Jarno Laaksonen

OpenGL rendering pipeline

Tradenomi

Tietojenkäsittely

Syksy 2017

KAJAANIN
AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Abstract

Author: Laaksonen Jarno

Title of the Publication: OpenGL rendering pipeline

Degree Title: Bachelor of Business Administration (UAS), Business Information Technology

Keywords: OpenGL, graphics programming, rendering, shader, pipeline, GLSL

The objective of this thesis was done with the intent of learning and studying for the job of a technical artist. The main objective was to find more information about OpenGLs rendering pipeline and how it processes through model data. It covers each step and explains their functions and how they are used by the programmer. The engine used during this thesis was written from scratch and is able load most model data types and implement their rendering with wanted written shaders.

Rendering pipeline refers to a game engines graphical process for reading model position and mesh data and turning it into pixels for the display. The pipeline process is relatively same for all game engines and thus learning one, allows you to learn all of them. For anyone working with the said pipeline and it's shaders, needs to know each step well to figure out how to achieve wanted effects. A lot of these steps are automated and out of programmers control, however knowledge of them is still required in case of indirect problems in the process.

OpenGL proves as an excellent learning tool and gives a great coverage of different applications for visual effects. This combined with community tutorials and multiple published books on the usage of it, it will make anyone a professional at using the graphical engine side of gameplay development.

Tiivistelmä

Tekijä: Laaksonen Jarno

Työn nimi: OpenGL renderöinti prosessi

Tutkintonimike: Tradenomi (AMK), tietojenkäsittely

Asiasanat: OpenGL, grafiikka ohjelmointi, renderöinti, varjostin, putkisto, GLSL

Tämä opinnäytteen aihe oli valittu aikomuksella opiskella tekniseksi artistiksi. Opinnäytteen tavoite on etsiä ja oppia lisää OpenGL:n renderöinti prosessista ja kuinka jokainen vaihe toimii yleisessä käytössä. Opinnäyte käy läpi jokaisen vaiheen renderöinnistä ja selittää kuinka haluttu malli data käydään läpi ja muunnetaan pikseleiksi. Tässä projektissa käytetty moottori on itsekirjoitettu ja pystyy lataamaan ja käyttämään visuaalista dataa oikein renderöimällä ne monitorille

Renderöinti prosessi viittaa pelimoottorien graafiseen toiminnallisuuteen jossa se lukee mallin sijainnin ja visuaalisen datan ja muuntamaan sen pikseleiksi ruudulle. Tämä prosessi on relatiivisesti sama kaikilla moottoreilla ja siten yhden oppiminen antaa ohjelmoijalle tietoa kaikista. Kuka tahansa joka työskentelee renderöinnin ja varjostimien kanssa, täytyy tietää jokainen vaihe läpikotaisin, jotta pystyy tuottamaan halutun efektin. Suuri osa prosessista on automaattisia, mutta niiden tunteminen on silti tärkeää siltä varalta, että jotain epäsuorasti menee pieleen.

OpenGL osoittaa olevansa hyvä alusta oppimiselle ja antaa suuren määrän erilaisia sovelluksia visuaalisille efekteille. Tämän lisäksi OpenGL:n yhteisö tarjoaa kaiken kattavia kursseja ja julkaistuja kirjoja sen käytöstä, voi tehdä kenestä vain ammattilaisen graafisen puolen pelikehityksestä.

Contents

LIST OF SYMBOLS

GPU: Graphics processing unit, the hardware in computers that is optimized for handling graphical algorithms.

Vertex: Singular point on an object, like a corner of a triangle or the point where two edges meet.

Normal: Directional vector that points away from the vertex

Variable: A labelled value used to store information for future reference and manipulation by a program

Program: Set of instructions that are used to control the behaviour of a computer.

Language: Specialised set of instructions for a computer that are used to create programs.

Struct: A data construct that contains multiple variables within itself

API: Application programming interface, a collection of tools, functions and subroutines used for building software.

1 Introduction

A rendering pipeline is the most important part of many game engines, each frame, all mesh data and shaders must be processed to create the images for the game itself: an efficient pipeline means a well optimised game. There are a few well known rendering pipeline APIs: DirectX from Microsoft, and Vulkan and OpenGL from the Khronos organisation.

OpenGL is an API that is compatible with most operating systems that allows the user to take data from files and render them onto a screen. It takes care of unneeded things like window control and setting up a functioning program depending on the operating system and allows the programmers to concentrate on what to render instead of how to render. The OpenGL APIs pipeline and how the data is processed at each stage of the pipeline will be the main subject discussed in this thesis, however much of the information can be applied to other APIs as well. The idea of this thesis is to summarise and give examples of the workings of each stage of the pipeline and give the reader a good idea of how the system works as a whole. The first part of the thesis will cover some of the basic concepts of a rendering pipeline. After they have been examined, the second chapter will explain each part of the pipeline, and what it does to the input data, as well as giving some examples of what the programmer can create. I started this thesis with the objective of learning more about the graphics side of game development and become a more well-rounded programmer, especially concerning graphical meshes and shaders.

2 Khronos and OpenGL

The Khronos Group was founded in 2000 to provide a structure for key industry players to cooperate within, for the creation of open standards that deliver on the promise of cross-platform technology [4]. Since then they have developed multiple royalty-free APIs for usage of dynamic media on various platforms. In 2006 the company was given the rights to continue development of OpenGL onwards from version 2.0 and since then have collaborated with multiple multimillion dollar media organizations that use their interfaces.

2.1 OpenGL

OpenGL (Open Graphics Library) is an openly distributed API for computer graphics. It is commonly used in the game development industry, and is seen as one of the two leading APIs used in rendering, alongside Direct3D. OpenGL is designed with the single purpose of helping programmers to render images, and is mainly used on the Windows operating system. OpenGL has gone through multiple iterations and is currently (as of December 2017) on version 4.6, and during its development it many new modern features and functions have been added, to help programmers optimise their graphics engines.

2.2 Vulkan

Vulkan is the next generation of OpenGL, designed to give programmers more direct control over graphics hardware and processing, allowing for better optimisation. While OpenGL allowed programmers to use it to render amazing scenery and create awe inspiring effects, the optimisation tools it gave to the creators were relatively limited and parts of the API such as memory management and error checking were not available for the user. With Vulkan, more comprehensive tools, from error checking to better hardware control, are given to the engineers and programmers, allowing them to make much more efficient graphical rendering.

3 Important aspects of rendering

There are some important aspects of rendering that are essential for the engine to properly translate 3D data into 2D screen imagery. Some of these are the building blocks of the engine itself, while others are methods that are absolutely required to create a working rendering engine.

3.1 Primitives

According to the OpenGL graphics system specification, primitives are defined by a group of one or more vertices [5, p. 5]. A vertex defines a point, an endpoint of an edge, or a corner of a polygon where two edges meet [5, p. 5]. OpenGL has three types of primitives: points, lines, and triangles. By the end of the rendering pipeline, the data pipeline receives will have been transformed into one of these types. OpenGL supports 12 different kinds of primitives, but almost all of them are derived from the previously mentioned primitives (these can be seen in image 1). The other primitive types that are supported by OpenGL include patches (which are used as inputs to the tessellator) and adjacency primitives (which are used as inputs to the geometry shader) [4. p. 86].

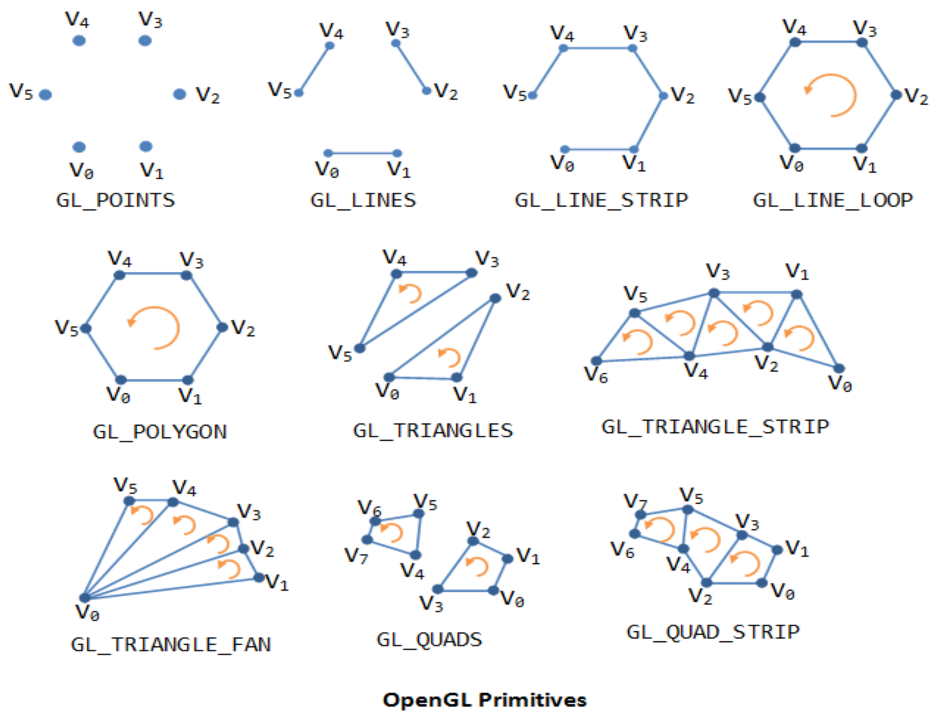


Image 1: The ten types of primitive OpenGL supports

Point, line and triangle are the three primitives that are supported by most rendering engines, as they are the most basic primitives of them all. A point is a singular vertex of a model, so a single model may have thousands or millions of points, if textures are mapped onto a point, it is referred to as a “point sprite”.

A line is a segment between two vertices, each one representing an endpoint. Multiple lines can also be connected together in two ways: a line strip is a collection of lines where the sequence of lines is not closed, while in a line loop the first and last vertex of the line sequence are also connected by a line (see Image 1).

The last of the basic primitives is triangle which is a collection of three vertices and three lines that form the outline of the triangle. These triangles are rarely by themselves and need to be connected, this is where the pipeline can use triangle strips to use vertices more efficiently: by allowing neighbouring triangles to share their vertices, the program can save space and resources (if two separate triangles use six points, putting their edges together uses two fewer points). A triangle fan is another primitive, similar to triangle strips, in which multiple triangles share a singular vertex, allowing additional triangles to be formed from only two extra vertices, again allowing the program to save space and resources.

As seen in the previous shown image, there are also quads as a form of primitive. However, since Khronos group obtained the development rights for OpenGL, this primitive was deprecated and is only used in older versions.

3.2 Shaders

There are two important components in the computer hardware that work together to render modern video games, the CPU (central processing unit) and the GPU graphics processing unit). If normal programming with code is what CPU handles, the shaders are what is handled by the GPU. A programmable shader is a way for developers to write custom algorithms that can operate on the data that composes their virtual scenes [4, p. 172].

3.3 Buffers

In programming, a buffer is a specialised block of memory. It’s most commonly used to refer to a “vehicle” that transfers data from one place to another, in this case, from the CPU to the GPU or vice versa. Buffers are extremely important in graphics programming, because data transfer is how a program gets object data from the CPU to the GPU for processing. This is essentially done by arranging all the integer values of the vertices, normals and texture coordinates into a list and telling the buffer which part is which so the data can be processed correctly.

Many different kinds of buffers are used within the OpenGL pipeline, but one important example used in rendering is “double buffering”. Double buffering is a method in which the engine uses two buffers to avoid technical issues from appearing on the screen, such as screen tearing as shown in image 2. It does this by using two buffers, for a single frame, one of the buffers data is used to display the image while the other buffer has the image data of the next frame rendered into it, in this way the “active” buffer then alternates between the two buffers. The method is important because the slowness of the rendering process may not be able to keep up with the games framerate and cause incomplete images to be displayed.

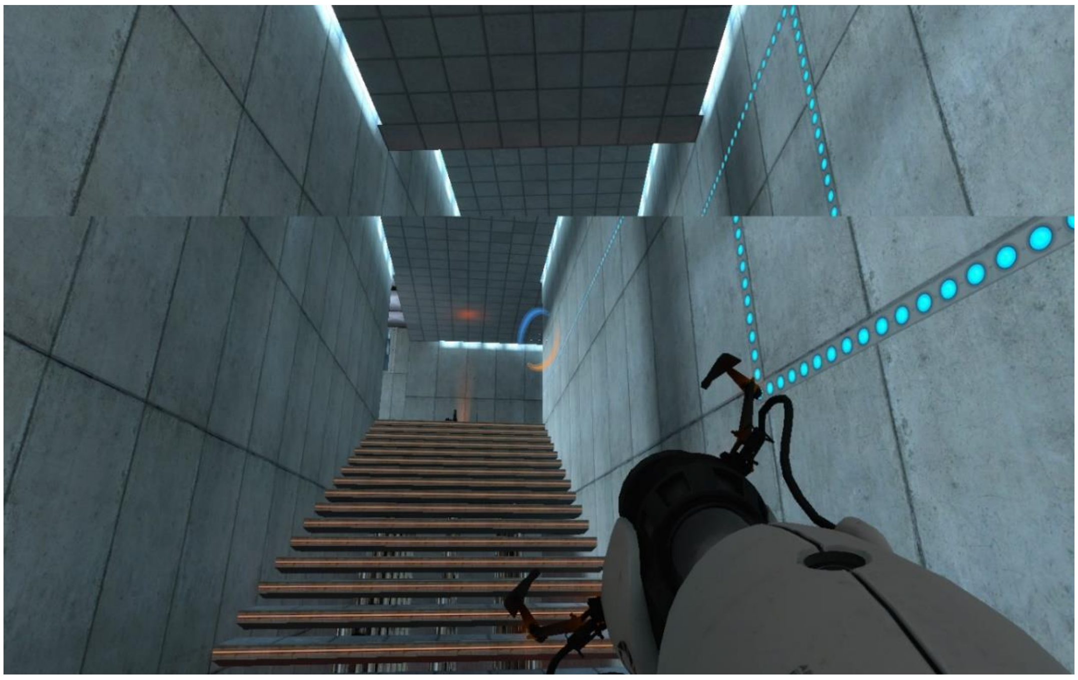


Image 2: Screen tearing in the game Portal.

Other important buffers include the framebuffer, which is a buffer holding inside it the final textures the program uses to render the image, and the depth buffer, which includes the distance of each pixel to the camera.

4 OpenGL Rendering Pipeline

4.1 Vertex specification

The first process in the OpenGL pipeline is setting up incoming object data for further processing through the pipeline. For an object to be rendered, it needs data that describes how it should be visually represented within the virtual world, and for this, the program uses an array of vertices. The vertex array can be quite big, where each tiny point on the model is one vertex, and each vertex is represented by three numbers in the array. The CPU is told to load and compile the model's data from a file and assign it to the created array. After creating the array, program will send it to the pipeline where it starts being processed by this step.

There are two kinds of objects that are used for vertex specification, Vertex Array Objects (VAO) and Vertex Buffer Objects (VBO). The two objects work together to store and define the vertex data of an object, VAO are used to define the data, and VBO are used to store it. A third object type can also be used, the Element Array Buffer (EBO) that is used to reduce reusing vertices in same position, by indexing only the important arrays (see Image 3).

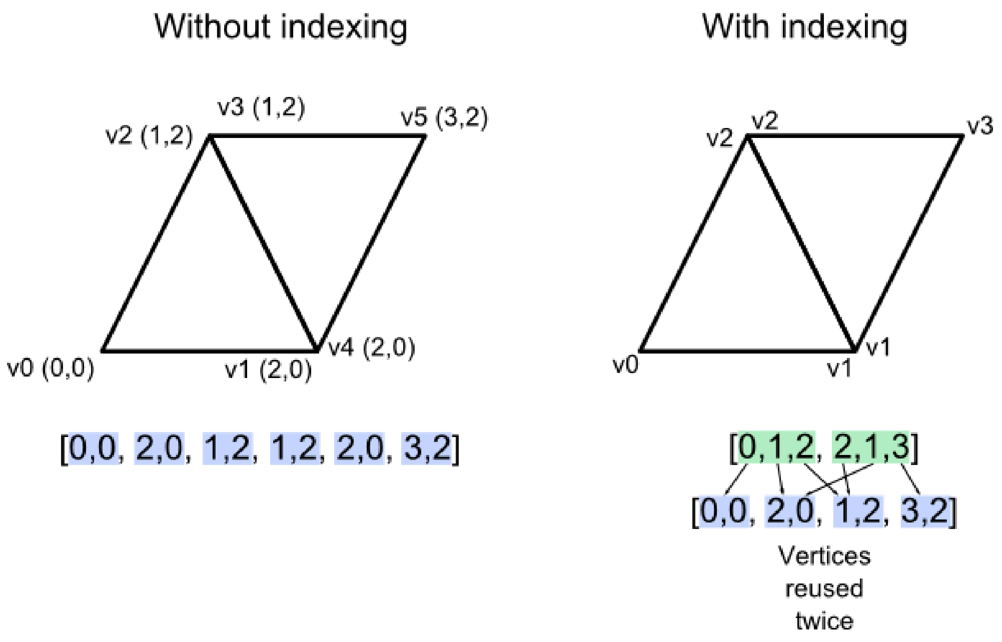


Image 3: Drawing two triangles with and without EBO.

The point of this first step is to take the information input by the programmer: positions, normals, texture locations and other data, and arrange it into sections that the shaders can read. VAO then tells the pipeline what sections in the buffer represent what kind of information and divides them into their own variables. For example if the first 3 values are the position vector and then next 3 represent the normals, VAO know which are which.

4.2 Vertex shader

The vertex shader is the first step on the GPUs side of the pipeline. All vertex data pipeline receives after Vertex specification is handled here, one by one and then sent forward to the next step. Vertex shader is used to transform vertex positions in the virtual space and configure the 2D coordinates it uses for screen.

The most common functions vertex shader serves are defining its position within world space with an MVP matrix (Model View Projection) and calculating the normal of the vertex. The MVP process is required to render models accurately in the correct positions, so to do this, the MVP matrix composes of three separate matrices that accurately describe the world space. The first matrix comprises the coordinates of the model in world space, the second, the view matrix describes the rotations of the camera and finally the projection matrix that includes the perspective values like the width of the camera (commonly known as field-of-view, or FOV), as well as the view distance and screen width. The vertex shader can also be used to calculate shading, though it is less accurate than doing so using a/the fragment shader, as it would be calculated per vertex and not per pixel.

Other uses for the vertex shader are mesh transformations and morphing. Mesh transformation is simply moving vertices from their original positions to shape the mesh, a simple example of this would be to move vertices along the normals to make the mesh look bloated. Mesh transformation can also be used in height mapping, in which the shader takes a flat mesh and transforms their Y-axis according to a grayscale image to create bumps, as shown in image 4.

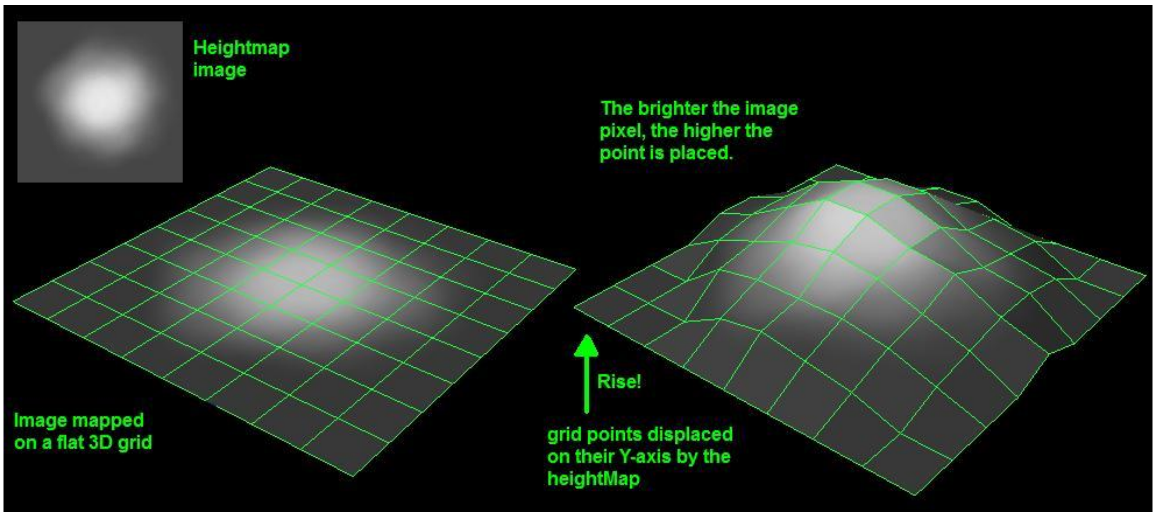

Heightmap image
The brighter the image pixel, the higher the point is placed.
Rise!
Image mapped on a flat 3D grid
grid points displaced on their Y-axis by the heightMap

Image 4: A simple example of height mapping.

4.3 Tessellation

Tessellation is a somewhat recent addition to OpenGL and graphics programming. The main usage of tessellation is to subdivide the edges of primitives, as well as create more vertices inside triangles, lines or quads. This subdivision is done on two levels, the outer and the inner levels. Simply put, the inner tessellation level controls the number of times primitives can “nest”, and the outer tessellation level controls the number of times to subdivide each edge [6]. Tessellation is used to create finer and more detailed meshes from models with a low polygon count (often called “low poly” models): this is done by creating vertices inside the primitive, that can then be elevated and curved, forming a series of curved triangles that add detail to the model (see Image 5).

This is all done in three sections within the tessellation process, the control shader, primitive generator and evaluation shader, two of these functions can be directly controlled by the programmer, however the primitive generator is fixed and can only be controller by the control shader.

Patches are a new primitive that were introduced alongside the tessellation shader, they are a user defined number of vertices that form a sequence, this sequence generally has no intrinsic form and has no predetermined vertex count unlike other primitives. Like to the tessellation described above, patches are used in the tessellator to create more primitives, except the form of the patch can be defined by the programmer, giving more control over the result.

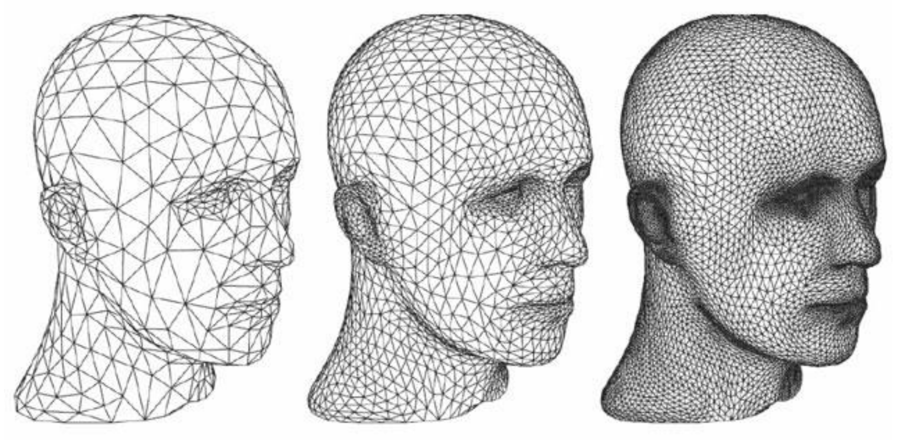


Image 5: A low poly model of a human face refined using tessellation

The process of tessellation starts after the vertex shader outputs the results from its process, at this point, if the tessellation is set to be used in the pipeline, it will arrive in the control shader. The control shader is used for one thing: telling the tessellator how much it needs to tessellate, using the outer and inner levels of tessellation, as previously discussed (see Image 6). This refers to the edges for outer and the space inside vertices for the inner.

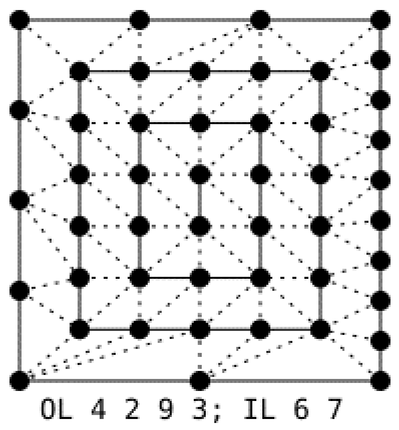


Image 6: A quad tessellated with outer and inner layers

After the control shader has set up the values, the automated tessellation continues within the primitive generator, creating new vertices inside the patch. The results are then sent to the evaluation shader that chooses the form the patches are received in, like triangles or lines. After this the evaluation shader calculates and decides the position of the new vertices, much like the process previously described for the vertex shader.

4.4 Geometry shader

The geometry shader is the last shader in the vertex processing line and, just like tessellation, it will be skipped if not explicitly activated. It was first added in OpenGL version 3.2 in 2009, and was seen by some as a mild disappointment: its capabilities were more limited than many hoped. This shader is possibly the least utilised shader, because the most sought after function (geometry tessellation) has not been implemented in a performance-friendly way (the separate tessellation shader was later added as a specific tool for geometry tessellation).

The main function of geometry shader is to view and modify entire primitives, rather than a single vertex. This gives the programmer greater control over processing of the model, for example: vertices could be positioned differently based on their neighbouring vertices, or could be given data used in future processes within the pipeline. The most common use for this shader is to create new primitives from the as-received primitives. For example, if the shader receives GL_POINTS (single point vertices), these could then be formed into a GL_TRIANGLE_STRIP by creating new vertices around the input vertices.

4.5 Vertex Post-processing

This is the last process for all the vertices that were processed by the previous stages. In post processing, the pipeline goes through a further set of operations before finally sending it to be processed into fragments for the screen space.

If the programmer ever needs to access the vertices created within the vertex processing stages discussed above, it can be done here. The transform feedback operation only uses the output of the last stage that processed the vertices (for example, if geometry or tessellation were not used, the input will be from the vertex shader). After receiving this data, the operation will format the vertices into a buffers, depending on the chosen buffer mode (Interleaved or Separate). Interleaved buffer format records all the data that was received by the transform feedback operation into a single buffer, while Separate will write the data into multiple buffer objects.

This operation can be very useful for simple physics calculations on processed vertices, and can allow the GPU to operate repeatedly on the same set of vertices. For example, to create a GPU based particle system, the programmer can initialise the system, give it starting values for each particle, and then transfer that data to the vertex processing stage. Then the processing stage calculates and executes the necessary movement of the particles and passes it to the post-processing. The transform feedback operation copies the processed data, as well as passing on the current data to be displayed. The copied data is then passed through the same pipeline, with each particle again moved; in this way the movement of particles can continue until the cycle is stopped. The benefit of this method is that the pipeline can do all calculations on the GPU without needing to move data back and forth between different hardware locations. This grants the pipeline an effective particle system, however, if these particles require more complex physics simulation, this will be less efficient.

Before the pipeline turns the primitives into pixels for the screen, it needs to remove any unnecessary clutter in the virtual space. Primitives entirely outside the view volume are not passed on further, since they are not rendered [7. p.19], whereas primitives that are partially inside the view volume require clipping [7. P.19]. This process is necessary to the performance of the pipeline, so the program only processes the primitives that are relevant. Each primitive is treated slightly differently in the clipping process, as the process sometimes only need to discard a portion of a primitive. The clipping process uses the view frustum to calculate whether a vertex is outside the view space (see Image 7).

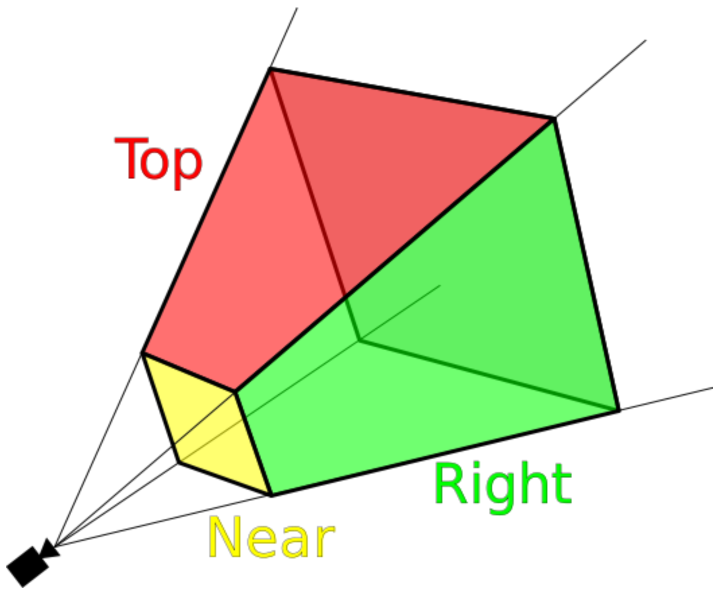


Image 7: The view frustum using a perspective projection model

In addition to the sides of the frustum, programmer can also define near and far planes to clip primitives that are too close or far from the camera. If all of the vertices of a primitives are outside of the given box, the whole primitive is discarded. A more complicated process is used if only a portion of the primitive is outside the frustum. In this case, the clipping process will discard the outside vertices, create a new vertex on the frustrum edge and connect it to the primitive.

4.6 Primitive Assembly

Primitive assembly is the last stop for primitives before rasterization, this is process converts the streams of vertices into sequences of base primitives (a list of line vertices generates line primitives and so on). The order of assembly of the primitives is as follows: any primitives created in the initial vertex rendering will be placed first, if tessellation was used in the pipeline, the patch primitives will be generated in order of their ID values, and finally the primitives generated by the geometry shader are placed last.

After the primitives are ordered, pipeline does a final operation on the triangle primitives: face culling.

Face culling is done to reduce strain on the rasteriser, similar to clipping (clipping is sometimes called view culling, as it culls any primitive out of the view). In the same manner, face culling discards any triangles that don’t face the camera: to determine which triangles face camera, the “winding order” is examined. The order of the vertices is specified in primitive assembly, and can have clockwise or counter-clockwise ordering, depending on which face the process is told is facing towards or away from the camera. This works because the ordering of the vertices becomes reversed if the direction it faces relative to the viewer is reversed (see Image 8).

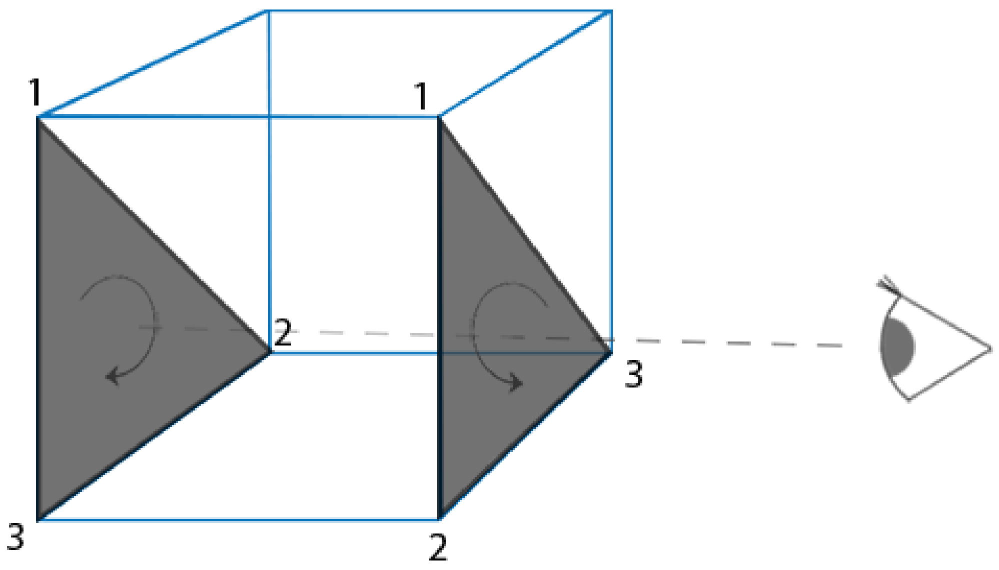


Image 8: The first triangle has the correct order, the one facing backwards has it reversed.

This process is automated and cannot be modified by the user, however the user can choose which triangles should be kept, with the others being discarded in two ways: by defining the correct winding order a triangle must have to not be discarded, as well as deciding whether forward-facing or backward-facing triangles are discarded.

4.7 Rasterization

Rasterization is the operation in which the pipeline converts primitives to be displayed into fragments, which are collections of values that contain a location in window space. Many things can be defined or activated by the programmer, including depth values and option values.

The first step in rasterization is transforming 3D coordinates to 2D space. This is done using perspective projection calculations in which the program uses camera space coordinates to calculate the distance and size a primitive would take on the screen. After the coordinates of each vertex of the primitive are calculated in screen space, the program needs to calculate which pixels the primitive overlaps. This can be done using many different methods but the simplest uses pixel centre overlapping. Simply put, if the pixels centre coordinates are inside the primitive or in the case of some primitives, are nearby, it creates a fragment in that screen location (see Image 9).

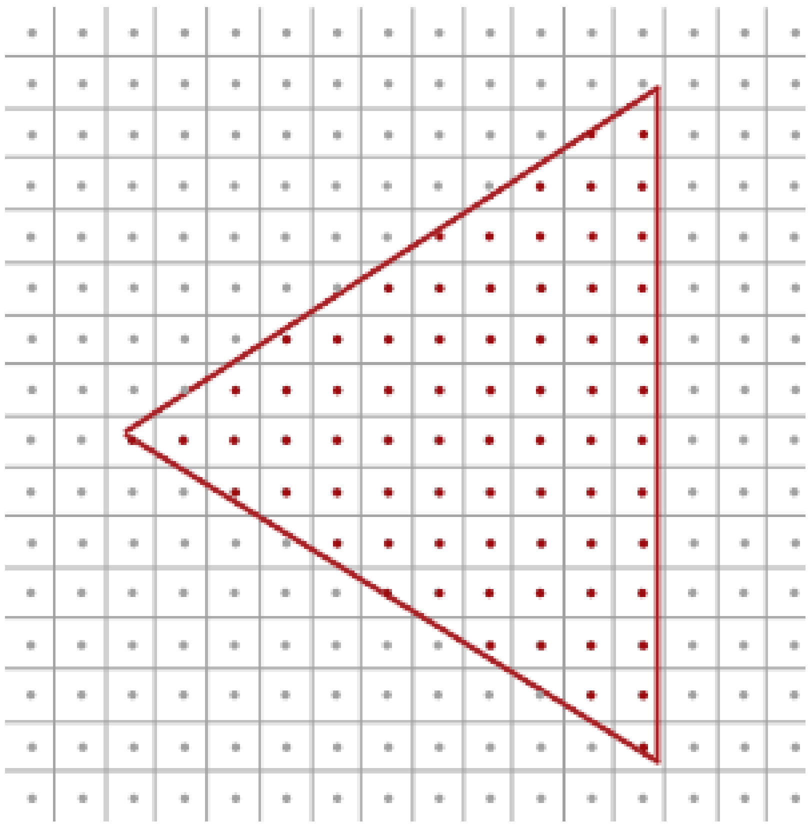


Image 9: An example of pixel centre overlapping (the pixels that would be used to display the triangle are marked with red dots).

OpenGL handles primitives relatively simply and has various things that can be defined by the user to change its behaviour. The point primitive would normally be displayed by the pixel it is closest to, however, this can be modified using a point size variable and enabling point sprites. This allows the program to draw point primitives as bigger than a single pixel. Lines are similar, the pixels are selected by calculating the pixels whose coordinates lie between the start and end of the line.

For triangles (and any other primitive that encloses an area), this process is more complicated. The processes of pixel selection for point and line primitives did not require intricate calculation, but the process for more complex primitives needs a more sophisticated method. OpenGL uses barycentric coordinates which is a method of where you can locate the position of any point inside a triangle using three values between 0 and 1 (the sum of the coordinates for any point is 1). These values describes the amount the coordinate is towards that vertex, with 0 meaning it doesn’t take the vertex direction at all into concideration and with 1 meaning that the direction is completely towards that vertex of the three corner vertices of a triangle. To use this coordinate system, you can for example take 0.33 of the first vertex, 0.33 of the second vertex and 0.33 of the third vertex so you arrive right at the middle of the triangle because it adds one third of each vertex directions to the equation.. This means the program is able to compare the locations of each pixel, and determine if the coordinates of that point are located inside the triangle to be displayed. Of course, this is a simplification of the process, and the full execution is not fully explained in the documentation of OpenGL.

Rasterization is chosen method of the popular modern rendering APIs, this is because the high efficiency it provides allows the program to do everything in real time. However, in the world of rendering there is another method that is used when image quality is more important than speed: ray tracing. Ray tracing is a method of rendering where the program calculates the rays that emanate in every direction from each light source, calculating all the reflections that would be created as light bounces off objects and into the camera. This method can be used to create realistic renditions of scenes that couldn't be easily generated using rasterization (see Image 10).

Image 10, a scene rendered with ray tracing, note the realistic reflections.

Due to the amount of processing required, real-time ray tracing does not currently have any application in rendering for video games. However, there is ongoing research trying to implement it in an efficient way that may be able to rival the performance of rasterization. With future increases in computing power, ray tracing might still prove to be useful in certain applications in real time rendering.

4.8 Fragment shader

Fragments that were created in the rasterization phase are a collection of values that include screen position, colour, depth and other assorted variables. These fragments are then used to determine the colour of each pixel on the screen. The fragment shader is the stage within the pipeline where user-defined code can directly modify these fragments. The incoming variables for this shader include things such as the coordinates of the fragment, the depth value, as well as the stencil value, however these cannot be modified by the user. In addition to these, the fragment also has other modifiable sample values, however modifying these is not often advised as it forces the fragments to go through resampling which may lead to poor performance.

The most common function for the fragment shader is shading. Shading is programming how a surface reacts to light, including how its colours may change, and if any reflection occurs. This is usually done in the fragment shader, as it is the most precise programmable stage of the pipeline. Texture coordinates and colour are also passed onto this shader, so it can take the correct pixel from the texture and apply it to the correct position on the model of the object this fragment belongs to (these are the basics of texturing).

4.9 Per-Sample Operations

Per-sample operations are the last stage of the pipeline, in which the program processes each fragment sample and does executes the required processes on them to create the final image that is displayed on the screen. A lot of these operations can also be performed before the fragment shader process, if the fragment shader is able to permit the pipeline to do it (or the process detects that it can do it without interference from the fragment shader).

Because the default framebuffer is owned by a resource external to OpenGL, it is possible that some pixels of the default framebuffer cannot be written to by the OpenGL process. Fragments aimed at such pixels are therefore discarded at this stage of the pipeline [8. on pixel ownership test]. The pixel ownership test is done so that the program knows that this pixel is not obscured by another window. If the test for a certain pixel returns false, the pixel is not owned by the OpenGL pipeline, and will be handled by whatever other process owns it.

4.10 Scissor Test

The scissor test in an optional operation that can be done if the user has enabled it. The operation simply takes a user defined box of the screen space and discards any pixel outside of it. This test can only work if the box size is given, along with a size and starting location.

4.11 Multisampling / Antialiasing

Because rendering uses pixels to create shapes, it may have to force slanted lines and round shapes to have jagged edges. This is because each pixel is a square, making it impossible to draw shapes that do not lie perfectly on a set of pixels, but with antialiasing programmers can get around this by creating semi-transparent pixels on the edges of the shapes by using multisampling. With multisampling, each pixel at the edge of a polygon is sampled multiple times. For each sample-pass, a slight offset is applied to all screen coordinates [8, on multisampling]. In rasterization, a “sample” refers to an area where program checks for intersection of primitives, in single sampling, the sample is taken to be at the centre of each pixel. In multi sampling, each pixel has multiple points inside it to check for intersection. Antialiasing uses this sampling to create semi-transparent pixels on the edges of triangles and lines to make displayed lines seem smoother (see Image 11). The sampling itself is done in rasterization, however the transparency of the pixels is done in per-sample operations.

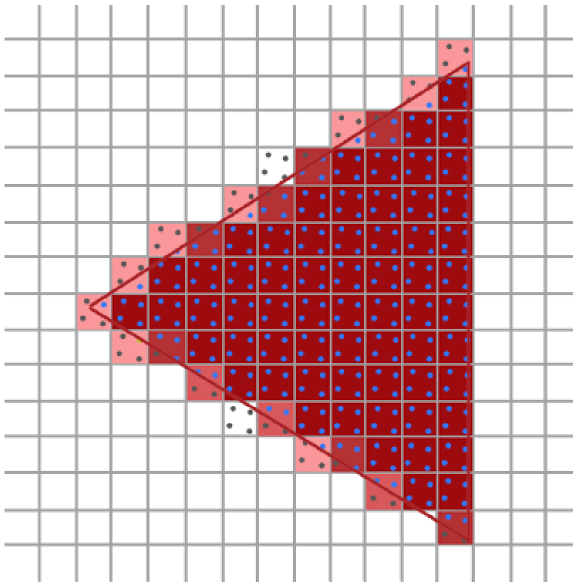


Image 11: A triangle with MSAA 4X, the blue points represent the samples inside each pixel that are located within the triangle (note how the more samples are inside the triangle, the darker the pixel is).

4.12 Depth test

As the final image is based on 3D objects turned into 2D imagery, the program has no implicit idea how far the objects are from the camera, so to account for this the pipeline has depth buffer. In the depth buffer, the pipeline has saved the distance from the viewer to the fragment and thus knows which is the the closest fragment it has. This test is required to render all the fragments in the correct order, otherwise some of the fragments that would be behind the object, might be rendered in front, leading to errors in the image (see Image 12). The exact functionality of depth test can be controlled by the user using it certain commands, for example you can tell it to put the back fragments to the front and vice versa. There are many different testing commands you can give in which all of them give different functionality. The exact operation of the depth test is as follows: the depths of two fragments are compared, and the further away of the fragments is discarded, the test is then repeated with more fragments, until the closest fragment at that pixel location has been found. This operation can also be done before the fragment shader if the fragment shader will not modify depth values, or if user forces it to do it before it (in this case, the fragment shader cannot redefine depth values).

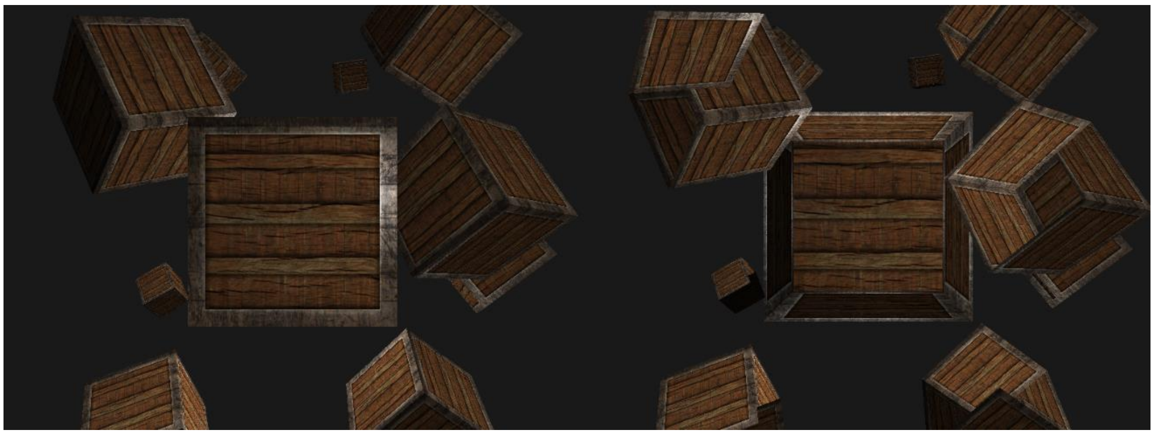


Image 12: A comparison of images rendered with depth testing (left) and without (right). Note how some of the back sides render on top of the front sides, leading to errors in the image when depth is not examined.

4.13 Stencil test

Stencil testing is a lot of like Depth testing, with the difference being that stencil tests can have the values that undergo comparison are user-defined, unlike in depth testing where the value is defined by the pipeline. A stencil buffer functions using 8 bits per stencil, meaning that the stencil can have up to 256 different values written to it and used for comparison. The rules for comparisons can also be defined (unlike depth testing), for example, the given value for a fragment is tested to the value that is written on the stencil buffer, rather than between different fragments. This testing can be used in many ways, and can be used for more complex procedures than depth testing. A good example is drawing and outline: by rendering an object normally, telling the stencil to write 1 to each pixel that contains the object, and then rendering a slightly larger object that discards any pixels that read 1 that fall within the stencil buffer, the only pixels rendered will be surrounding the edge of the first object (see Image 13).

Image 13: A model with an outline

4.14 Blending

Blending is simply blending values together: in the case of the pipeline, this stage blends together colour values. Blending combines the incoming source fragment’s R, G, B, and A values with the destination R, G, B, and A values stored in the framebuffer at the location of the fragment [3. p. 247]. The input for the blending operation is generally the fragment colour values that have been output from the shader. There are multiple ways the user can define the operation, such as whether the operation sums the values, subtracts the values, which value would subtract from which, choosing the smaller of the values or choosing larger of the values, and so on. The most common use for blending is transparency, transparency being defined by the alpha values of the incoming RGBA data.

4.15 Dithering

Dithering is a method used in rendering when the program wants to lower the quality of an image with a restricted colour palette. The computer limits the colours by rounding the colour values to the closest bit value that the program can use in the restricted colour chart. Normally doing this without dithering would cause the image to become very ugly with obvious edges due to rounding, but with dithering, the program can distribute each pixel’s colour within the adjacent pixels to reduce the perceived drop in quality.

5 Implementation

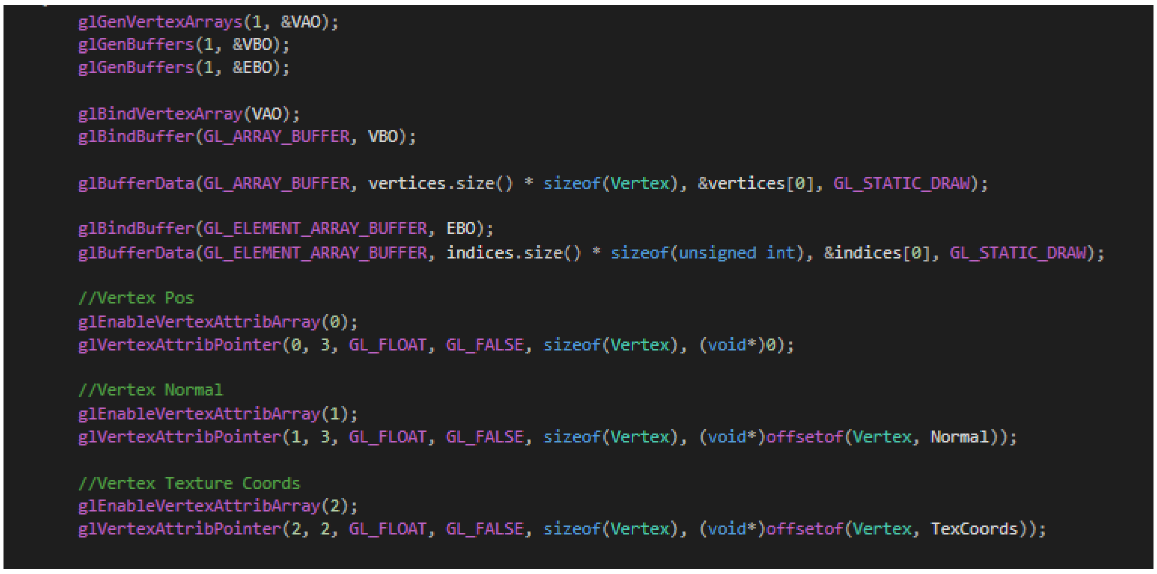
Using the previous mentioned methods and tools, professional developers can create amazing effects and renders of the games you can see today. While the different rendering pipelines differ in order and optimization, many of the main shaders and functions are widely used. The following chapter will go through steps of some basic implementations of the pipeline as well as explaining how they were implemented. Not all steps will be discussed in this chapter, either because they aren't user-definable, or do not provide anything meaningful to discuss that will not have already been covered.

5.1 Preparation

Before the pipeline can start rendering anything, it needs information of what to draw, given in vertex attributes. The most widely used attributes are vertex position, the normal and texture coordinates, and much can be accomplished just using these. In addition to the attributes, the user can also transfer additional data to be used. This needs to be done each frame as the data will usually change over time. The basic steps of rendering on the CPU side is to tell pipeline to use a certain shader, give it the information needed for rendering, and then giving the shader a draw command.

Uploading the vertex attributes is relatively simple once the user understands the basics of it, for instance Image 15 shows an example of vertex attribute data. It all starts with generating and binding a vertex array object to act as a provider of data, and then writing said data into the buffers to be used. After everything is prepared, the user first tells the program the size of the vertex data for the array buffer and element array buffer, using the glBufferData function, so the amount of space between each array can be calculated.

After this step the program will need to know the size of each individual attribute so that it can know where they begin and end within the array. This size is given in bytes, in the case seen in Image 15 the float has a size of 4 bytes, so in the first glVertexAttribPointer function the CPU gives the location of the array, along with how many elements this array has. In the case of the Vertex Pos (that is, vertex position), consists of float variables and it has three of them so the type and amount gets defined in the function. After the type is defined the function needs to know if the data needs to be normalized, and the size of the vertex data structure. The last given parameter of this function lets the function know how the data in the array is offset, for the Vertex Pos this is set as zero as it will be the first piece of data. In the case shown in Image 14, the normal and texture coordinates have non-zero numbers, as the function will need to know how far down the array it must go access these variables.

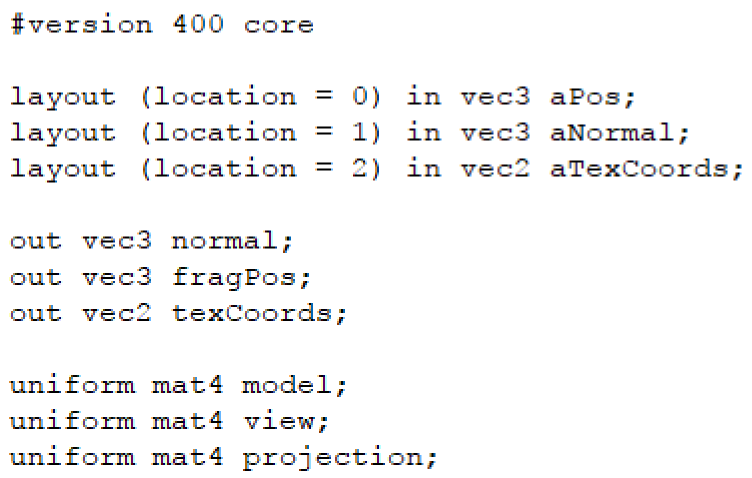


```
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);
glGenBuffers(1, &EBO);

glBindVertexArray(VAO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);

glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(Vertex), &vertices[0], GL_STATIC_DRAW);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(unsigned int), &indices[0], GL_STATIC_DRAW);

//Vertex Pos
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)0);

//Vertex Normal
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, Normal));

//Vertex Texture Coords
glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, TexCoords));
```

Image 14: An example of buffering vertex attribute data.

5.2 Basic render

To demonstrate the operation of the pipeline, a good start is to process a basic model rendering. To output the desired render of given shapes and/or models, the shader must to receive and forward many different values (see Image 15).



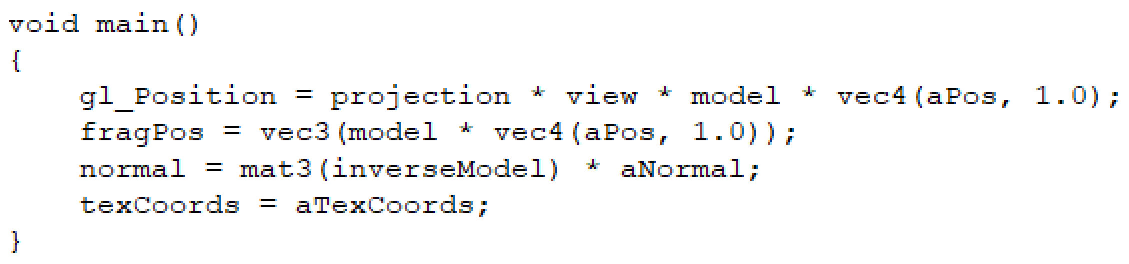
```
#version 400 core

layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoords;

out vec3 normal;
out vec3 fragPos;
out vec2 texCoords;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
```

Image 15: Vertex shader variables

The first variables input into the shader are “layout” variables, these are variables that the pipeline gets from the vertex specification stage, the location number indicates the location in the vertex array that was given during preparation. After that are the “out” variables, which are data that the vertex shader can send forward to other shaders. This data can be sent either by themselves or combined in a struct, which is mostly up to the preference of the user.

After the outgoing variables, the image contains uniform variables with three 4x4 matrices, containing the model, view and projection (MVP) components that the vertex shader uses to correctly position the mesh. The MVP matrices contain important data that is related to the rendered model, camera and the screen. The model component contains the position, scale and rotation of the mesh in the world, the view component contains the directional matrix of the camera, which is calculated using the front facing up directions, and the projection component that has the field of view, aspect ratio value (calculated as screen width divided by screen height), as well as the distance of the near and far planes, so as to know when to clip objects that are outside of these distances.

After these values are sent, the user can simply tell the shader the name of a variable and the desired value, the shader can start processing it in the main function (see Image 16).



```
void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    fragPos = vec3(model * vec4(aPos, 1.0));
    normal = mat3(inverseModel) * aNormal;
    texCoords = aTexCoords;
}
```

Image 16: A simple vertex shader function.

This is essentially all that is needed to produce an accurate image of a model on screen. The gl_Position is the uniform variable that the pipeline uses for the vertex position, it’s calculated by multiplying the MVP matrices with the local position of the vertex it retrieves from the mesh file. After that, the fragment position, normal position and texture coordinates are calculated, these will be used in later processes. With this, a basic shape should form, that resembles the model given, but first the shader needs to know how to handle each fragment. For now the model just needs to be drawn, so the fragment shader just gives it a white colour so a visible output is generated (See Image 17).

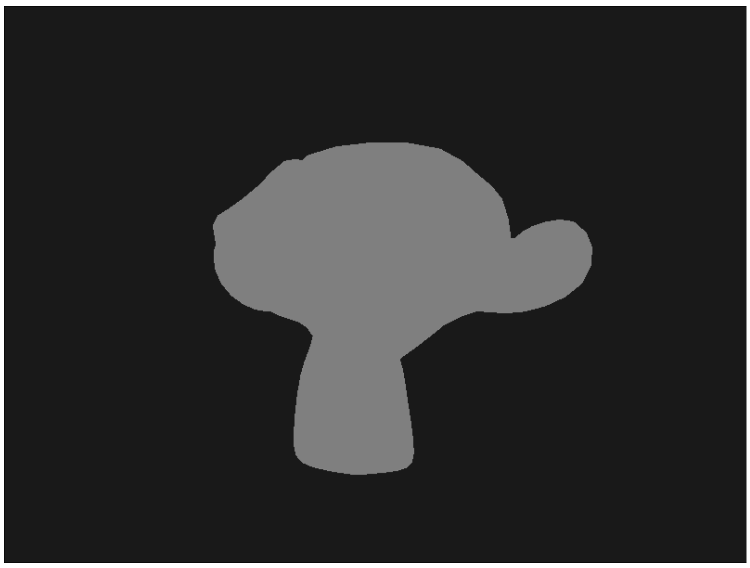
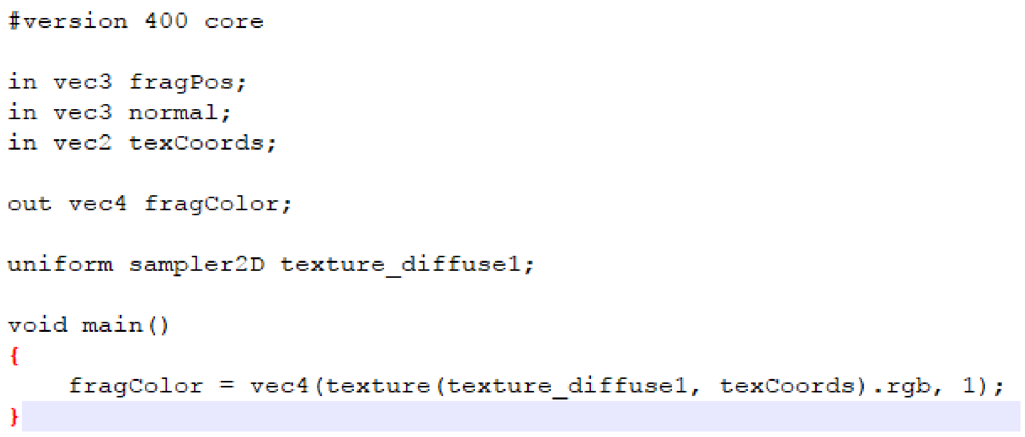


Image 17: A simple rendering of the famous Suzanne model, using the steps outlined above.

5.3 Texturing

Models by themselves are very bland looking, to fix this issue textures are applied. Textures are 2D images that are wrapped around a 3D object using texture coordinates.

These coordinates are one of the attributes that were buffered during preparation, and are essential for proper texturing. During the loading of a texture, it generates a unique id and binds it to the texture data, then at the start of each rendering loop, it retrieves each texture that the model uses. Implementing textures within the shader is simple, as OpenGL’s shader language provides the texture() function, to look up the correct texture location for the desired texture slot (see Image 18).



```
#version 400 core

in vec3 fragPos;
in vec3 normal;
in vec2 texCoords;

out vec4 fragColor;

uniform sampler2D texture_diffuse1;

void main()
{
    fragColor = vec4(texture(texture_diffuse1, texCoords).rgb, 1);
}
```

Image 18: implementation of texture in fragment shader

Textures can be used for various things, whilst a basic colour texture can be used for simple visuals, often more complex textures are used to give the object an illusion of bumps, and lighting maps that dictate what parts of an object will react to light in a certain way. In video games, most models have all their textures mapped to one texture, and during creation it is the job of the model creator to map all the vertex positions to the correct points in the texture. With the previous code for each fragment the code gets a specific colour for that position. In this example, a concrete texture has been used (see Image 19).

Image 19: A rendering of the Suzanne model using a concrete texture.

5.4 Simple Shading

Shading an object refers to simulating how it reacts to light, and this can be done using less than 50 lines of shader code. This is all done using shading models, where each model depends on the frequency and algorithms of the light calculation. When designing a shading implementation, the computations need to be divided according to their frequency of evaluation [7. p.113]. This frequency refers to the accuracy of the computation, with lowest frequency referring to per-model calculations, and the highest frequency referring to the per-pixel calculations.

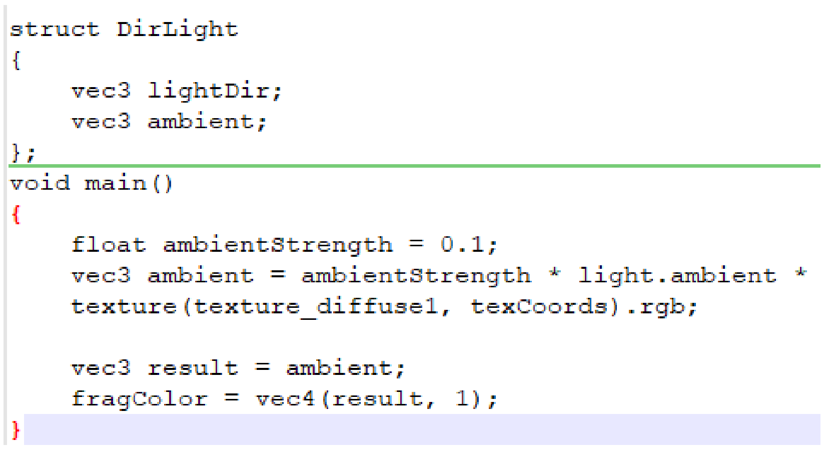
The most commonly used shading model is the Phong shading model, which uses two methods called Phong interpolation and Phong reflection. Phong shading is based on interpolating the normal of the vertices of the fragments and using those interpolated normals to calculate the reaction to incoming light. Besides the interpolation, Phong also creates an illumination model that is in three parts: ambient, diffuse and specular reflection. Each of these can be individually modified within the shader so that the model looks like it has different properties (for example, metallic surfaces reflecting light more than wooden surfaces).

The ambient part of shading is light that technically isn’t there, a lot of the time in video games ambient lighting is treated the same globally and doesn’t have a designated source, simulating indirect lighting. Indirect lighting refers to the light that will have bounced off many other objects before it reached the desired object. This is difficult and resource-intensive to calculate, so to solve this issue, most lighting systems use a fake ambient colour to make the reflection look more real.

Diffused or Lambertian reflection refers to when light hits an object with a rough surface, and is reflected in all directions [9]. This shading is the most important of the three parts as this is the reflection that dominates interactions like the fragment facing a light source. This brightness is independent from the observer, so the calculations don’t change no matter what angle the object is viewed from.

Finally, is specular reflection, this is the direct reflection from a shiny surface that can be seen on smooth objects and is used to make an object shine. This reflection is calculated between the viewer and the light source to see if the light would reflect towards the observers’ eyes. Specular is used and modified depending on the object, to make it seem rougher or smoother creating the illusion that the object is made of specific materials. Combining these three different lighting techniques, a realistic looking object that seems to interact with light can be created.

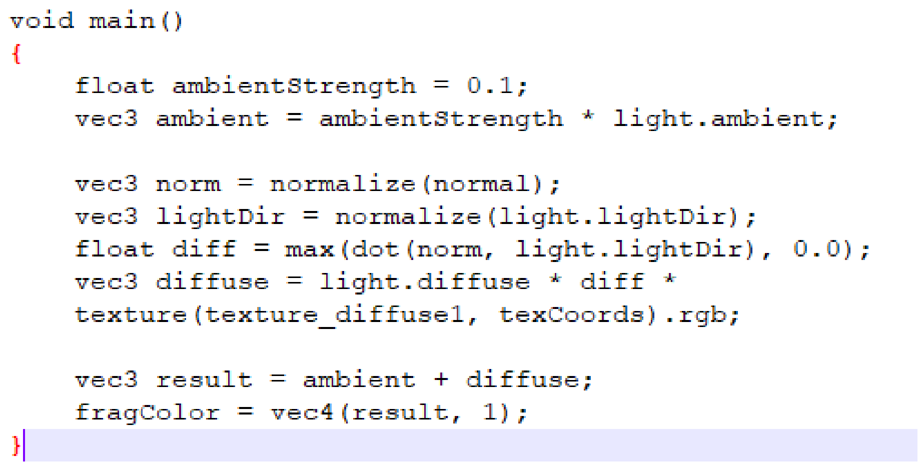
Programming an ambient light is simple, for this the shader has a directional light struct containing all the values necessary to create a light, and that will be expanded as this goes on. However, for now it only contains a direction and ambient colour (a simple shade of red). The shader then combines it with the texture colour of the object, the result vector is used to combine all the elements after calculations image 20.



```
struct DirLight
{
    vec3 lightDir;
    vec3 ambient;
};
void main()
{
    float ambientStrength = 0.1;
    vec3 ambient = ambientStrength * light.ambient *
    texture(texture_diffuse1, texCoords).rgb;

    vec3 result = ambient;
    fragColor = vec4(result, 1);
}
```

Image 20: An implementation of ambient lighting.

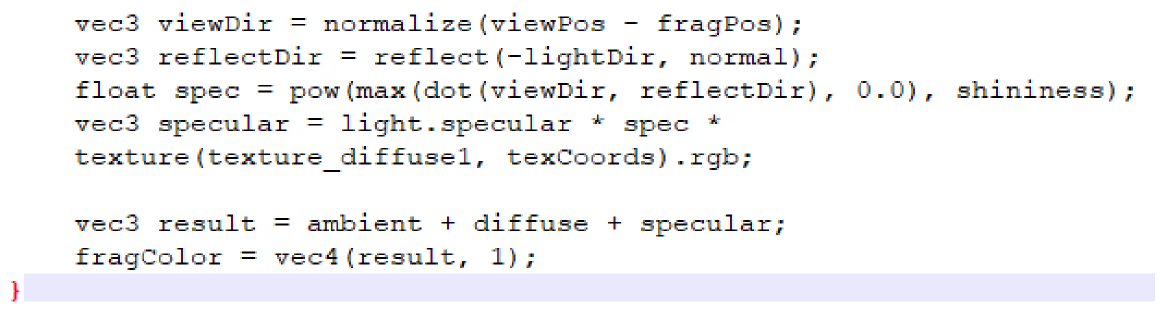
Implementing diffuse reflection requires more work. Unlike ambient reflections where all objects receive light from all directions, diffuse light takes into consideration the direction and range of the light when simulating light reflection. To do this the shader needs to compare the vertex normal direction and the light source vector directions. This is needed to calculate how directly the light hits that spot on the mesh and is calculated by using a dot product. Dot products compare the growth of two vectors and return a value of positive or negative depending on the directions of the vectors. If both vectors are normalized to a length of 1, the function returns a value between one and minus one, with one meaning both the normal and light direction are parallel and minus one meaning they are anti-parallel. Using this outcome, the shader can calculate how strongly the light colour affects that region on the fragment. After the colour value is calculated it is then added to the ambient light to get the result (an implementation of diffuse reflection is in Image 21).



```
void main()
{
    float ambientStrength = 0.1;
    vec3 ambient = ambientStrength * light.ambient;

    vec3 norm = normalize(normal);
    vec3 lightDir = normalize(light.lightDir);
    float diff = max(dot(norm, light.lightDir), 0.0);
    vec3 diffuse = light.diffuse * diff *
    texture(texture_diffuse1, texCoords).rgb;

    vec3 result = ambient + diffuse;
    fragColor = vec4(result, 1);
}
```

Image 21: An implementation of diffuse reflection.

The last part of the shading model is specular light, which refers to light that is directly reflect towards the viewer rather than in all directions like the diffuse reflection. This light gives sense of shininess to objects and is different depending on the angle the viewer views the object. Thus, the direction from viewer to the fragment is relevant to the calculations of the light. A function exists within the OpenGL shading language that does the calculating for the user on how a light would bounce off a surface, by comparing the direction of the incoming light with the normal of the position. Then the direction of reflected light is compared with the direction to the viewer and has a dot product made from the two vectors. The result of the dot product is then put to the power of the shininess factor (see Image 22 for an implementation of these processes). The shininess factor is a value that determines how much specular reflection the object gives, the smaller this factor is, the bigger the area of specular reflection is.



```
    vec3 viewDir = normalize(viewPos - fragPos);
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), shininess);
    vec3 specular = light.specular * spec *
    texture(texture_diffuse1, texCoords).rgb;

    vec3 result = ambient + diffuse + specular;
    fragColor = vec4(result, 1);
}
```

Image 22: An implementation of specular reflection.

When all of this is combined, the object can be seen to react to light in a way appropriate for a basic lighting model. Of course, there are many, more modern and realistic models that create better looking lighting, such as physics based rendering techniques, however, for this implementation, the Phong shading model is certainly adequate (for the overall outcome of these processes, see Image 23). Note the detail added when compared to the previous model, with the light shining from above the features are seen in more detail and become distinct, with specular as the white light on the head and eyes giving it a bit of shine to give the viewer information of where the light shines from.

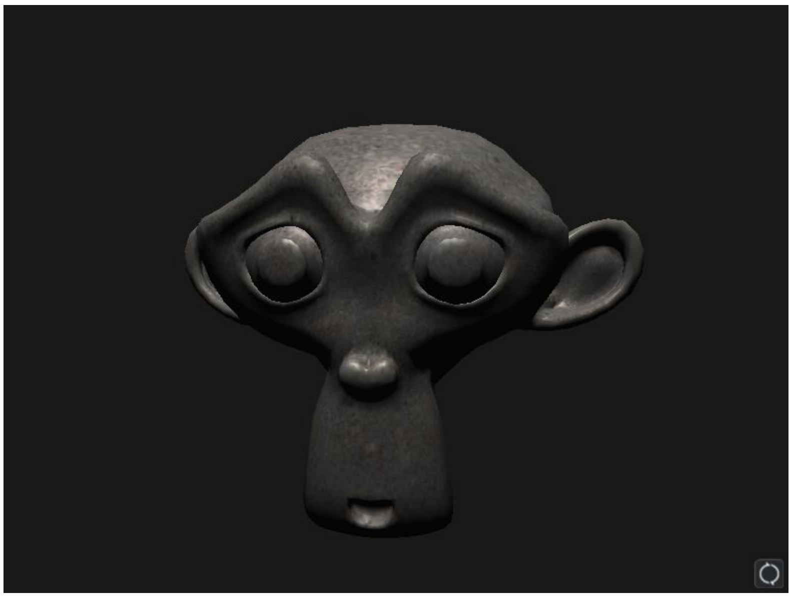


Image 23: The Suzanne model, lit using the lighting techniques discussed above.

6 Conclusion

All things considered, OpenGLs pipeline allows a programmer to create amazing imagery. With each part of the process having plethora of uses and ways to create, it is a great tool for any game. Throughout the years since the acquisition of OpenGL, the Khronos organisation has improved it with the addition of features like tessellation and geometry shaders as well as by optimising it further. Each step of the OpenGL pipeline is useful in one way or another and a clever programmer can always realise the full potential of these steps if they understand them well. For example, without knowing how vertices are handled, modifying them would be difficult, and creating effective lighting would be difficult if the programmer doesn’t know how the pipeline handles interpolation between vertex and fragment shaders. Getting to know the inner workings of these process is important and should be first step in learning how to become an efficient graphics programmer.

Along with OpenGL, Khronos have also continued their development of Vulkan, which is seeking to become the next gen pipeline, giving developers more tools, and allowing the industry to move forward, with a focus on real time rendering. There are already many games that have used Vulkan to create detailed and impressive visuals such as the recent game Doom (2016), that has amazing performance considering the high visual fidelity. Learning how to effectively use these APIs is the key to become a good graphics programmer.

7 Sources

1. A blog on raytracing. (2017). Retrieved from https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing

2. The official Khronos web site. (2017). Retrieved from https://www.khronos.org/about/

3. Segal, M., Akeley, K., Frazier, C., Leech, J., & Brown, P. (2010). The OpenGL R Graphics System: A Specification (Version 4.0 (Core Profile)-March 11, 2010),

4. Sherrod, A. (2008). Game graphic programming Cengage Learning.

5. Shreiner, D., Sellers, G., Kessenich, J., & Licea-Kane, B. (2013). OpenGL programming guide: The official guide to learning OpenGL, version 4.3 Addison-Wesley.

6. Rideout, P. (2010). Blog on triangle tessellation. Retrieved from http://prideout.net/blog/?p=48

7. Akenine-Möller, T., Haines, E., & Hoffman, N. (2008). Real-time rendering CRC Press.

8. Khronos wiki on OpenGL. (2017). Retrieved from https://www.khronos.org/opengl/wiki/Main_Page

9. Komura, T. (Unknown year). A lecture on shading models. Retrieved from http://www.inf.ed.ac.uk/teaching/courses/cg/lectures/slides5.pdf