**Intro**

Oh boy is this going to be a long one... Computer graphics are something that I am extremely passionate about and interested in. I have so many things to say on the subject and I recommend reading at least some of this before you delve into something like my notes on OpenGL. This will cover some of the history of computer graphics, the fundamental maths required, 2D and 3D rendering, and other graphics related concepts. Without further ado, let us delve into this most fascinating topic!

**What Are Graphics?**

 I’m sure you know what graphics are. They are a visual representation of one or more objects represented on some sort of display. Computer graphics is its own field because an entirely new medium had to be invented to display data from a computer. For a painting, you’d use a canvas, for a drawing, paper, but for computers, the modern computer display had to be created.

**History of Computer Graphics**

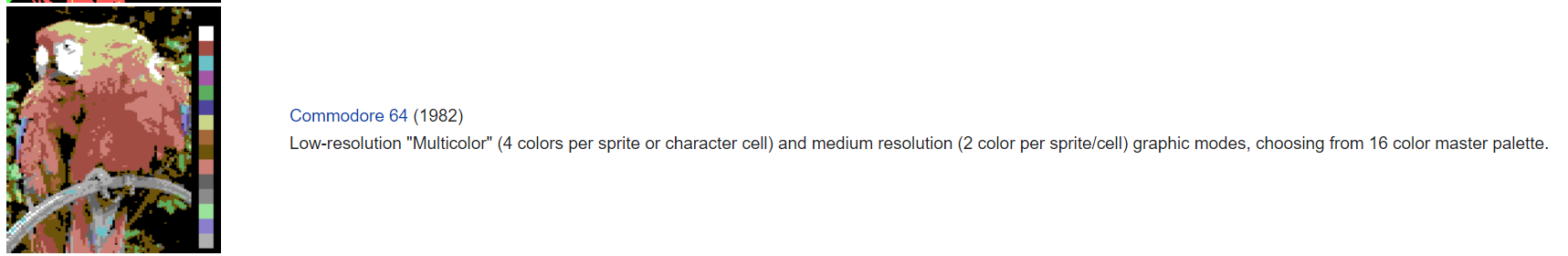
If you have ever heard of a CRT TV, you might know that this stands for Cathode Ray Tube. Cathode-ray tubes were first invented in the early 1920s and were primarily used for oscilloscopes at the time (read my electronics notes if you want to learn more about oscilloscopes). Essentially, these devices were/still are used to display electronic waves and frequencies. In 1950, a man named Ben Laposki used a cathode ray oscilloscope with sine wave generators amongst other electric circuits to create abstract art. Not too long after, in 1951, the Whirlwind I computer at MIT was the first computer created to display real-time data onto a screen. Back then, computers were still using vacuum tubes, and the Whirlwind I was created to allow pilots to experience a flight simulator as the Cold War was in effect. This computer also displayed onto an oscilloscope and had to create an entirely new architecture to run fast enough to display the image. The Whirlwind I ran 16x faster than other computers at the time (keep in mind home computers did not exist at the time so I am referring to computers created by very rich research labs and government facilities) eg. MIT. The new architecture meant performing arithmetic in bit-parallel mode which all modern CPUs still use to this day. In 1955, MIT created what is known as the light pen as part of the Whirlwind project. The light pen was ahead of its time, so you may be surprised that such a technology existed all the way back in 1955. Essentially, the pen would detect the light emitted from an oscilloscope as the cathode electron beam fired across the screen, and it would record the timing of this to the computer, resulting in a method of sending hand drawn images to a computer. You have probably heard of rumors that pong was the first game ever invented. When I was a kid, I was surprised to hear that as a matter of fact, table tennis was actually the first game to be invented! I carried this thought with me until researching for this paper, when I discovered that in actuality, neither one were the first game. The first game to ever be invented was called space war. What’s interesting about space war is that it too ran on an oscilloscope, and it used what are called vector graphics. Old consoles like the Vetrex and games such as asteroids were also vector based graphics. Vector graphics don’t use the concept of a bitmap. They aren’t aware of “pixels”, rather, they are mathematical functions that can describe lines which make up an image. As such, vector graphics are always equal to, if not smoother, than raster graphics. They can scale to any resolution and never lose detail, whereas scaling a raster graphic down means loss of data, so it is very difficult to regain that data perfectly. On modern displays, vector graphics need to undergo a process called “rasterization” which converts the vector image into a raster/bitmap graphic and displays it in screen space (we will look into rasterization more in depth later). Back to spacewar though, it was developed in 1962 by Steve Russel, Martin Graetz, Wayne Wiitanen, Bob Saunders, and others. The game was surprisingly complex. Definitely more-so than pong at least. Two spaceships called “the needle” and “the wedge” would be controlled by one player each and would attempt to shoot one another while being pulled towards the gravitational pull of a star, simply called “the star”. Players had limited torpedoes to fire, as well as limited fuel. I recommend reading the wiki on this one because there are some other details that I need not mention here but are truly fascinating if you enjoy video game history. In 1963, the computer mouse was invented. The first mouse design was essentially a block of wood with two wheels controlling two potentiometers (variable resistors). The large wheels meant that the mouse would drag when moving it in the perpendicular direction. Since potentiometers also have a min and max value as they turn, the mouse could only go so far in each direction. This meant that even picking up the mouse wouldn’t reset the offset position of the cursor, you’d actually have to drag the mouse back to its origin point. I highly recommend The 8-bit Guy’s video on YouTube about the history of computer mice and how they evolved from potentiometers, to light detectors, to cameras. In 1972 Pong was invented, and it is likely that this was considered the first computer game by many because it was really the first one available to the public. It also used raster a.k.a. bitmap graphics, as opposed to vector graphics like Spacewar. I suppose I should elaborate on raster graphics a bit more. Almost all digital media uses raster graphics now. This might seem silly, since vector graphics are scalable, and retain their detail no matter what resolution the screen is, and additionally take up a static amount of storage, however, vector graphics require the programmer to write mathematical functions to describe an entire image as a collection of lines. This is not only extremely resource intensive, but also very impractical, and causes issues when trying to implement complex color (side note, I’d encourage you to look up a mathematical concept called the “Fourier Series” which are a very neat example of vector graphics). Bitmap/raster graphics are a **bitmap,** or 2D-array if you prefer, of RGB(A) values. This is much easier for a computer to read, as it can simply set the luminance of the RGB lights in each pixel to the corresponding value in the on-screen pixel buffer. The trade off with raster graphics is that they take up space depending on the color range and resolution of the image, among other things.

That is pretty much where I am going to end the history of computer graphics, as the rest is fairly common knowledge. I will leave a link here: <https://deseng.ryerson.ca/dokuwiki/mec222:brief_history_of_computer_graphics> which outlines my study guide for the key historical events we discussed. Perhaps now is a good time to discuss a few terms since they will be useful going forwards. Just be aware that I will likely describe more terms as they become relevant. I won’t be using many terms applicable in 3-dimensional space until we have worked through 2D graphics.

**pixel:** A pixel is an array of 3 light emitting diodes (LEDs), one tinted Red, one Green, and one Blue, which create the illusion of other colors when mixed. Each LED typically has a brightness value between 0-256 (sometimes referred to as RGB888 describing 8 bits per channel), and by altering the brightness of these values, we can produce unique colors. The possible range of colors in the 888 model is 256^3 = 16,777,216. A fourth channel, alpha (RGB**A**) can be added to produce opacity. This does not add to the range of colors possible by RGB, but rather adds to ability to make colors translucent.

**voxel:** Likely you are aware of what a voxel is, but perhaps you did not know that it was called a voxel. Voxels are 3 dimensional pixels. These do not really exist in reality (not practically at least), so we generally use the term conceptually. We would refer to Minecraft as a voxel game because it uses 3 dimensional cubes to draw all of its objects.

**Bitmap:** A 2-dimensional array of RGB values that corresponds to the physical pixels on your screen. Bitmaps themselves cannot be 3-dimensional in case you were wondering. Screens are 2-dimensional, thus there is no need for a 3-dimensional bitmap.

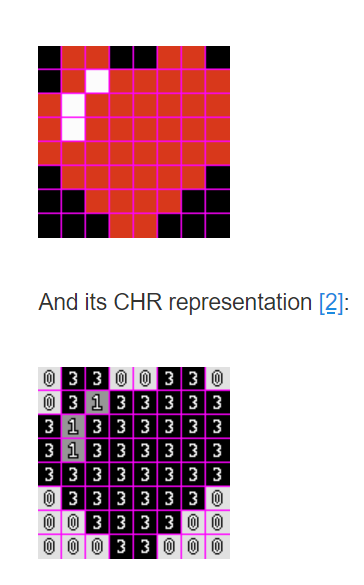
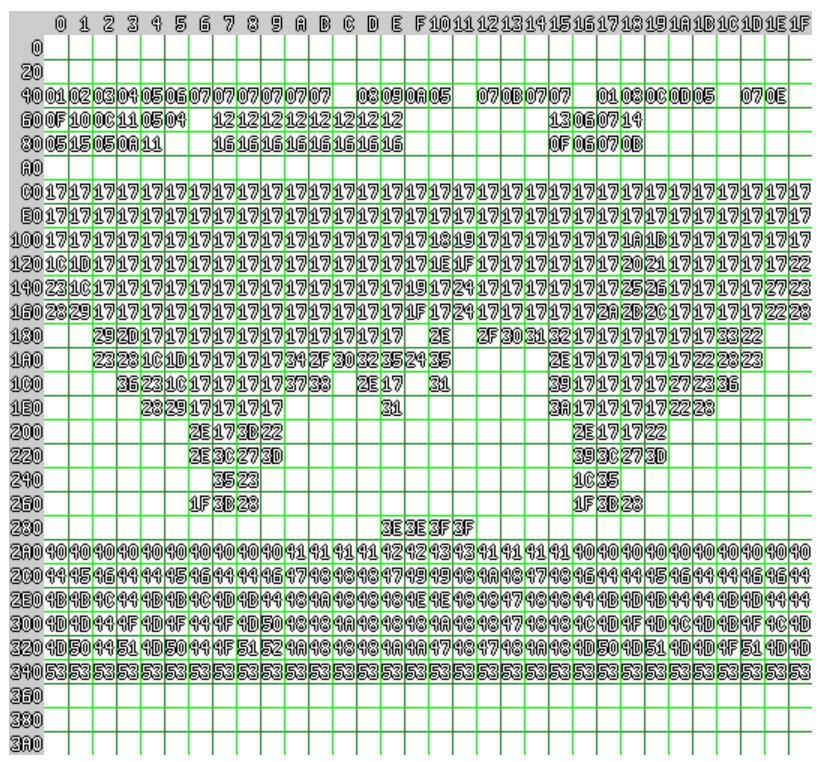
**Color pallet:** A color pallet is a conscious decision to limit the color range at a given time. For older video games especially, using RGB888 was simply too resource heavy. You’d be using 24 bits (that’s 3 bytes) per pixel!!! By limiting each object to a specific color pallet, we could simply use an indexing table to tell the pixel at a given location the value associated with it in the pallet. For example, each pixel could instead use a 4-bit value (16 possible values) and then have a color pallet of 16 colors, each color with 8 bits per-pixel, that way, instead of using 24 bits per pixel, we use 4 bits per-pixel in order to look up a color value in the color pallet which would take up 16 x (8^3) = 8192 bytes. As you can see, the Commodore 64 which was a very capable computer for it’s time had a color pallet of 16 colors and resolution of 320x200 pixels as you can see here: 

**Bit Depth:** Going hand in hand with the color pallet, the bit-depth of a pixel is the number of bits used to obtain sed pixel’s color value from the pallet. In the example I provided in the previous paragraph, we used 4 bits, which yielded 16 possible values as indexes for our color pallet of 16 colors. In this example, the bit-depth of each pixel is 4 then. A bit depth of 1 means that we can only have two possible colors, since one bit has two states (0 or 1), and so on, and so forth.

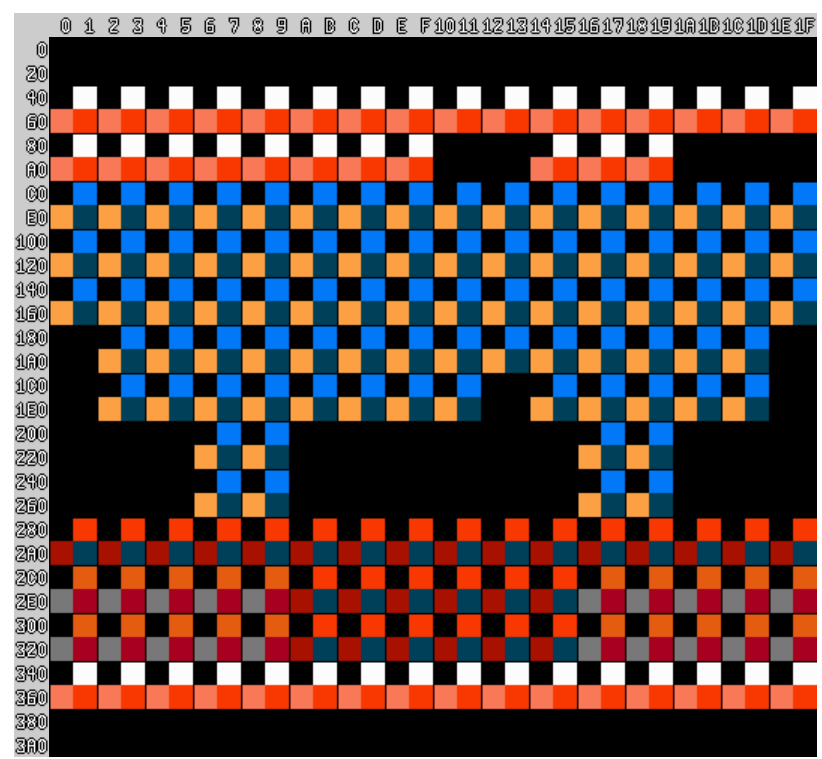
**Tile:** Soon we will discuss tiles. Instead of buffering each pixel into a bitmap (sometimes called a framebuffer), we can instead split the screen into quadrants called tiles. We can then define a smaller array of pixels called a tile which fits the size of one of these quadrants. By re-using tiles and applying transformations to them such as rotation and mirroring, and differing color pallets, we can save on resources.

**Sprite:** Sprites are somewhat of a loose term depending on where you look. A sprite is often a collection of tiles put together to create a larger image, however, sprites can also be 1 tile, or not even use a tile-based system. They are more often than not, associated with bitmap graphics.

Looking at the NES’ Graphics System

 Personally, it was really eye-opening for me to look at an older piece of technology like the Nintendo Entertainment System, and see how it implemented graphics. The pinnacle of 2D graphics of the era was arguably the SNES, however, the graphics system in that console was a major advancement compared to that of the NES, and thus it is likely a bit too complicated as a starting point. I recommend reading this article: <http://www.dustmop.io/blog/2015/04/28/nes-graphics-part-1/> as it is what I will be referencing for now. The NES splits the screen into subsections called blocks which are 16x16 pixels in area. These blocks are then split further into smaller subsections which are called tiles (8x8px) as we looked into earlier in the glossary. Below is Castlevania split into blocks (light green) and tiles (dark green). Adding the hexadecimal values next to each tile on the x and y axis will yield the position of the tile located at that x,y coordinate. One interesting thing to note about computer graphics is that it is very common to have position (0, 0) at the top left of the screen, unlike a cartesian plane used for math where (0, 0) is located at the bottom left. This is simply because of how CRT devices work, starting at the top left, scanning all the way to the right, and then jumping down one scan line and repeating the scan from left to right. The NES uses its own named format for obtaining color via a pallet index, called CHR. The CHR represents the raw pixel art, devoid of color or position. This is essentially the bit depth we discussed earlier, only it’s a 2d array of pixel bit depths, not just one pixel. Each pixel has a bit-depth of 2 on the NES, meaning that there are 4 possible colors that any pixel in a given tile can assume. “But there are obviously more than 4 colors on the screen”, you might be thinking. This is because we can define more than 1 pallet, and use pallet swapping on a per-tile basis. Zoom in on any given tile however, and you will find that it can only ever contain 4 unique colors. Here is the tile of the heart zoomed in. 0 is typically used as a transparency bit, meaning that no color is displayed there (kind of). In reality, a transparent bit is always a shared color with the background layer. As you can see in this image, the background color is set wherever a 0 exists in the CHR representation. A “nametable” assigns tiles located in Visual RAM (VRAM) to a specific location on the screen. This is pretty straightforward, just assign the location address (comprised of the 2 bytes along the x and y axis in the first image) an index value which corresponds to a particular tile in VRAM. Here is what the nametable would look like if you were to write out all the tile indexes in their proper locations. Keep in mind that this is only the tiles on the sprite layer, and not the background layer, or window layer. 

Since we have only added the tile in CHR format, we haven’t specified which pallet to actually use. The NES uses a pallet of 64 global colors. From this pallet, the developers would need to select the colors they wanted to use in their sub-pallets used during compile-time. Each screen can have a maximum of 4 sub-pallets, each of which contains 3 unique colors, and one shared background color used for the transparency effect. In order to actually map the pallet used for each individual tile in the nametable, another table called the attribute table is used. Each attribute tile in the attribute table is given a bit-depth of 2 bits per tile, and directly corresponds to the pallet to be used for coloring the CHR formatted tile at the equivalent location. All of the data combined between the CHR formatted tiles, the nametable, pallets, and the attribute table make up 4096+960+16+64 = 5136 bytes on a given screen. This is quite a commendable use of resources for the NES.

**Advanced Terms**

**Maths Used for Graphics**

**3D Graphics**

Ah yes, the moment I’m sure you’ve been waiting for. As beings who exist in the 3rd dimension, it is only natural for us to take an interest in 3 dimensional mediums, as they are so much more immersive. Unfortunately, 3D graphics are extremely complex. Like very, very complex. There **will** be things that I cannot explain that we will need to treat as a black box. I am basically going to be ripping all of this information from Javid, aka. One Lone Coder’s series on creating a 3D graphics engine verbatim. This series was very well done, and I highly recommend that you watch the entire thing, possibly a few times to fully grasp everything that he goes over. That being said, I’d like to explain the same concepts that he covers in my own words, and my one critique of his series is that he jumps through things a bit too quickly to fully grasp. Hopefully, reading this at your own pace will help you understand a bit better.

So what makes 3D graphics so difficult? Well, a few things in all honestly. Firstly, there is the fact that heavy maths are required to render anything to our screen. This is because there is no way for us to practically describe all 3 dimensional objects in a scene or game using only 2D vertices; and even if we could, there are many other calculations where the 3rd dimension can still help us with even if we rasterize everything onto a 2D screen at the end of the day. Secondly is just how resource intensive it is. There are many obligatory algorithms that we must implement if we want our 3D visuals to be displayed with any sort of fidelity. Now, I’m not trying to scare you away from 3D graphics. In fact, if you are comfortable with the basics of trigonometry and vectors, you should be able to follow along, but just understand that, as I mentioned, there will be certain things beyond the scope of my explanatory abilities.

I want to briefly summarize what is required to display a simple 3D graphic(s) to the screen in a high-level/overview list:

1. Creating a window

2. Having a framework that can execute code on a per-frame basis

3. Describing a projection matrix

4. Inputting our 3D coordinates into a vertex buffer

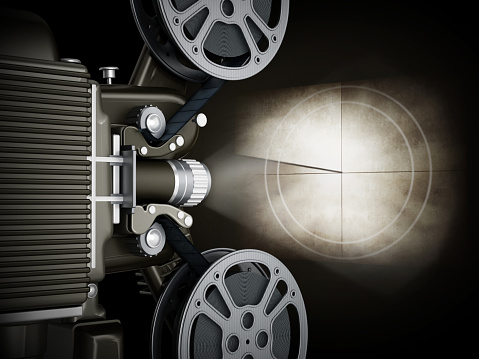
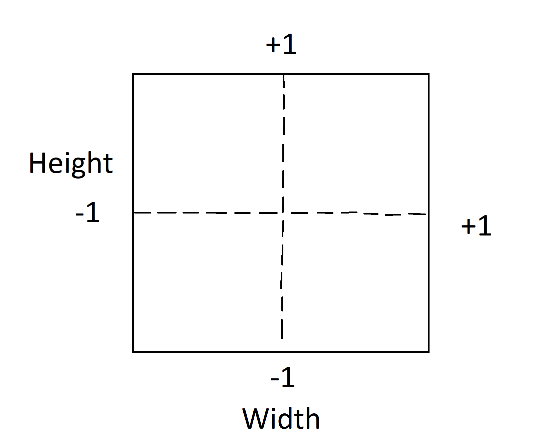
5. Calculating the unit normal of each tri

6. Optimizing vertices using clipping and culling

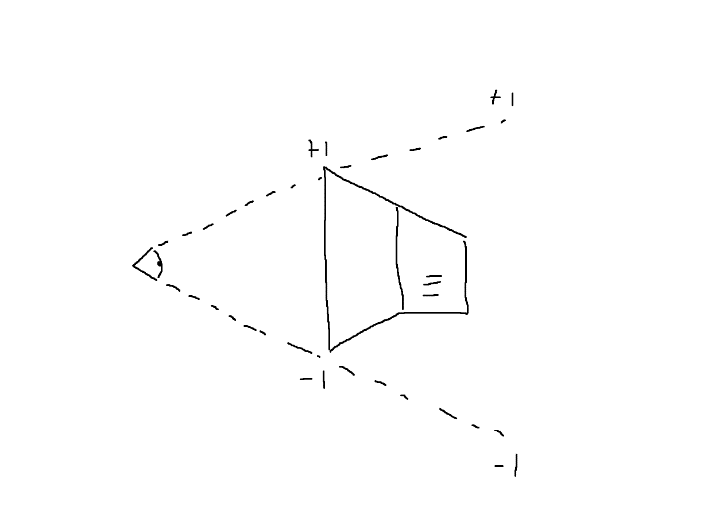
7. Projecting our 3D coordinates into 2D screen space

8. Rasterization

These steps are only loosely in order, as there is no definitive way to do this. You might even be able to skip certain steps, however, this is the basic jist of what you can expect. Obviously, we begin with creating a window since this will be the medium on which we draw our 3D graphics. Side note: I won’t be looking into code here. One Lone Coder does an excellent job with his code, however, do note that it is written in C++ which is somewhat advanced (then again, you’re reading about 3D graphics, I’d assume you have some knowledge of lower-level languages). I will also be doing my own explanation of OpenGL in another document which is a graphics API that you might be interested in checking out. With that little side tangent out of the way, allow me to resume my train of thought. The second step which I outlined is to implement some sort of method of executing code on a per-frame basis. Generally, in game engines, there are 2 areas for writing code: One for general code execution (usually when we create the window), and one that is called each time a new frame is drawn to the screen. You can see this implemented in Javid’s own game engine, the Console Game Engine. A similar approach exists in OpenGL. Since we would presumably like to animate our graphics, it is important that we are able to redraw them every frame. This would also include performing the required matrix transforms for every vertex we wish to update every single frame. I think by the end of this you will have an appreciation for just how fast computers need to execute code.

 Now we get into projection matrices, which is where most of the heavy math resides. Since we use a 3D coordinate system (usually we like vectors in 3D graphics), and we need to **cast** them onto a 2D plane, we introduce an intermediary transform equation called the projection matrix. This can literally be thought of as a projector, projecting an image onto a screen. The human eye functions quite similarly to how a projector works, only in the reverse direction. Instead of casting light outwards, our eyes let light into the cornea and receptive rods and cones in the back of our eyes, which then send electrical impulses to both halves of our brain, thus allowing us to perceive the physical object surrounding us. That was a bit more information than was necessary. The point is that we too have a cone-shaped viewing angle. This is called our Field of View or FOV for short, which you may have seen in a game’s settings. Keep this in that back of your head, since we’ll be revisiting it shortly. First, though, I wanted to touch base on normalization since we will be doing a lot of it. If you know vectors, you’re probably familiar with scaling them to a factor of 1 to get a unit vector. A unit vector will have it’s x,y,z components represented as i,j,k. That aside, scaling things to a factor of 1 is very useful since we can then scale them back up to any ratio. When we’re dealing with displays, there are many aspect ratios that they can be in: 800x600, 1280x960, 1920x1080, 2560x1440, 3840x2160, etc. These are actually the **pixel Ratios**. The aspect ratio would be more like 4:3, 16:9, etc. If we split our screen into 4 quadrants, with coordinates (0,0), (1,0), (-1,0), (0,1), (0,-1), we effectively normalize it (excuse my crude drawing, there were no good demonstrations online). Keep in mind as we go along that a ratio is in fact, just a fraction, or percentage. The y component goes in as the numerator, and the x component as the denominator which will yield some number that we can scale by to get the appropriate measurement to scale. In this case, the y component would be out height (h), and x is our width (w), and let’s call aspect ratio (a). The equation for aspect ratio, is then a = h/w.

With the aspect ratio in mind, let’s take a quick look at the start of our projection matrix. We will have some vector in 3D space, let’s call ‘v’. v = [x,y,z] Now, in order to apply the projection matrix, we need to multiply v by the projection matrix to get the location that v will appear on our 2D screen. We can begin to formulate this. We know that we need to account for aspect ratio so that our 3D vertex is scaled properly, ignorant to whichever resolution we are using. Here is what we have so far: p = [(h/w) \* x, y, z] Since we are translating to 2D space, z does not need to be scaled along with the screen. By the way, think of z as going **into** the screen. z is our third dimension that will allow us to make objects more near or far away. I believe that the reason the aspect ratio does not need to be applied to y is because our screen is landscape, not portrait. Since aspect ratio defines the ratio between both height and width, it get’s applied to the dominant axis of the screen, which happens to be x. Otherwise, if the screen was in portrait mode eg. 9:16, then it would get applied to y, but in all honesty I couldn’t find an appropriate answer to this so take that with a grain of salt.

I want to revisit the eyeball thing I was mentioning earlier. This will likely be easiest if we draw an image from a side profile, so that only the y and z plane are visible, however, the FOV only applies to x and y, since our screen space is 2D! Remember: z is only used for calculations! Not actual coordinates! I can’t stress that enough. That being said, here is an image of your eye’s FOV:

Rather than a cone, the correct term to describe what we’re looking at is called a frustum.

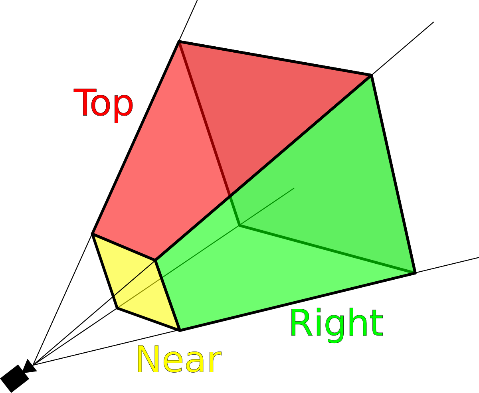
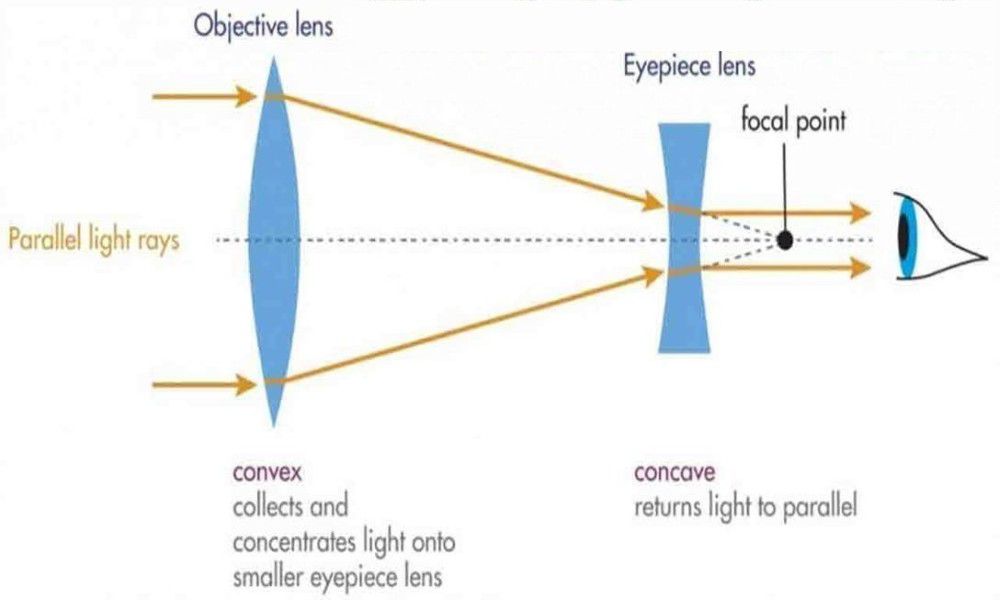
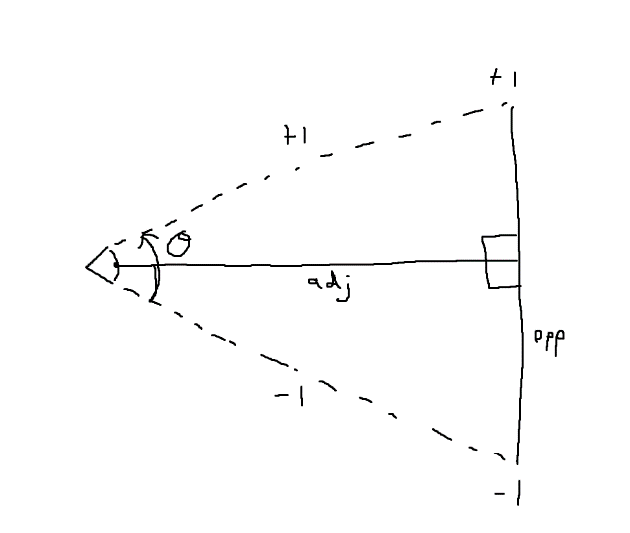


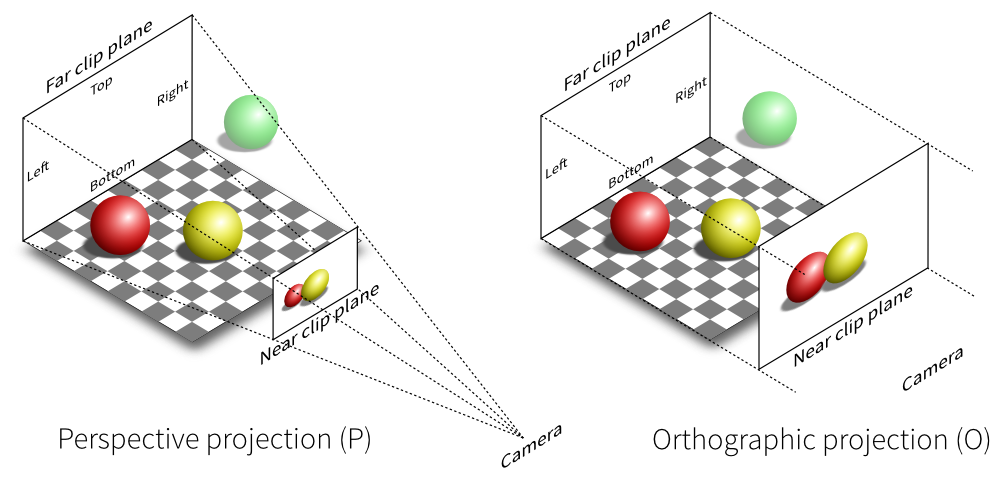
Figure 1 Frustum

Note how we can take any point along the boundaries of the frustum, and give it a normalized value of -1, 0, or +1. This is why distant object appear smaller even though they are in fact, larger than they appear. In other words, you could view a tree directly in front of you and conclude that it must be huge since it obstructs your entire view, and then take a tree of equal size, place it 1km away, and you’d conclude that it’s the size of a tack (not really, but I’m making an example here). If we narrow the FOV, this has the effect of zooming in. On the contrary, widening the frustum has the effect of zooming out. This is somewhat hard to grasp since we can’t actually do that with our own eyes, however this is how telescopes and binoculars function. This actually have a large FOV on the outer lens, and then focus that light into a narrow stream before it reaches your eye on the inner lens.

 So how can we describe this near/distant object phenomenon mathematically for our projection matrix? Well, let’s consider cutting the triangle I drew earlier used to represent a 2D view of our FOV frustrum in half to make up two right angle triangles. That’s right were going to be doing some trigonometry. As angle theta increases, so will the opposite side of our FOV. I’ll just skip straight to the equation and explain why it works afterwards. The equation for FOV will be . Now, you can likely gather that we are splitting theta in half since there are two triangles after bisecting our FOV frustum. It is important to keep in mind that this transformation applies to both the x component of our projection matrix, as well as the y component. That means that the x and y components of each vertex undergoing the transformation will be multiplied by our equation. It is also important to recall that our x and y planes are the only ones that get normalized, not z. The z plane could be any number such as 100. z in the diagram on the left would be the adjacent side since our z plane as you should recall is the one that goes **into** the screen. Let’s say the adjacent size z is 100 then, our x component from our 3D vector has already been normalized in the projection matrix to be 0.5, and that our viewing angle theta is 90 degrees. Now we could perform our FOV equation: x = 0.5 \* = 0.5. So with an FOV of 90 degrees, our values don’t undergo any change. This is because tan(45) is 1 as you might know. Changing the viewing angle to 80 though, for example, would produce: x = 0.5 \* tan(80/2) = 0.42. But hold on, there’s an issue. If you look at this, and really take a moment to contemplate what is happening to each vertex, you realize that as the viewing angle gets smaller, the vertex position scales inwards proportionally which would actually result in a zoom out effect. But we said earlier that shrinking our FOV should have the effect of zooming in. This is easily solved by taking the inverse of the equation: .

We can now pop this into our projection matrix: p = [(h/w) \* \* x, \* y, z]

Okay, now for arguably the most complex part which is the z axis. I said earlier that z could be any number you want, and this is true since we don’t actually use z in order to project the vertices, so it can be whatever it wants to be. However, I also mentioned that we could use it for important calculations and so we ought to normalize it in the same vane as we did with the x and y plane. The z plane is unique in that it does not start at the screen unlike x and y do. What I mean is that z starts from the players head, and therefor there is an offset from the players head to the screen before z actually “begins”. x and y simply start at some corner of the screen.



The image above is really quite a good illustration of the z plane, and also perspective vs orthographic cameras which we will look into at some point. We are obviously concerned with a perspective projection. You’ll notice the near clip plane, which we can call Znear and the far clip plane which we can call Zfar. You’ll understand what clipping is when we get to that. Znear is some distance away from the player’s head, and similarly, Zfar is also some distance away. Zfar-Znear will yield the z distance from the nearest clipping plane to the furthest clipping plane ie. the distance viewable by the player in the z direction where Zfar would be equivalent to the horizon. Once we have the distance Zfar-Znear, we want to normalize it as we have done for x and y. By dividing Zfar by Zfar-Znear we are scaling our distance between Znear and Zfar down by a factor of Zfar so that we get a normalized value between 0-1 which we can then scale back up if necessary. Here’s the equation: The most difficult thing to explain here is that we are scaling down by a factor of Zfar, which doesn’t account for the normalized gap between the player’s head and Znear. In other words, we normalized Zfar, but we need to subtract the normalized distance between the player’s head and Znear, otherwise our z value will be further than it should be. To get the normalized distance from the player’s head to Znear we do which is just our normalized equation multiplied by Znear \* =

This confused me for a while until I realized that I was doing the order of operations wrong like a dunce. Let’s say that we have a vector in our 3D space with a z component set to 3.6. Then let’s just say that our screen is 1 unit away from our head, and that Zfar ie. the clip plane where things stop being drawn to the screen, is set to 10. We plug that into our equation: - = - = 3.6 \* - = 3.6 \* 1.111... – 1.111.. = 4 – 1.111... = 2.89. You can play around with this as well as FOV and some other sliders at this website: <http://learnwebgl.brown37.net/08_projections/projections_perspective.html> which I also found to have some good information on projection matrices.

Let’s revisit our projection matrix again, this time plugging in out newly acquired z scalar:

p = [(h/w) \* \* x, \* y, z \* - ] This looks quite messy but if we replace aspect ratio with ‘a’, FOV with ‘f’ and the z scalar with ‘q’, this is what it looks like:

p = [afx, afy, zq – Znearq]. There is one last minor component that we must consider; and that is that when objects lie further away on the z plane (closer to Zfar), they appear to move less along the x and/or y plane. Take parallax scrolling for example, which was a common trick used in old Gameboy era games. The relationship between z and x/y is that they are inversely proportional ie. as z gets further (larger), changes in x and y (x or y if you prefer) lessen, and vice versa. All we need to do then, is divide x and y by z: p = [, , zq – Znearq].

Now that we have the projection matrix, it will be easier if we convert this equation to matrix form. At the 25 minute mark, Javid get’s into this in his video. If you’re not familiar with matrix multiplication, it might look quite confusing, but if you take a minute to watch it a few times or look up matrix multiplication for yourself, you’ll see that it’s really not that bad. Essentially, we take our vector from 3D space, and multiply it by our projection matrix, to get a new vector on our 2D screen space. When we convert the projection matrix to matrix form, we will get a 3x3 matrix where each row is multiplied by it’s corresponding letter in the 3D matrix, and then once each row is multiplied, each column is summed to get the new vector in 2D screen space. So let’s take our 3D vector, v: v = [x,y,z] and our projection matrix: p = . Our x component in our vector v will be multiplied by the top row so that we get [afx, 0, 0], y will be multiplied by the middle row to get [0, fy, 0] and z will be multiplied by the bottom row to get [0, 0, zq]. Then we’d sum each column: [afx + 0 + 0] = afx, [0 + fy + 0] = fy and [0 + 0 + zq] = zq. Our new vector in 2D screen space would then be [afx, fy, zq]. However, you’ll notice were missing -Znearq in our z component, as well as z as the divisor for x and y. To put -Znearq in our matrix, we must extend this to be a 4x4 matrix (we can only have square matrices) so that we can add -Znearq in the z column. Therefor p = and this will give us our new vector: [afx, fy, zq - Znearq]. Finally, we want to divide x and y by z, but this is not possible when multiplying matrices since we can only add. Therefor multiplying x and y by 1/z will need to be a secondary operation. We can still, however, use our fourth column that is occupied by anything as a means of storage for z. Then we can add a fourth component to our 3D vector v, which will just be a 1, so that when we multiply, it remains unchanged. So in actuality, our 3D vector would look something like v = [x,y,z,1], our projection matrix in matrix form would look something like: p =

and our new 2D vector in screen space would look like v2 = [afx, fy, zq-Znearq, z].

And I think there is where we will end this section. Javid goes on in his video to implement this projection matrix in code which I highly recommend you watch even if you’re not entirely familiar with C++ because it is always a good mental exercise to translate ideas such as mathematics into computer code. That being said, most of the rest of his video explains issues that might arise with the code once it’s already been implemented, and as I mentioned before, this document is really just to demonstrate the concepts and theory behind computer graphics, not how it’s actually coded. If you’d like, you can also check out my document on OpenGL to learn more about that, although I’ll be referencing another series by The Cherno who does a fantastic job at explaining the intricacies of OpenGL. Anyways, if you didn’t understand the projection matrix, don’t worry, it took me a while to grasp it. Fundamentally, we are dealing with aspect ratio, FOV, clipping, and the parallax effect, cramming all of that into a vector matrix, and multiplying each vertex in our 3D space in order to transform/translate them onto our screen as 2D points. As Javid states, you can treat the projection matrix as a black box, where, even if you don’t understand the maths, so long as the concepts are there, you can simply use it blissfully ignorant of how it works under the hood. Next up, I’d like to take at some tricks that developers use to optimize drawing graphics to the screen.

**Optimizations**

**UV Texturing**