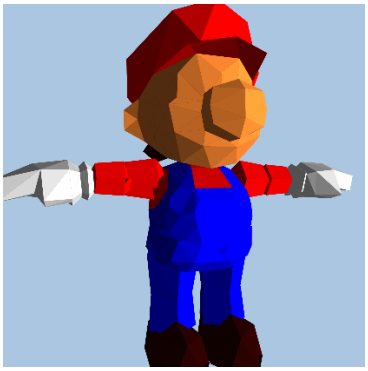
3D Mario, 2D TV

1 – Introducing



I am an avid fan of mathematics and computing, and I often find my favourite areas of both tend to overlap in the form of computer graphics. I'll take you through a journey all about how computers render 3-dimensional models onto 2-dimensional screens, all with one goal in mind: getting this 3D model of Super Mario rendering to a computer screen with nothing but blood, sweat, and sine.

2 – Storing

Before we start, how does your computer store Mario in the first place? Simple - almost every 3D model is just a set of many triangles giving the illusion of a smooth surface.

Spheres with an increasing number of triangles

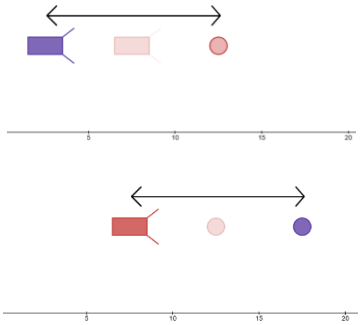
Each triangle is made of 3 vertices. If we can project a single 3D vertex onto our screen, we can rinse and repeat for each vertex and fill in the resulting triangle to get a 2D depiction of Mario's signature Italian physique.

3 – Transforming

Now to do the dirty work of turning a 3D vertex into a 2D point. Let's think about all the different variables we have to consider:

- The position of the vertex (V_x, V_y, V_z)
- The position of the camera (C_x, C_y, C_z)
- The rotation of the camera (θ_x, θ_y)
- The Field-of-view (α)

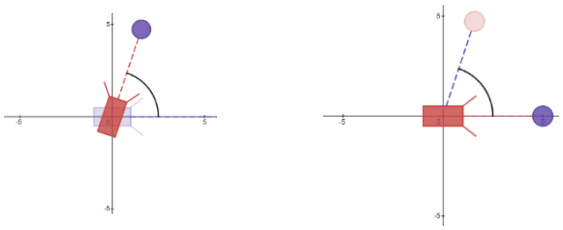
In fact, we can eliminate the position of the camera by just subtracting it from our vertex position. If you think about it, if the camera moves 5 units back from an object, it's the same as the object moving 5 units forward.



That's right, whenever you move the camera in a game, the entire world is actually moving the opposite direction from you! This lets us use the "relative offset" (vertex position – camera position) in our calculations. With V as the vertex position and C as the camera's position, we can very simply describe this relative offset R:

$$R = \begin{bmatrix} R_x \\ R_y \\ R_z \end{bmatrix} = \begin{bmatrix} V_x - C_x \\ V_y - C_y \\ V_z - C_z \end{bmatrix}$$

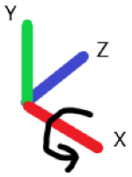
Now we can think of the camera as being at the origin of the world, with vertices existing relative to it. We can also pull the same trick for the camera's rotation, since rotating the camera is the same as rotating the vertex by the opposite angle.



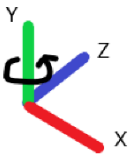
You may notice that I only listed the pitch (θ_x) and yaw (θ_y) for the camera's rotation, but not the roll. This is just because most 3D renderers don't use roll - the fact it makes the matrices much simpler is just a nice side effect (I swear).

Speaking of which, recall the rotation matrices for rotations around the X-axis (pitch) and Y-axis (yaw):

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta_x) & \sin(\theta_x) \\ 0 & -\sin(\theta_x) & \cos(\theta_x) \end{bmatrix}$$



$$\begin{bmatrix} \cos(\theta_y) & 0 & -\sin(\theta_y) \\ 0 & 1 & 0 \\ \sin(\theta_y) & 0 & \cos(\theta_y) \end{bmatrix}$$



We apply these transformations to R to rotate it around both axes, but also negate θ_x and θ_y as we're doing the opposite of the camera's rotation:

$$T = \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(-\theta_x) & \sin(-\theta_x) \\ 0 & -\sin(-\theta_x) & \cos(-\theta_x) \end{bmatrix} \begin{bmatrix} \cos(-\theta_y) & 0 & -\sin(-\theta_y) \\ 0 & 1 & 0 \\ \sin(-\theta_y) & 0 & \cos(-\theta_y) \end{bmatrix} \begin{bmatrix} R_x \\ R_y \\ R_z \end{bmatrix}$$

I'll multiply out those matrices, so we only need to do one multiplication:

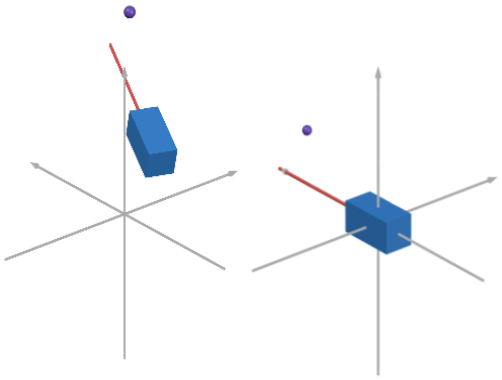
$$T = \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} = \begin{bmatrix} \cos(-\theta_y) & 0 & -\sin(-\theta_y) \\ \sin(-\theta_x)\sin(-\theta_y) & \cos(-\theta_x) & \sin(-\theta_x)\cos(-\theta_y) \\ \cos(-\theta_x)\sin(-\theta_y) & -\sin(-\theta_x) & \cos(-\theta_x)\cos(-\theta_y) \end{bmatrix} \begin{bmatrix} R_x \\ R_y \\ R_z \end{bmatrix}$$

Then let's use $\cos(-x) \equiv \cos(x)$ and $\sin(-x) \equiv -\sin(x)$ to simplify, and substitute our definition of R to get a single equation:

$$T = \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} = \begin{bmatrix} \cos(\theta_y) & 0 & \sin(\theta_y) \\ \sin(\theta_x)\sin(\theta_y) & \cos(\theta_x) & -\sin(\theta_x)\cos(\theta_y) \\ -\cos(\theta_x)\sin(\theta_y) & \sin(\theta_x) & \cos(\theta_x)\cos(\theta_y) \end{bmatrix} \begin{bmatrix} V_x - C_x \\ V_y - C_y \\ V_z - C_z \end{bmatrix}$$

Perfect, now we can get whatever angle of Mario's beautiful form we want.

To consolidate, we've taken a 3D coordinate and transformed it so that the camera is at the origin and pointing in the positive Z direction.

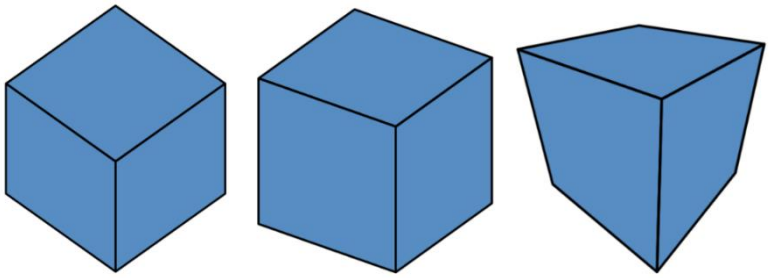


Before *After*

4 - Projecting

But our coordinate is still 3-dimensional! Unless this is a hologram, we're gonna need to squish that transformed position into the confines of a 2D screen.

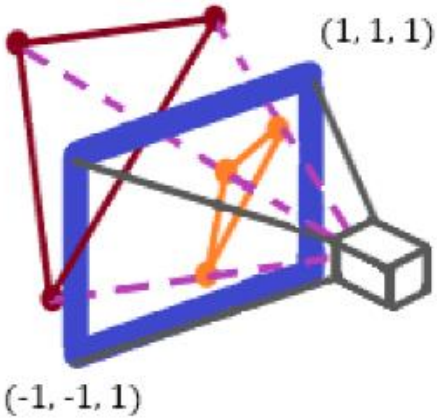
There are many 3D projections you may have heard of, such as:



Isometric *Orthographic* *Perspective*

We'll use the perspective projection to mimic reality, where objects get smaller as they get further away. We take the new T_x and T_y, then divide by the distance in the direction the camera is looking, which remember is just T_z!

By dividing every point by its distance in the camera's direction, we flatten everything onto the 2D plane z = 1:

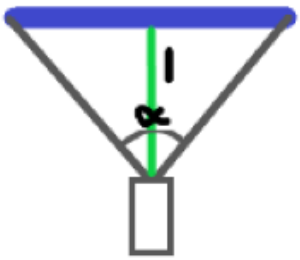


This gives an equation for our actually-final, perspectively-projected, 2-dimensional point, P:

$$P = \left(\frac{T_x}{T_z}, \frac{T_y}{T_z}\right)$$

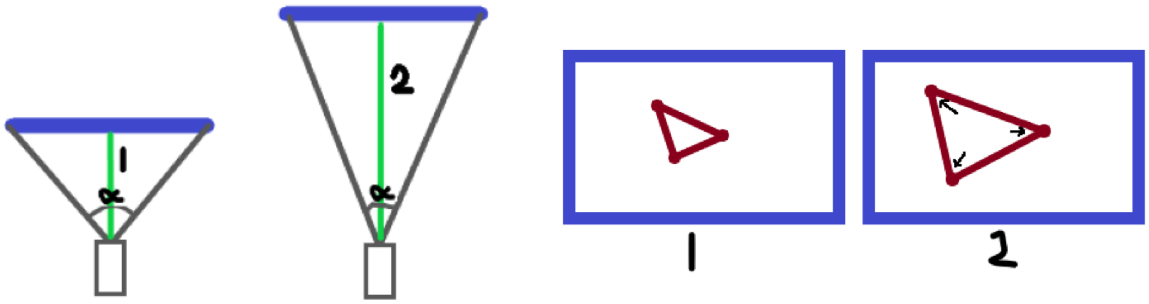
Just kidding we're not done yet; we still need to tackle Field-of-view.

The computer screen will be the area from (-1, -1, 1) to (1, 1, 1) on the plane, meaning the Field-of-view is just the angle between the camera and each side of this "screen-space":



A top-down view of the camera

We want the dimensions of the screen to be constant, but that angle should be able to change. The only way to do that is to change the distance to the screen. You can see here how a lower Field-of-view means the screen gets further away, and you can imagine how the camera zooms in:

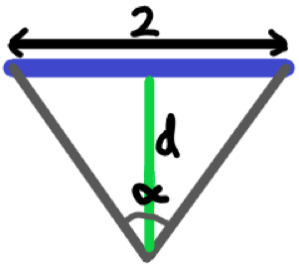


Now we know that a greater screen distance zooms it in more, we can just multiply P by that distance, so it gets bigger if we are more zoomed in. Or, if we label that distance "d":

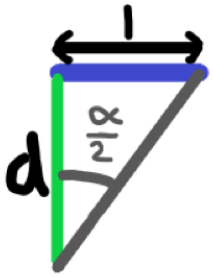
$$P = \left(\frac{dT_x}{T_z}, \frac{dT_y}{T_z}\right)$$

Wonderful, excellent. But I want to be able to give Field-of-view as an angle, not as some abstract-distance-to-a-perspective-projected-screen, dammit! Let's do some trig then.

Let "α" be the Field-of-view and the width of the screen be 2 (-1 → 1):



We can look at one half of the triangle:



And from this we know:

$$\tan\frac{\alpha}{2} = \frac{1}{d}$$

$$d = \frac{1}{\tan\frac{\alpha}{2}}$$

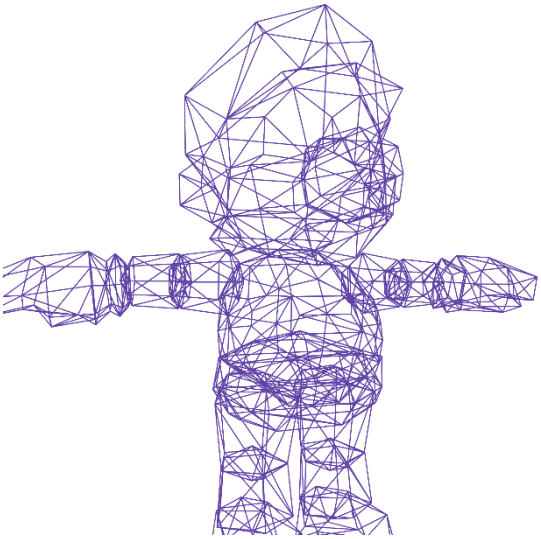
This is a lovely little equation, telling us that our standard distance of 1 is actually an FOV of 90°, or that an FOV of 30° needs a distance of about 3.7.

Since we multiplied by the distance earlier, we can just substitute our equation for distance to get our final equation to project a point relative to the camera on to the screen:

$$P = \left(\frac{T_x}{T_z \tan\frac{\alpha}{2}}, \frac{T_y}{T_z \tan\frac{\alpha}{2}}\right)$$

Huh, quite simple in the end. Anyway, don't worry, we're now out of Field-of-view hell. What matters is we can now change the zoom of our camera, and all of our variables we had to consider at the start have been accounted for.

So now we just repeat that entire process for all of Mr Mario Jumpman-Mario's 424 vertices, then connect them up and we see this:

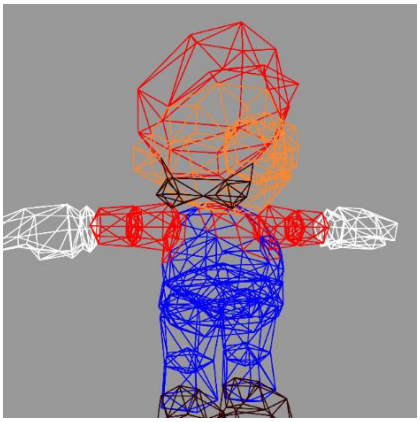


Wow. Isn't he beautiful? I mean, he is a bit see-through. But… he's there, and that's what matters. That's it, we did it. I mean he could have a bit of colour to him… no, no it's fine.

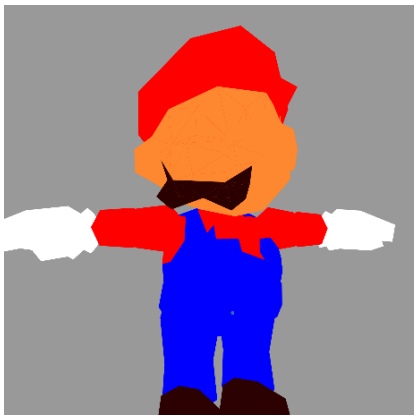
Ugghhh okay we can do better than that. All that maths and he isn't even opaque? Do you know how many times I had to rewrite that FOV explanation just for this purple wireframe?!

5 – Colouring

You know how I said at the beginning that every 3D model has a list of triangles? Well, we also get a little extra bit of information with each triangle: its colour. So, let's just add the colours straight over from the model:



Then we can just make all the triangles opaque:

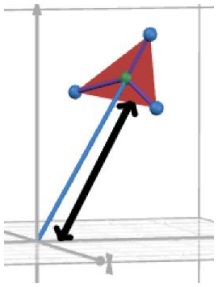


And we're fini- wait that doesn't look right. The triangles are rendering in the wrong order, with the hair on the back of Mario's supple head rendering in front of him? I think this calls for another chapter.

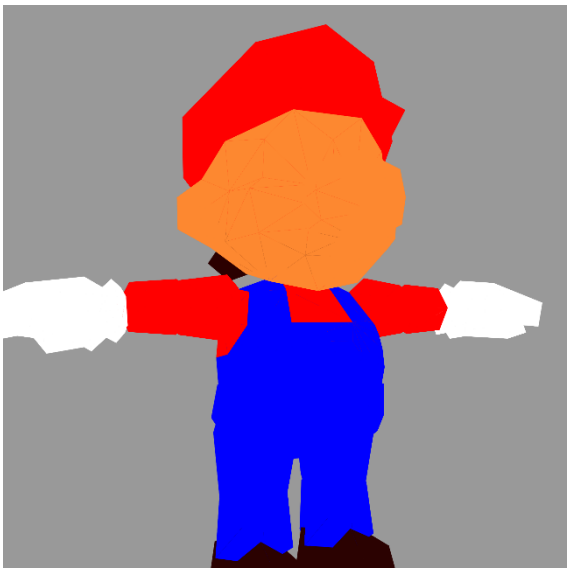
6 - Z-Sorting

This is a common problem in computer graphics, but our current process makes it easy to fix. Right now, we just render the triangles on top of each other in the order that they're stored, which is currently random. So, if we could sort the triangles based on their distance from the camera, then Mario would be appearing properly like the beautiful Euclidean boy he is.

Remember the transformed position T we calculated earlier just before projecting the vertex into 2D? Well, we can just get a triangle's 3 vertices at that stage and average them to get a crude approximation for the triangle's "centre". If we then take the magnitude, we get a distance from the camera to the triangle.



Once again, repeat for every triangle, then sort them based on those distances, and you get this:

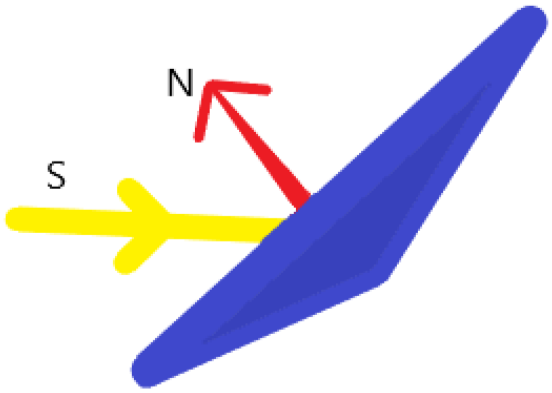


7 – Lighting

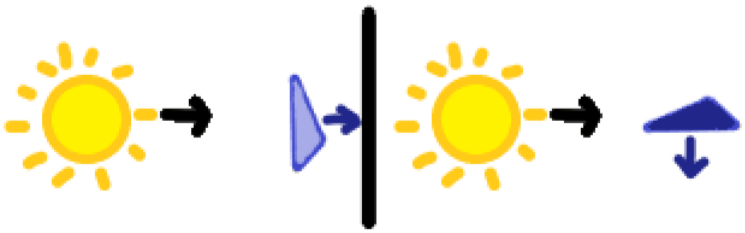
We're nearly done, but there's still more we can do. I'm seeing no details in Mario's curves, no shine in his shoes, no wisdom in his eyes… actually he has no eyes.

Anyway, Mario needs some lighting.

Enter Lambertian reflectance. All we need is a direction vector S for the sun, and the normal vector N of each triangle:



Notice that if we make the normal vector go the opposite way (i.e. we negate it), then as the sunlight vector gets closer to that negative normal, the brightness should increase, and as the sun gets closer to perpendicular, it should get darker.



Parallel vectors make it bright, perpendicular vectors make it dark

This is encapsulated perfectly in our trusty cosine function. Where an angle of 0° gives us 1 and an angle of 90° returns 0. Also recall the definition of the dot product of two vectors:

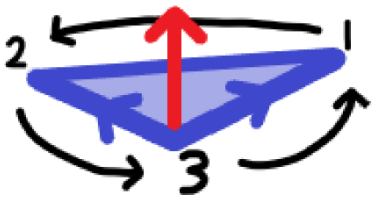
$$\mathrm{A} \cdot \mathrm{B} = |\mathrm{A}||\mathrm{B}| \cos(\theta)$$

This simple vector operation gives us a cosine function very easily! If we just make sure S and N have unit magnitudes, then:

$$S \cdot -N = \cos(\theta)$$

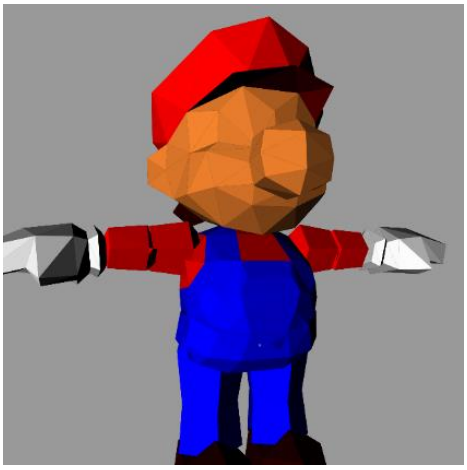
And that's all well and good, but how can we get the normal vector of a triangle? Well, the only thing we know is its vertices, so let's use those. We can work out two edge vectors with simple subtractions, then take a "cross product" - a magical operation that outputs a vector perpendicular to the 2 inputs vectors. This gives us a handy vector which is normal to the surface, a "normal vector" if you will.

But the result of a cross product might end up pointing in either direction, how do we know it will point the right way for our lighting calculation? Luckily, the way that Mario is stored in the computer comes in handy once again. 3D models are stored with the vertices of every triangle always going counterclockwise when looking against the normal:

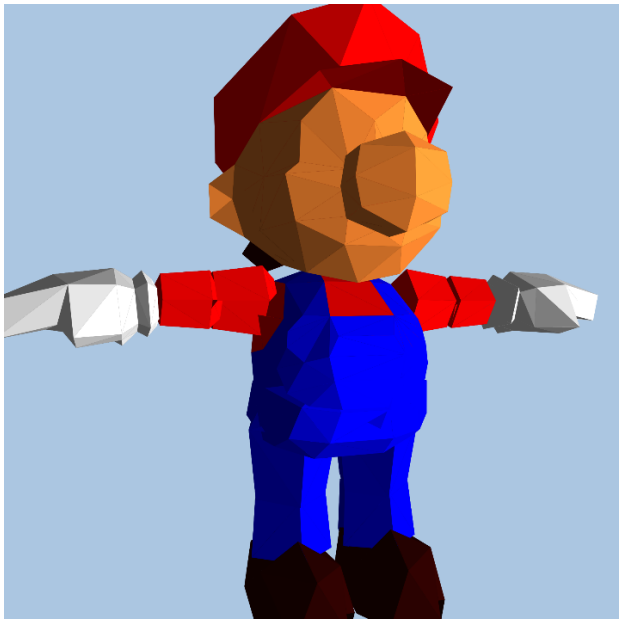


This means the cross product of the edges always gives the right normal vector. How neat!

Let's take a step back: we have the normal vector of a triangle, and can get a scalar value based on how closely it faces the sun. So, we'll just multiply this value with the triangle's colour from earlier to get our final colour. If we do this for every triangle, we should get a fully lit 3D model of Mario:



There he is. There are a few more tweaks to make it look a bit better, namely adding an “ambient light” so the darkest areas aren’t completely black and also rejecting any triangles behind the camera. With that, you now have everything you need to take a set of 3D vertices and colours, and make a pretty image out of them in 2D.



8 – Limitations

As a computer graphics nerd, it’s been incredibly to cool to understand how computers render 3D objects to 2D screens right from its mathematical foundations, but of course I can’t deny there are some limitations.

The main one being that Mario has no eyes! This is because his eyes are stored as a separate “texture” – an image which is projected onto each triangle, whereas I can only change the colour of a whole triangle at once.

Similarly, the per-triangle lighting can be a cool aesthetic, but it would be nice if we could interpolate normal between triangles to get smoother lighting. I haven’t researched the maths behind this, but even if I could do that – once again I can only change the colour of a whole triangle.

Something you won’t have seen is that it runs quite slowly! Mario’s model is made up of 788 triangles and it only runs at about 7 frames per second. And in my mind, this feels both too fast and too slow? You’ve seen all the maths that needs to happen hundreds of times over just to make one frame, but on the other hand modern computers can render hundreds of thousands of triangles at hundreds of frames per second! I suppose I’ll just call it a lesson about how quick computers are these days…

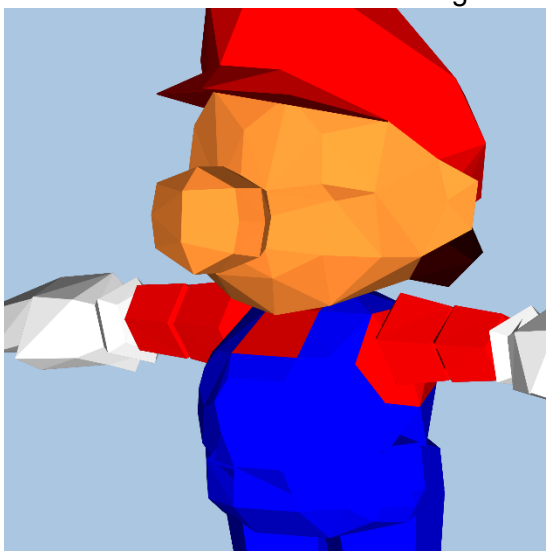
Amusingly, these are all limitations of Desmos, which I’ve been using to compute and show all of these renders. I don’t regret choosing Desmos though as these limitations keep the scope of this essay small enough, and it also has closer ties to mathematics than computing, which often makes me express things more creatively than how I might code them.

9 – The fun bit

You can check out the final Desmos project with Mario right here:

https://www.desmos.com/calculator/lmvk3sfume

Now we've gotten to the end, let's play around!



Orthographic:

https://www.desmos.com/calculator/vlanmlkpwf

It didn't have to go like this.

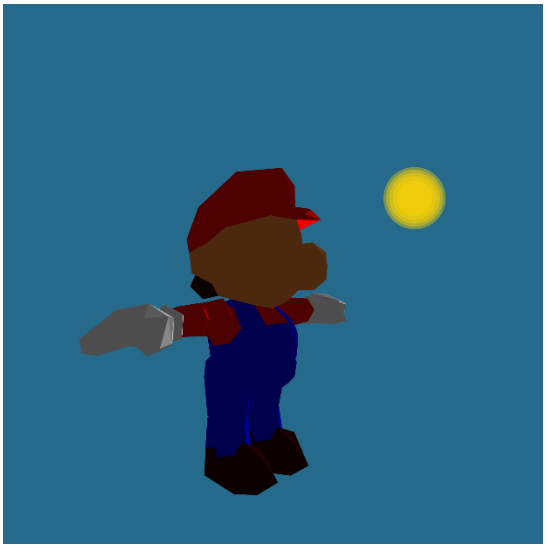
What if we just stopped at

$$P = (T_x, T_y)$$

Day-night cycle:

https://www.desmos.com/calculator/ozqfrmc0co

Pretty convincing just by making the background colour more dynamic and adding a sun in the light direction.



Literally just me:

https://www.desmos.com/calculator/6bauu719zy

Theoretically I could scan my head and colour every triangle to give an uncannily realistic replica, all within the confines of Desmos.

But that would be stupid.