong shading refers to an interpolation technique for surface shading in 3D computer graphics. It is also called Phong interpolation or normal-vector interpolation shading).

## RenderMan

Pixar RenderMan (formerly PhotoRealistic RenderMan), is proprietary photorealistic 3D rendering software produced by Pixar Animation Studios. Pixar uses RenderMan to render their in-house 3D animated movie productions and it is also available as a commercial product licensed to third parties. RenderMan defines cameras, geometry, materials, and lights using the RenderMan Interface Specification. This specification facilitates communication between 3D modeling and animation applications and the render engine that generates high quality images. Additionally RenderMan supports Open Shading Language to define textural patterns. Historically, RenderMan used the Reyes algorithm to render images with added support for advanced effects such as ray tracing and global illumination. Support for Reyes rendering and the RenderMan Shading Language were removed from RenderMan in 2016.

RenderMan currently uses Monte Carlo path tracing to generate images.

## RenderWare

RenderWare is a 3D API and graphics rendering engine used in video games, Active Worlds, and some VRML browsers. RW was developed by Criterion Software Limited (which used to be a wholly owned subsidiary of Canon but is now owned by Electronic Arts). It originated in the era of software rendering on PCs prior to the appearance of GPUs, competing with other libraries such as Argonaut's BRender and RenderMorphics' Reality Lab (the latter was acquired by Microsoft and became Direct3D).

## Vulkan

Vulkan is a low-overhead, cross-platform 3D graphics and compute API. Vulkan targets high-performance realtime 3D graphics applications such as video games and interactive media across all platforms. Compared with OpenGL and Direct3D 11, and like Direct3D 12 and Metal, Vulkan is intended to offer higher performance and more balanced CPU/GPU usage. Other major differences from Direct3D 11 (and prior) and OpenGL are Vulkan being a considerably lower level API and offering parallel tasking. Vulkan also has the ability to render 2D graphics applications, however it is generally suited for 3D. In addition to its lower CPU usage, Vulkan is also able to better distribute work amongst multiple CPU cores.
Vulkan is intended to provide a variety of advantages over other APIs as well as its spiritual predecessor, OpenGL. Vulkan offers lower overhead, more direct control over the GPU, and lower CPU usage. The overall concept and feature set of Vulkan is similar to Direct3D 12, Metal and Mantle.

# Shading Languages

## HLSL

The High-Level Shader Language or High-Level Shading Language (HLSL) is a proprietary shading language developed by Microsoft for the Direct3D 9 API to augment the shader assembly language, and went on to become the required shading language for the unified shader model of Direct3D 10 and higher.

HLSL is analogous to the GLSL shading language used with the OpenGL standard. It is very similar to the Nvidia Cg shading language, as it was developed alongside it. HLSL shaders can enable profound speed and detail increases as well as many special effects in both 2d and 3d computer graphics.

### Example Vertex Shader

```
VertexShaderOutput VertexShaderFunction(VertexShaderInput input)
{
    VertexShaderOutput output;

    float4 worldPosition = mul(input.Position, World);
    float4 viewPosition = mul(worldPosition, View);
    output.Position = mul(viewPosition, Projection);

    return output;
}
```

### Example Fragment Shader

```
float4 PixelShaderFunction(VertexShaderOutput input) : COLOR0

{
    return AmbientColor * AmbientIntensity;
}
```

*To view more example shaders and all the different shader types available in HLSL we can go here: http://rbwhitaker.wikidot.com/hlsl-tutorials*

# GLSL

OpenGL Shading Language (abbreviated: GLSL or GLslang), is a high-level shading language with a syntax based on the C programming language. It was created by the OpenGL ARB (OpenGL Architecture Review Board) to give developers more direct control of the graphics pipeline without having to use ARB assembly language or hardware-specific languages.

### Example Vertex Shader

```
void main(void)
{
   vec4 a = gl_Vertex;
   a.x = a.x * 0.5;
   a.y = a.y * 0.5;


   gl_Position = gl_ModelViewProjectionMatrix * a;

}
```

### Example Fragment Shader

```
void main (void)
{
   gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0);
}
```

*For more examples, we can look at: http://glslsandbox.com/*

*And for more information on the different shaders available we can look at: https://www.khronos.org/opengl/wiki/Category:OpenGL_Shading_Language*

# RSL

An example of the old RenderMan Shading Language (RSL):

```
surface metal(float Ka = 1; float Ks = 1; float roughness = 0.1;)
{
  normal Nf = faceforward(normalize(N), I);
  vector V = - normalize(I);
  Oi = Os;
  Ci = Os * Cs * (Ka * ambient() + Ks * specular(Nf, V, roughness));
}
```

## CG

Cg or C for Graphics is a high level shading language created by NVIDIA to simplify vertex and fragment shader programming. Though it now supports geometry and tessellation shaders as well. Note, that Cg has been discontinued and NVIDIA doesn't recommend to use it for new projects.

Although Cg shares many syntactical similarities with C/C++, some features were modified to accommodate the inherent differences between CPU programming and GPU programming.

### Example Vertex Shader

```
struct VertIn {
    float4 pos   : POSITION;
    float4 color : COLOR0;
};

// output vertex
struct VertOut {
    float4 pos   : POSITION;
    float4 color : COLOR0;
};

// vertex shader main entry
VertOut main(VertIn IN, uniform float4x4 modelViewProj) {
    VertOut OUT;
    OUT.pos     = mul(modelViewProj, IN.pos); // calculate output coords
    OUT.color   = IN.color; // copy input color to output
    OUT.color.z = 1.0f; // blue component of color = 1.0f
    return OUT;
}
```

# Rendering Techniques

## Architectural rendering

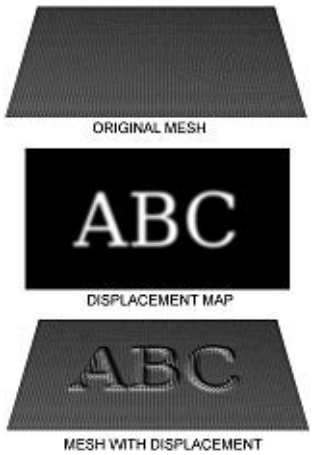
Architectural rendering, or architectural illustration, is the art of creating two-dimensional images or animations showing the attributes of a proposed architectural design.

## Chromatic aberration

Chromatic aberration (abbreviated CA; also called chromatic distortion and spherochromatism) is an effect resulting from dispersion in which there is a failure of a lens to focus all colors to the same convergence point. It occurs because lenses have different refractive indices for different wavelengths of light. The refractive index of transparent materials decreases with increasing wavelength in degrees unique to each

## Displacement mapping

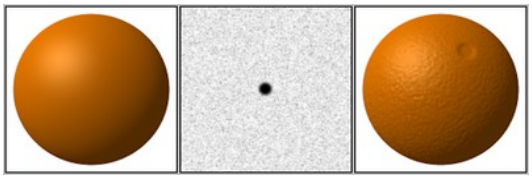
Displacement mapping is an alternative computer graphics technique in contrast to bump mapping, normal mapping, and parallax mapping, using a (procedural-) texture- or height map to cause an effect where the actual geometric position of points over the textured surface are displaced, often along the local surface normal, according to the value the texture function evaluates to at each point on the surface. It gives surfaces a great sense of depth and detail, permitting in particular self-occlusion, self-shadowing and silhouettes; on the other hand, it is the most costly of this class of techniques owing to the large amount of additional geometry.



## Bump Mapping

Bump mapping is a technique in computer graphics for simulating bumps and wrinkles on the surface of an object. This is achieved by perturbing the surface normals of the object and using the perturbed normal during lighting calculations. The result is an apparently bumpy surface rather than a smooth surface although the surface of the underlying object is not changed. Bump mapping was introduced by James Blinn in 1978.

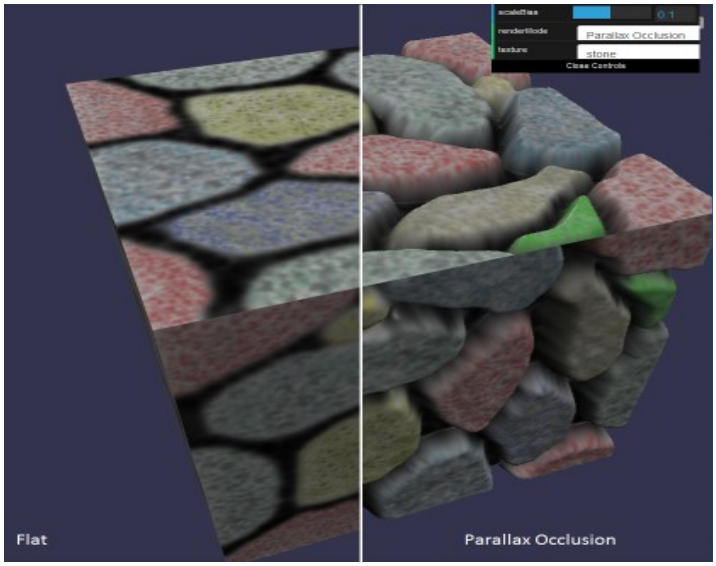
Normal mapping is the most common variation of bump mapping used.



## Parallax Mapping

Parallax mapping (also called offset mapping or virtual displacement mapping) is an enhancement of the bump mapping or normal mapping techniques applied to textures in 3D rendering applications such as video games. To the end user, this means that textures such as stone walls will have more apparent depth and thus greater realism with less of an influence on the performance of the simulation.

Parallax mapping is implemented by displacing the texture coordinates at a point on the rendered polygon by a function of the view angle in tangent space (the angle relative to the surface normal) and the value of the height map at that point. At steeper view-angles, the texture coordinates are displaced more, giving the illusion of depth due to parallax effects as the view changes.

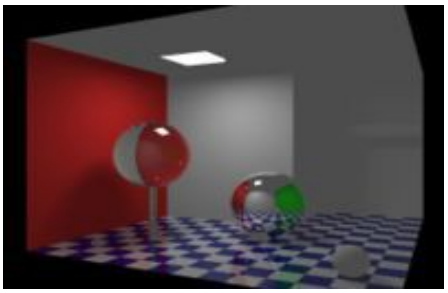


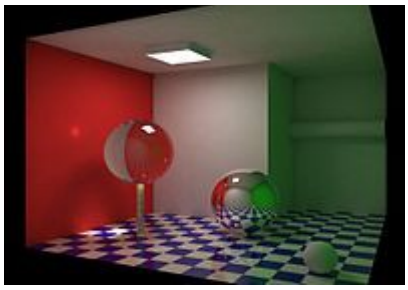
## Self-Shadowing

Self-Shadowing is a computer graphics lighting effect, used in 3D rendering applications such as computer animation and video games. Self-shadowing allows non-static objects in the environment, such as game characters and interactive objects (buckets, chairs, etc.), to cast shadows on themselves and each other. For example, without self-shadowing, if a character puts his or her right arm over the left, the right arm will not cast a shadow over the left arm. If that same character places a hand over a ball, that hand will cast a shadow over the ball.

## Global illumination

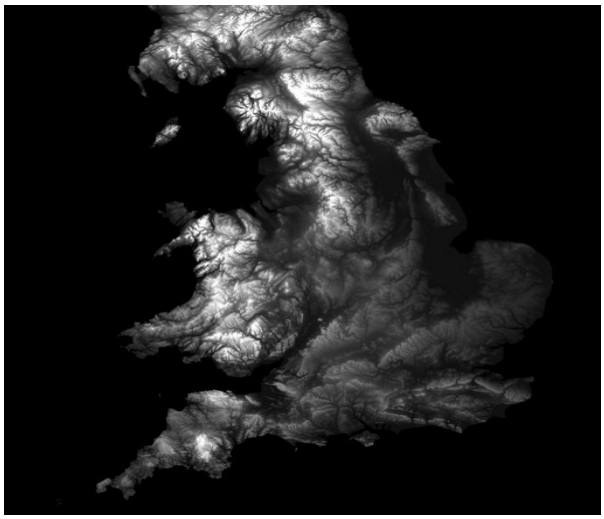
Global illumination (shortened as GI), or indirect illumination, is a general name for a group of algorithms used in 3D computer graphics that are meant to add more realistic lighting to 3D scenes. Such algorithms take into account not only the light that comes directly from a light source (direct illumination), but also subsequent cases in which light rays from the same source are reflected by other surfaces in the scene, whether reflective or not (indirect illumination).





## Heightmap

A heightmap or heightfield is a raster image used to store values, such as surface elevation data, for display in 3D computer graphics. A heightmap can be used in bump mapping to calculate where this 3D data would create shadow in a material, in displacement mapping to displace the actual geometric position of points over the textured surface, or for terrain where the heightmap is converted into a 3D mesh.



## High dynamic range rendering

High-dynamic-range rendering (HDRR or HDR rendering), also known as high-dynamic-range lighting, is the rendering of computer graphics scenes by using lighting calculations done in high dynamic range (HDR). This allows preservation of details that may be lost due to limiting contrast ratios. Video games and computer-generated movies and special effects benefit from this as it creates more realistic scenes than with the more simplistic lighting models used.

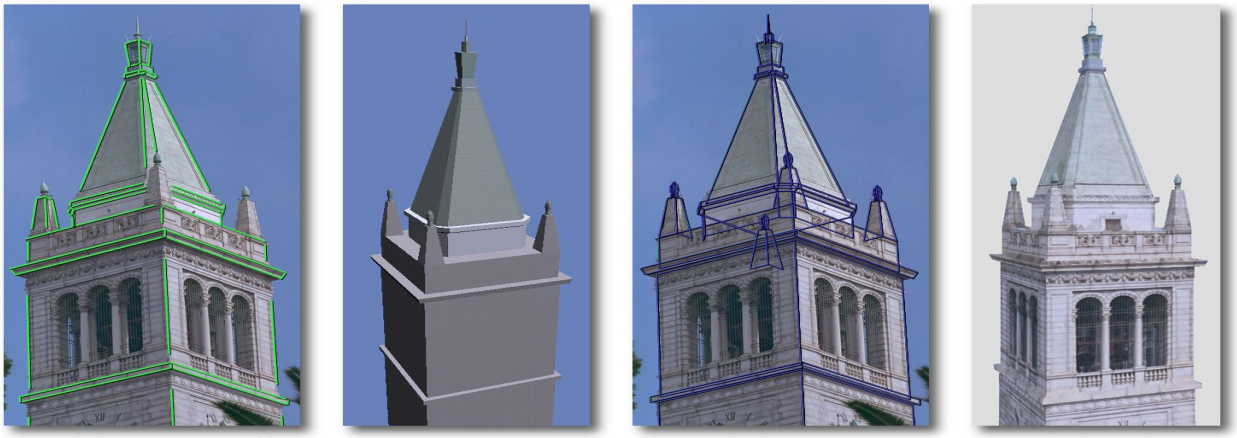
Graphics processor company Nvidia summarizes the motivation for HDR in three points: bright things can be really bright, dark things can be really dark, and details can be seen in both

## Image-based modelling and rendering

Image-based modelling and rendering (IBMR) methods rely on a set of two-dimensional images of a scene to generate a three-dimensional model and then render some novel views of this scene.

Modeling and Rendering Architecture from Photographs

Debevec, Taylor, and Malik 1996



Original photograph with marked edges | Recovered model | Model edges projected onto photograph | Synthetic rendering

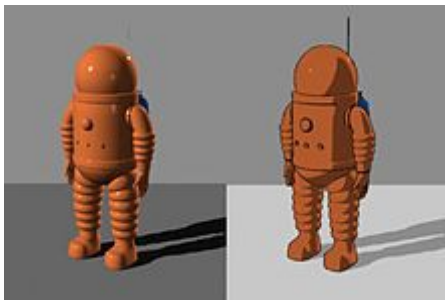
## Motion blur

Motion blur is the apparent streaking of rapidly moving objects in a still image or a sequence of images such as a movie or animation. It results when the image being recorded changes during the recording of a single exposure, either due to rapid movement or long exposure.

## Non-photorealistic rendering

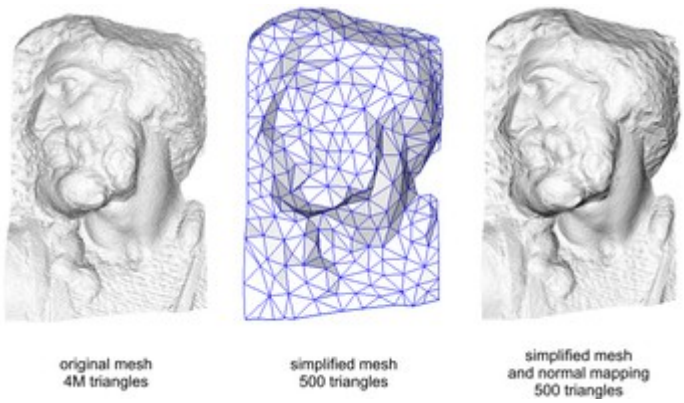
Non-photorealistic rendering (NPR) is an area of computer graphics that focuses on enabling a wide variety of expressive styles for digital art. In contrast to traditional computer graphics, which has focused on photorealism, NPR is inspired by artistic styles such as painting, drawing, technical illustration, and animated cartoons. NPR has appeared in movies and video games in the form of "toon shading", as well as in scientific visualization, architectural illustration and experimental animation. An example of a modern use of this method is that of cel-shaded animation.

Cel shading or toon shading is a type of non-photorealistic rendering designed to make 3-D computer graphics appear to be flat by using less shading color instead of a shade gradient or tints and shades. Cel-shading is often used to mimic the style of a comic book or cartoon and/or give it a characteristic paper-like texture. There are similar techniques that can make an image look like a sketch, an oil painting or an ink painting. It is somewhat recent, appearing from around the beginning of the twenty-first century. The name comes from cels (short for celluloid), the clear sheets of acetate, which are painted on for use in traditional 2D animation.



## Normal mapping

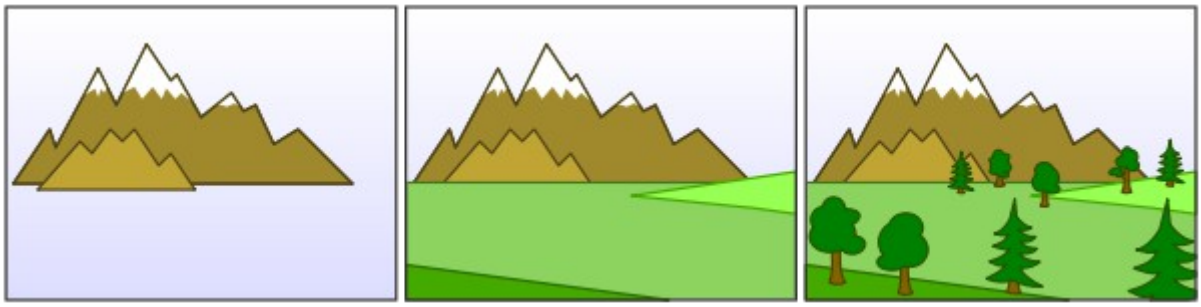
normal mapping, or Dot3 bump mapping, is a technique used for faking the lighting of bumps and dents – an implementation of bump mapping. It is used to add details without using more polygons. A common use of this technique is to greatly enhance the appearance and details of a low polygon model by generating a normal map from a high polygon model or height map.



## •Painter's algorithm

The painter's algorithm, also known as a priority fill, is one of the simplest solutions to the visibility problem in 3D computer graphics. When projecting a 3D scene onto a 2D plane, it is necessary at some point to decide which polygons are visible, and which are hidden.

The distant mountains are painted first, followed by the closer meadows; finally, the trees, are painted. Although some trees are more distant from the viewpoint than some parts of the meadows, the ordering (mountains, meadows, trees) forms a valid depth order, because no object in the ordering obscures any part of a later object.



## Physically based rendering

Physically based rendering or PBR is a model in computer graphics that seeks to render graphics in a way that more accurately models the flow of light in the real world. Many PBR pipelines (though not all) have the accurate simulation of photorealism as their goal, often in real time computing.

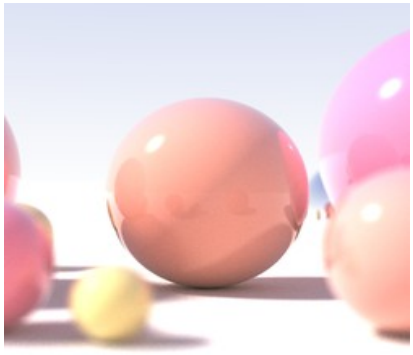
PBR is often characterized by - but not necessarily limited to - an approximation of a real, radiometric bidirectional reflectance distribution function to govern the essential reflections of light, the use of reflection constants such as specular intensity, gloss, and metallicity derived from measurements of real-world sources, accurate modelling of global illumination in which light bounces and/or is emitted from objects other than the primary light sources, conservation of energy which balances the intensity of specular highlights with dark areas of an object, Fresnel conditions that reflect light at the sides of objects perpendicular to the viewer, and accurate modelling of roughness resulting from micro-surfaces.

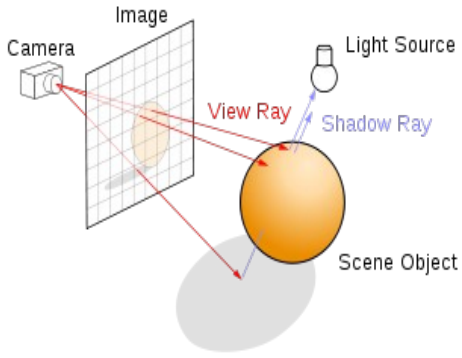
## Radiosity

Radiosity is an application of the finite element method to solving the rendering equation for scenes with surfaces that reflect light diffusely. Unlike rendering methods that use Monte Carlo algorithms (such as path tracing), which handle all types of light paths, typical radiosity only account for paths (represented by the code "LD*E") which leave a light source and are reflected diffusely some number of times (possibly zero) before hitting the eye. Radiosity is a global illumination algorithm in the sense that the illumination arriving on a surface comes not just directly from the light sources, but also from other surfaces reflecting light. Radiosity is viewpoint independent, which increases the calculations involved, but makes them useful for all viewpoints.

## Ray tracing

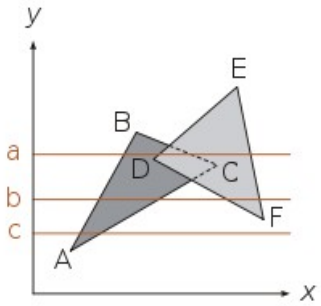
Ray tracing is a rendering technique for generating an image by tracing the path of light as pixels in an image plane and simulating the effects of its encounters with virtual objects. The technique is capable of producing a very high degree of visual realism, usually higher than that of typical scanline rendering methods, but at a greater computational cost. This makes ray tracing best suited for applications where the image can be rendered slowly ahead of time, such as in still images and film and television visual effects, and more poorly suited for real-time applications like video games where speed is critical. Ray tracing is capable of simulating a wide variety of optical effects, such as reflection and refraction, scattering, and dispersion phenomena (such as chromatic aberration)





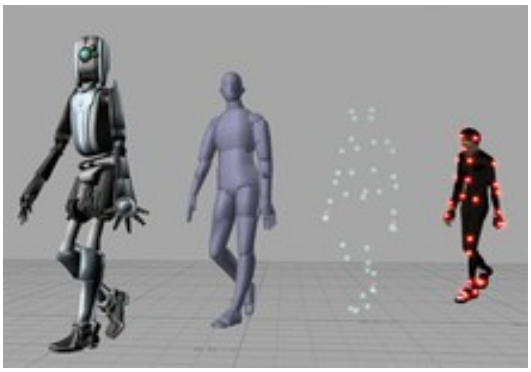
## Scanline rendering/Scanline algorithm

Scanline rendering (also scan line rendering and scan-line rendering) is an algorithm for visible surface determination, in 3D computer graphics, that works on a row-by-row basis rather than a polygon-by-polygon or pixel-by-pixel basis. All of the polygons to be rendered are first sorted by the top y coordinate at which they first appear, then each row or scan line of the image is computed using the intersection of a scanline with the polygons on the front of the sorted list, while the sorted list is updated to discard no-longer-visible polygons as the active scan line is advanced down the picture.




## Virtual model

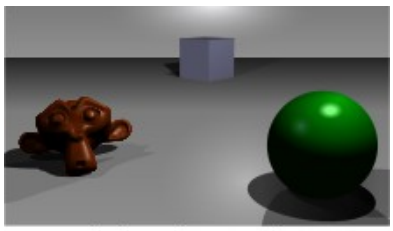
3D modeling (or three-dimensional modeling) is the process of developing a mathematical representation of any surface of an object (either inanimate or living) in three dimensions via specialized software. The product is called a 3D model. Someone who works with 3D models may be referred to as a 3D artist. It can be displayed as a two-dimensional image through a process called 3D rendering or used in a computer simulation of physical phenomena. The model can also be physically created using 3D printing devices.



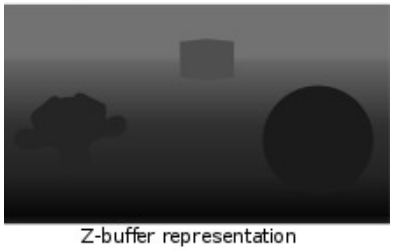
## Z-buffer algorithms

z-buffering, also known as depth buffering, is the management of image depth coordinates in 3D graphics, usually done in hardware, sometimes in software. It is one solution to the visibility problem, which is the problem of deciding which elements of a rendered scene are visible, and which are hidden. The painter's algorithm is another common solution which, though less efficient, can also handle non-opaque scene elements.

When an object is rendered, the depth of a generated pixel (z coordinate) is stored in a buffer (the z-buffer or depth buffer). This buffer is usually arranged as a two-dimensional array (x-y) with one element for each screen pixel. If another object of the scene must be rendered in the same pixel, the method compares the two depths and overrides the current pixel if the object is closer to the observer. The chosen depth is then saved to the z-buffer, replacing the old one. In the end, the z-buffer will allow the method to correctly reproduce the usual depth perception: a close object hides a farther one. This is called z-culling.



A simple three-dimensional scene



Z-buffer representation

## Texture Mapping

Texture mapping is a method for defining high frequency detail, surface texture, or color information on a computer-generated graphicor 3D model.
Texture mapping originally referred to a method (now more accurately called diffuse mapping) that simply wrapped and mapped pixels from a texture to a 3D surface. In recent decades the advent of multi-pass rendering and complex mapping such as height mapping, bump mapping, normal mapping, displacement mapping, reflection mapping, specular mapping, mipmaps, occlusion mapping, and many other variations on the technique (controlled by a materials system) have made it possible to simulate near-photorealism in real time by vastly reducing the number of polygons and lighting calculations needed to construct a realistic and functional 3D scene.

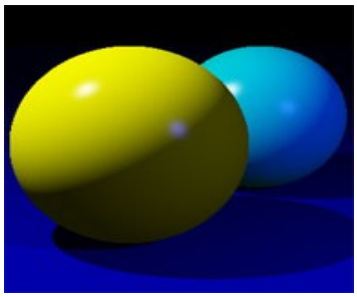
## Reflection Mapping

Environment mapping, or reflection mapping, is an efficient image-based lighting technique for approximating the appearance of a reflective surface by means of a precomputed texture image. The texture is used to store the image of the distant environment surrounding the rendered object.

## Specular Mapping

The quantity used in three-dimensional (3D) rendering which represents the amount of reflectivity a surface has. It is a key component in determining the brightness of specular highlights, along with shininess to determine the size of the highlights.

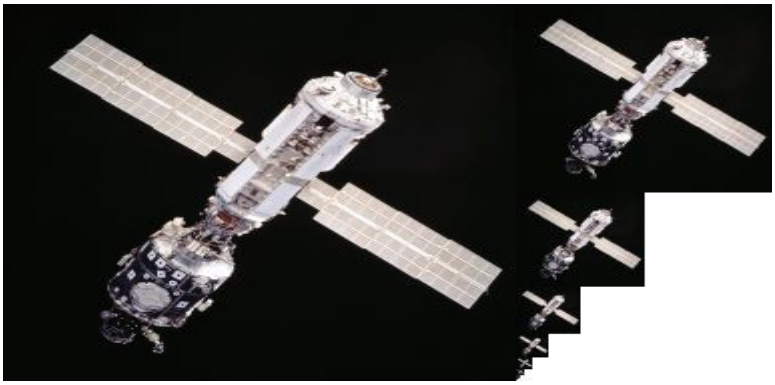
It is frequently used in real-time computer graphics and ray tracing, where the mirror-like specular reflection of light from other surfaces is often ignored (due to the more intensive computations required to calculate it), and the specular reflection of light directly from point light sources is modelled as specular highlights.



## MipMaps

The quantity used in three-dimensional (3D) rendering which represents the amount of reflectivity a surface has. It is a key component in determining the brightness of specular highlights, along with shininess to determine the size of the highlights.

It is frequently used in real-time computer graphics and ray tracing, where the mirror-like specular reflection of light from other surfaces is often ignored (due to the more intensive computations required to calculate it), and the specular reflection of light directly from point light sources is modelled as specular highlights.



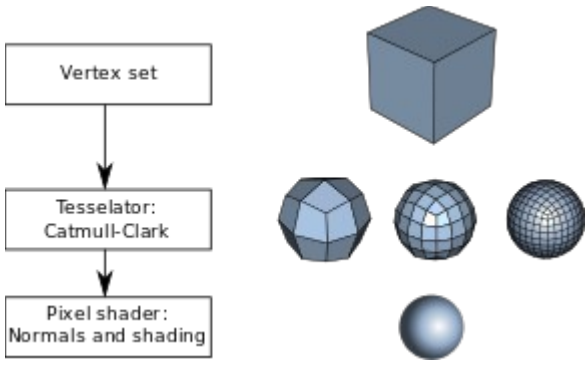
## Ambient Occlusion

Ambient occlusion is a shading and rendering technique used to calculate how exposed each point in a scene is to ambient lighting. For example, the interior of a tube is typically more occluded (and hence darker) than the exposed outer surfaces, and the deeper you go inside the tube, the more occluded (and darker) the lighting becomes.



## Tessellation

Tessellation is used to manage datasets of polygons (sometimes called vertex sets) presenting objects in a scene and divide them into suitable structures for rendering. Especially for real-time rendering, data is tessellated into triangles, for example in OpenGL 4.0 and Direct3D 11.

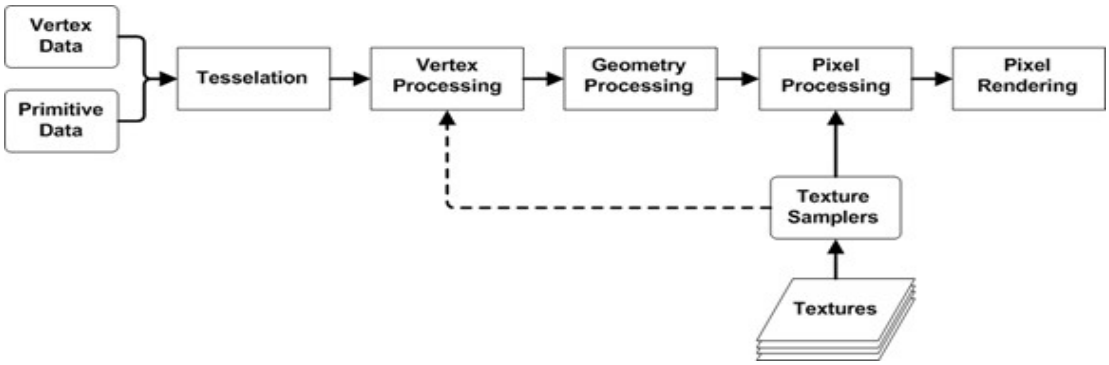



# SHADERS

Shading refers to depicting depth perception in 3D models or illustrations by varying levels of darkness.

Shading refers to the process of altering the color of an object/surface/polygon in the 3D scene, based on things like (but not limited to) the surface's angle to lights, its distance from lights, its angle to the camera and material properties (e.g. bidirectional reflectance distribution function) to create a photorealistic effect. Shading is performed during the rendering process by a program called a shader.

Shading is also dependent on the lighting used. Usually, upon rendering a scene a number of different lighting techniques will be used to make the rendering look more realistic. Different types of light sources are used to give different effects.



## Diffuse Lighting

Diffuse lighting is the light that reflects off of an object in all directions (it diffuses). It is what gives most objects their color.

## Ambient lighting

An ambient light source represents an omni-directional, fixed-intensity and fixed-color light source that affects all objects in the scene equally. Upon rendering, all objects in the scene are brightened with the specified intensity and color. This type of light source is mainly used to provide the scene with a basic view of the different objects in it. This is the simplest type of lighting to implement and models how light can be scattered or reflected many times producing a uniform effect.

Ambient lighting can be combined with ambient occlusion to represent how exposed each point of the scene is, affecting the amount of ambient light it can reflect. This produces diffuse, non-directional lighting throughout the scene, casting no clear shadows, but with enclosed and sheltered areas darkened. The result is usually visually similar to an overcast day.

## Directional lighting

A directional light source illuminates all objects equally from a given direction, like an area light of infinite size and infinite distance from the scene; there is shading, but cannot be any distance falloff.

## Point lighting

Light originates from a single point, and spreads outward in all directions.

## Spotlight lighting

Models a Spotlight. Light originates from a single point, and spreads outward in a cone.

## Area lighting

Light originates from a small area on a single plane. A more realistic model than a point light source.

## Volumetric lighting

Light originating from a small volume, an enclosed space lighting objects within that space.

Shading is interpolated based on how the angle of these light sources reach the objects within a scene. Of course, these light sources can be and often are combined in a scene.
The renderer then interpolates how these lights must be combined, and produces a 2d image to be displayed on the screen accordingly.

## Distance falloff

Theoretically, two surfaces which are parallel are illuminated the same amount from a distant light source, such as the sun. Even though one surface is further away, your eye sees more of it in the same space, so the illumination appears the same.

The left image doesn't use distance falloff. Notice that the colors on the front faces of the two boxes are exactly the same. It appears that there is a slight difference where the two faces meet, but this is an optical illusion caused by the vertical edge below where the two faces meet.

The right image uses distance falloff. Notice that the front face of the front box is brighter than the front face of the back box. Also, the floor goes from light to dark as it gets farther away.

This distance falloff effect produces images which appear more realistic.

## Interpolation techniques

When calculating the brightness of a surface during rendering, our illumination model requires that we know the surface normal. However, a 3D model is usually described by a polygon mesh, which may only store the surface normal at a limited number of points, usually either in the vertices, in the polygon faces, or in both. To get around this problem, one of a number of interpolation techniques can be used.

## Flat shading

Here, a color is calculated for one point on each polygon (usually for the first vertex in the polygon, but sometimes for the centroid for triangle meshes), based on the polygon's surface normal and on the assumption that all polygons are flat. The color everywhere else is then interpolated by coloring all points on a polygon the same as the point for which the color was calculated, giving each polygon a uniform color (similar to in nearest-neighbor interpolation). It is usually used for high speed rendering where more advanced shading techniques are too computationally expensive. As a result of flat shading all of the polygon's vertices are colored with one color, allowing differentiation between adjacent polygons. Specular highlights are rendered poorly with flat shading: If there happens to be a large specular component at the representative vertex, that brightness is drawn uniformly over the entire face. If a specular highlight doesn’t fall on the representative point, it is missed entirely. Consequently, the specular reflection component is usually not included in flat shading computation.

## Smooth shading

In contrast to flat shading where the colors change discontinuously at polygon borders, with smooth shading the color changes from pixel to pixel, resulting in a smooth color transition between two adjacent polygons. Usually, values are first calculated in the vertices and bilinear interpolation is then used to calculate the values of pixels between the vertices of the polygons.

Types of smooth shading include:

Gouraud shading

Phong shading

## Gouraud shading

1.Determine the normal at each polygon vertex.

2.Apply an illumination model to each vertex to calculate the light intensity from the vertex normal.

3.Interpolate the vertex intensities using bilinear interpolation over the surface polygon.

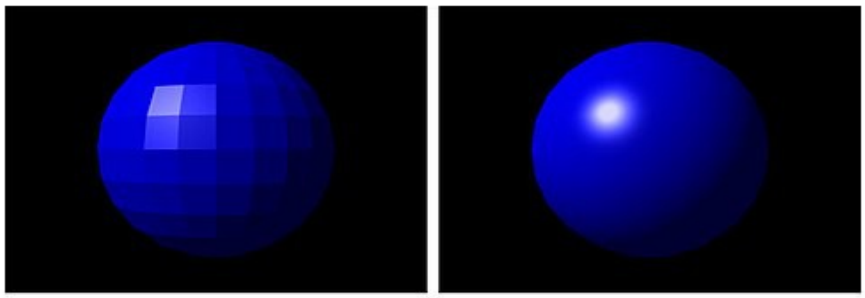
## Phong shading

Phong shading is similar to Gouraud shading, except that instead of interpolating the light intensities, the normals are interpolated between the vertices. Thus, the specular highlights are computed much more precisely than in the Gouraud shading model:

1.Compute a normal N for each vertex of the polygon.

2.From bilinear interpolation compute a normal, Ni, for each pixel. (This must be renormalized each time.)

3.Apply an illumination model to each pixel to calculate the light intensity from Ni.



FLAT SHADING PHONG SHADING

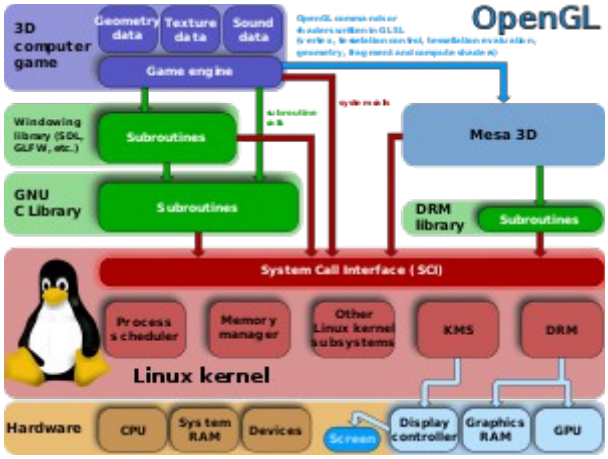
## Other Approaches

Both Gouraud shading and Phong shading can be implemented using bilinear interpolation. Bishop and Weimer proposed to use a Taylor series expansion of the resulting expression from applying an illumination model and bilinear interpolation of the normals. Hence, second degree polynomial interpolation was used. This type of biquadratic interpolation was further elaborated by Barrera et al., where one second order polynomial was used to interpolate the diffuse light of the Phong reflection model and another second order polynomial was used for the specular light.

Spherical Linear Interpolation (Slerp) was used by Kuij and Blake for computing both the normal over the polygon as well as the vector in the direction to the light source. A similar approach was proposed by Hast, which uses Quaternion interpolation of the normals with the advantage that the normal will always have unit length and the computationally heavy normalization is avoided.

# OpenGL and the OpenGL Shading Language

**OpenGL Current version: 4.6**
**GLSL Current version: 4.60**




Open Graphics Library (OpenGL) is a cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics. The API is typically used to interact with a graphics processing unit (GPU), to achieve hardware-accelerated rendering.

Silicon Graphics Inc., (SGI) started developing OpenGL in 1991 and released it in January 1992; applications use it extensively in the fields of computer-aided design (CAD), virtual reality, scientific visualization, information visualization, flight simulation, and video games. Since 2006 OpenGL has been managed by the non-profit technology consortium Khronos Group

The OpenGL specification describes an abstract API for drawing 2D and 3D graphics. Although it is possible for the API to be implemented entirely in software, it is designed to be implemented mostly or entirely in hardware.

The API is defined as a set of functions which may be called by the client program, alongside a set of named integer constants (for example, the constant GL_TEXTURE_2D, which corresponds to the decimal number 3553). Although the function definitions are superficially similar to those of the programming language C, they are language-independent. As such, OpenGL has many language bindings, some of the most noteworthy being the JavaScriptbinding WebGL (API, based on OpenGL ES 2.0, for 3D rendering from within a web browser); the C bindings WGL, GLX and CGL; the C binding provided by iOS; and the Java and C bindings provided by Android.

In addition to being language-independent, OpenGL is also cross-platform. The specification says nothing on the subject of obtaining, and managing an OpenGL context, leaving this as a detail of the underlying windowing system. For the same reason, OpenGL is purely concerned with rendering, providing no APIs related to input, audio, or windowing.

The OpenGL Shading Language (GLSL) is the principal shading language for OpenGL. While, thanks to OpenGL Extensions, there are several shading languages available for use in OpenGL, GLSL (and SPIR-V) are supported directly by OpenGL without extensions.

GLSL is a C-style language. The language has undergone a number of version changes, and it shares the deprecation model of OpenGL. The current version of GLSL is 4.60.

GLSL is a lot like C/C++ in many ways. It supports most of the familiar structural components (for-loops, if-statements, etc). But it has some important language differences.

# DirectX and the High-level Shading Language

**Direxts X current version: DirectX 12**
**HLSL current version: HLSL Shader Model 6.0**

Microsoft DirectX is a collection of application programming interfaces (APIs) for handling tasks related to multimedia, especially game programming and video, on Microsoft platforms. Originally, the names of these APIs all began with Direct, such as Direct3D, DirectDraw, DirectMusic, DirectPlay, DirectSound, and so forth. The name DirectX was coined as a shorthand term for all of these APIs (the X standing in for the particular API names) and soon became the name of the collection. When Microsoft later set out to develop a gaming console, the X was used as the basis of the name Xbox to indicate that the console was based on DirectX technology. The Xinitial has been carried forward in the naming of APIs designed for the Xbox such as XInput and the Cross-platform Audio Creation Tool(XACT), while the DirectX pattern has been continued for Windows APIs such as Direct2D and DirectWrite.

Direct3D (the 3D graphics API within DirectX) is widely used in the development of video games for Microsoft Windows and the Xboxline of consoles. Direct3D is also used by other software applications for visualization and graphics tasks such as CAD/CAM engineering. As Direct3D is the most widely publicized component of DirectX, it is common to see the names "DirectX" and "Direct3D" used interchangeably.

The DirectX software development kit (SDK) consists of runtime libraries in redistributable binary form, along with accompanying documentation and headers for use in coding. Originally, the runtimes were only installed by games or explicitly by the user. Windows 95 did not launch with DirectX, but DirectX was included with Windows 95 OEM Service Release 2. Windows 98 and Windows NT 4.0both shipped with DirectX, as has every version of Windows released since. The SDK is available as a free download. While the runtimes are proprietary, closed-source software, source code is provided for most of the SDK samples. Starting with the release of Windows 8 Developer Preview, DirectX SDK has been integrated into Windows SDK.

HLSL, which is short for High Level Shader Language, is the programming language that we will use. It is pretty much the standard language, when you are using DirectX. (OpenGL uses a language called GLSL, which is similar, but different.) HLSL is fairly similar in structure to the programming language C. If you have done much with C or C++, this is probably a relief to you. Even C# and Java are somewhat similar to it. But if you are still a little worried about HLSL, don't be. We will take it fairly slowly, and we will go through it from the ground up.

# Vulkan

Vulkan, formerly named the "Next Generation OpenGL Initiative" (glNext), is a grounds-up redesign effort to unify OpenGL and OpenGL ES into one common API that will not be backwards compatible with existing OpenGL versions.

The initial version of Vulkan API was released on 16 February 2016. Eventually, Vulkan API will replace OpenGL entirely.

Intended advantages of Vulkan over previous-generation APIs include:

Vulkan API is well suited for high-end graphics cards as well as for graphics hardware on mobile devices (OpenGL has a specific subset for mobile devices called OpenGL ES; it's still an alternative API in Vulkan supporting devices).

In contrast to Direct3D 12, Vulkan is available on multiple modern operating systems; like OpenGL, the Vulkan API is not locked to a single OS or device form factor. As of release, Vulkan runs on Android, Linux, Tizen, Windows 7, Windows 8, and Windows 10 (third party support for iOS and macOS is also available)

Reduced driver overhead, reducing CPU workloads.

Reduced load on CPUs through the use of batching, leaving the CPU free to do more computation or rendering than otherwise.

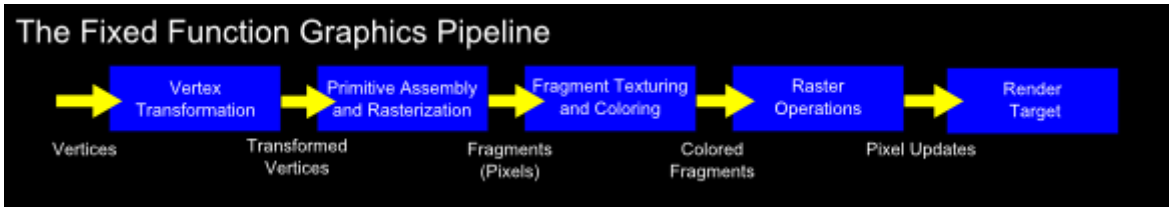
Better scaling on multi-core CPUs. Direct3D 11 and OpenGL 4 were initially designed for use with single-core CPUs and only received augmentation to be executed on multi-cores. Even when application developers use the augmentations, the API regularly does not scale well on multi-cores.

OpenGL uses the high-level language GLSL for writing shaders which forces each OpenGL driver to implement its own compiler for GLSL that executes at application runtime to translate the program's shaders into the GPU's machine code. Vulkan drivers are supposed to ingest instead shaders already translated into an intermediate binary format called SPIR-V (Standard Portable Intermediate Representation), analogous to the binary format that HLSL shaders are compiled into in Direct3D. By allowing shader pre-compilation, application initialization speed is improved and a larger variety of shaders can be used per scene. A Vulkan driver only needs to do GPU specific optimization and code generation, resulting in easier driver maintenance, and eventually smaller driver packages (currently GPU vendors still have to include OpenGL/CL).

Unified management of compute kernels and graphical shaders, eliminating the need to use a separate compute API in conjunction with a graphics API.

## The History of API's
## The Programmable Graphics Pipeline

This pipeline has the responsibility of taking commands to draw a group of vertices (usually from your 3D model) with all of the correct texturing, lighting, and positioning. The original fixed function pipeline looked something like the image below:




On the left, vertices are passed into the pipeline. At this point, the vertices are in "model coordinates", meaning that the locations of each vertex are relative to the model. In fact, these vertices all have the exact same locations as they do in the model file. Our first step is to transform these vertices to the right location on the screen, which is usually determined by the usual world, view, and projection matrices. This transformation is done in the Vertex Transformation step. The result of this is vertices that are located in the correct location (transformed vertices). During this step, each triangle (or other primitive) is handled separately.

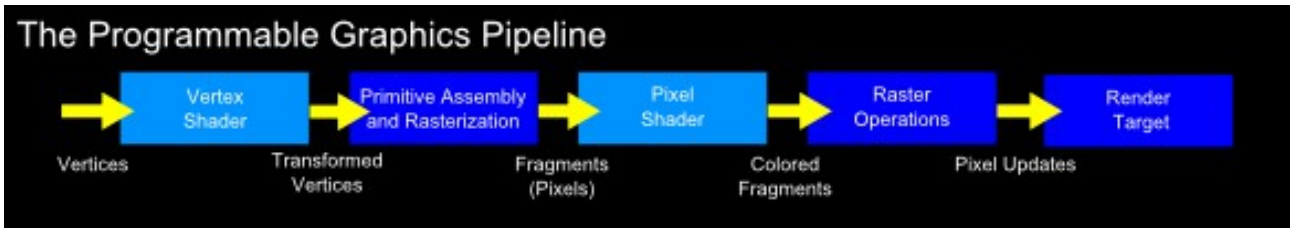
The next step is to put the pieces back together and begin rasterization, which is the process of taking a geometric shape and converting it into the pixels that will be drawn for it in the end. This is done in the second step, called Primitive Assembly and Rasterization.

After this step, we have what are called "fragments", but which essentially, are pixels with some more information to help us determine (in the next step) how the pixel should be colored. For instance, texturing and coloring information will be passed along with the fragment.

The third step is Fragment Texturing and Coloring, where the exact color of the pixel is determined, based on the texture being used and the color information associated with the fragment. After that, a few other pixel operations are done during the Raster Operations stage,

including possibly fog and alpha blending, and the final value for a pixel is determined and placed in the render target, which is then drawn on the screen.

The programmable graphics pipeline is fairly similar to this, with a couple of differences. The programmable graphics pipeline is shown below:




The major differences come in the first and third steps. Rather than having vertex transformations performed by the graphics library, you get to write your own program to do this. This is called a programmable vertex processor, or Vertex Shader. You will write a simple program that will be compiled and sent to the graphics card, and run for every vertex you draw. The rasterization step is similar to the one in the Fixed Function Pipeline. However, the fragment coloring and texturing is also done with your own program. Once again, you will write a simple program that will be compiled and sent over to the graphics card, and then it will be executed on every pixel that is drawn.

The programmable graphics pipeline allows you to do anything you can dream up during these two steps. In the future, it is likely that more and more of this pipeline will become programmable. In fact, DirectX 10 as support for another type of shader called a geometry shader, which allows you to generate more geometry (more vertices) on the graphics card, which means that you will be able to get away with sending even less information over to the graphics card in some cases.

The big drawback is that now you have to write all of the functions that you want to use, including texturing and lighting calculations. This can be a daunting task for anyone. The XNA framework supplies you with the BasicEffect class, which, hopefully you have used quite a bit by now. In this class, they have implemented all of the basic shading stuff for you, and give you an easy interface to use it with.

In fact, the BasicEffect class ends up feeling fairly similar to the original Fixed Function Pipeline. The BasicEffect class is quite powerful, and it is easy to use, and for many simpler games, this might be enough. In these HLSL tutorials, though, we are going to go on and create our own shaders. For any sophisticated game, you will likely need to use a big variety of shaders to get the effects that you want. For instance, I browsed through the directory structure for Microsoft Flight Simulator X, and discovered that there was a directory that contained hundreds, if not thousands, of shader files! In the tutorials to come, we will go through the basics of creating and using your own shaders in an XNA game.

## Differences between OpenGL and Direct3D

In general, Direct3D is designed to virtualise 3D hardware interfaces. Direct3D frees the game programmer from accommodating the graphics hardware. OpenGL, on the other hand, is designed to be a 3D hardware-accelerated rendering system that may be emulated in software. These two APIs are fundamentally designed under two separate modes of thought.

As such, there are functional differences in how the two APIs work. One functional difference between the APIs is in how they manage hardware resources. Direct3D expects the application to do it, OpenGL makes the implementation do it. This trade-off for OpenGL decreases difficulty in developing for the API, while at the same time increasing the complexity of creating an implementation (or driver) that performs well. With Direct3D, the developer must manage hardware resources independently; however, the implementation is simpler, and developers have the flexibility to allocate resources in the most efficient way possible for their

application.

Until about 2005, another functional difference between the APIs was the way they handled rendering to textures. The Direct3D method (SetRenderTarget()) is convenient, while prior versions of OpenGL required manipulating pixel buffers (P-buffers). This was cumbersome and risky: if the code-path used in a program was different from that anticipated by a driver maker, the code would fall back to software rendering, causing a substantial performance drop. However, broad support for the frame buffer objects extension, which provided an OpenGL equivalent of the Direct3D method, successfully addressed this shortcoming, and the render target feature of OpenGL brought it up to par with Direct3D in this aspect.

Outside of a few minor functional differences which have mostly been addressed over the years, the two APIs provide nearly the same level of function. Hardware and software makers generally respond rapidly to changes in DirectX, e.g., pixel processor and shader requirements in DirectX 9 to stream processors in DirectX 10, to tessellation in DirectX 11. In contrast, new features in OpenGL are usually implemented first by vendors and then retroactively applied to the standard.