Bringing the virtual world to life through a mathematical lens

Yu Xin Lin

April 2026

1 Optical flow and Pixel motion

Our everyday lives are deeply intertwined with what happens online, whether it is checking news, playing games, or messaging friends on social media. In each of these examples, we are bringing what is happening in our world into the virtual space. For a change, I want to show you how games, filters, and even furniture stores bring the online space to stand right in front of you by exploring the world of augmented reality.

Augmented reality is when we overlay what we see in real life with things that clearly don't exist, Pokémon Go being an example you have likely seen or heard before, IKEA's retail furniture placement, and social media filters. The question is, how do these actually work? How do they know where the floor is and how to change the size based on perspective? Well, that's what we will be finding out today!

We'll start with the most difficult problem with augmented reality: mapping out our room. Since we can only rely on our camera sensor to map out the area, we can start by finding edges and corners. This is done by finding big changes in the colour gradient:

The equations

$$I_x = \frac{\partial I}{\partial x}, \quad I_y = \frac{\partial I}{\partial y}$$

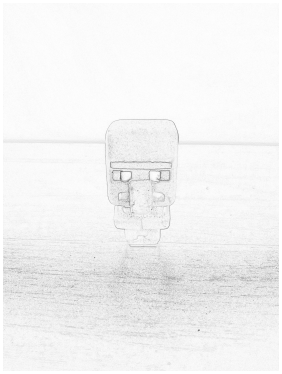
describes the change in gradient of the image brightness across the image horizontally and vertically. Say we have an image:

If I_x is < 0, the intensity is decreasing in the x-direction; if $I_x \approx 0$, then there has been no change locally; and if I_x > 0, then intensity is increasing. The same applies to I_y.

Combining these gives us the gradient magnitude:

$$|\nabla I| = \sqrt{I_x^2 + I_y^2}$$

which produces an image that highlights regions of strong intensity change. This allows us to identify the rough outlines of objects in the scene, giving us trackable feature points as well as information about the orientation of edges.



When we move around, so do these trackable points, which we will track the motion of via optical flow (specifically the Lucas-Kanade Method).

The key assumption we are making when calculating the pixel motion is that the brightness does not change instantly between frames, instead being a gradual

change. With this assumption, we can also assume that when one pixel moves to another place

$$I(x, y, t) \approx I(x + dx, y + dy, t + dt)$$

(where x and y are points on the new image and t is the change in time between the frames) that it is still the same point in the real world and has roughly the same brightness, and is just in a new position in the new frame.

Knowing this we can expand and approximate the right-hand side using a Taylor expansion ($f(x+dx) \approx f(x)+f'(x)dx$ - we can ignore $dx^2, dy^2, dxdy$, etc. since dx, dy, dt are already very small, that the rest is negligible):

$$I(x + dx, y + dy, t + dt) \approx I + I_x dx + I_y dy + I_t dt$$

Substituting this back into the original assumption, we get:

$$\begin{aligned} I(x, y, t) &= I(x, y, t) + I_x dx + I_y dy + I_t dt \\ \implies & I_x dx + I_y dy + I_t dt = 0 \end{aligned}$$

Dividing by dt:

$$\begin{aligned} & \frac{I_x dx + I_y dy + I_t dt}{dt} = 0 \\ \implies & I_x u + I_y v + I_t = 0 \end{aligned}$$

Where:

- $u = \frac{dx}{dt}$ →horizontal velocity
- $v = \frac{dy}{d}$ → vertical velocity
- I_x, I_y→ spatial gradients
- I_t → change over time

Which means that the motion of the pixel is constrained by how brightness changes in space and time and so if brightness is changing over time, it must be because the pixel is moving across a gradient.

In our new equation, we do have a slight issue, there are two unknowns! To solve this issue, instead of looking at only one pixel, we consider a small patch of pixels (following the Lucas-Kanade idea).

Now we have multiple equations:

$$I_{x1}u + I_{y1}v = -I_{t1}$$
$$I_{x2}u + I_{y2}v = -I_{t2}$$
$$...$$

Which we can rewrite in matrix form as:

$$A = \begin{pmatrix} I_{x1} & I_{y1} \\ I_{x2} & I_{y2} \\ \vdots & \vdots \end{pmatrix}, \mathbf{x} = \begin{pmatrix} u \\ v \end{pmatrix}, \mathbf{b} = \begin{pmatrix} -I_{t1} \\ -I_{t2} \\ \vdots \end{pmatrix}$$

Now we have an overdetermined system (we have more equations than we do unknowns), which means we cannot solve it directly. So instead of solving:

$$A\mathbf{x} = \mathbf{b}$$

we will instead be solving:

$$\mathbf{x}_{\min} = \arg\min_{\mathbf{x}} \|A\mathbf{x} - \mathbf{b}\|^2$$

Now this equation looks scary, so let's break it down. Earlier, we said that $I_x u + I_y v + I_t = 0$; however, in reality, there is always noise and lighting, so it is only ≈ 0. Our equation is trying to find the value of $\mathbf{x}$ that minimises this error as much as possible. The vector $A\mathbf{x} - \mathbf{b}$ finds us our error, and by squaring the result, it makes larger errors more undesirable regardless of whether it is positive or negative. Basically, all we are doing here is trying to find a compromise between all the equations for each pixel in the patch we chose.

This gives us the solution:

$$\mathbf{x} = (A^T A)^{-1} A^T \mathbf{b}$$

Assuming that $A^T A$ is invertible, also known as the linear least squares solution, which now allows the camera motion to be tracked.

What we have effectively built so far is a way to estimate how small image features move between frames. However, this motion is still only expressed in 2D image space. To turn this into something useful for augmented reality, we need to connect these 2D motions back to the 3D structure that produced them. In other words, we want to move from tracking pixels to understanding how the camera is moving through a real 3D world. Estimating pixel motion alone is not enough to build a stable augmented reality system. We also need to understand the 3D structure of the world, while simultaneously tracking the camera's position within it.

2 SLAM!

This is the idea behind SLAM: Simultaneous Localisation and Mapping. Instead of treating these as separate problems, SLAM solves them together. As the camera moves, it tracks distinctive feature points in the image (such as corners and edges) and follows how they shift across multiple frames. Over time, it identifies which observations correspond to the same physical points in the real world. From this, it gradually builds a sparse 3D map of the environment while also estimating the trajectory of the camera through space.

At each moment, the camera's position and orientation are represented by a transformation matrix:

$$T = \begin{pmatrix} R & \mathbf{t} \\ 0 & 1 \end{pmatrix}$$

Here $R \in SO(3)$, meaning R belongs to the group of all valid 3D rotation matrices (i.e. all matrices that rotate space without stretching or distorting it), and $\mathbf{t} \in R^3$ represents translation in 3D space.

A 3D point X in the world is projected into the image using:

$$\mathbf{x} \sim KTX$$

where K is the intrinsic camera matrix (which encodes properties like focal length and sensor geometry), T is the camera pose, X is a 3D point in homogeneous coordinates, and $\mathbf{x}$ is its corresponding 2D image location. The intrinsic camera matrix K describes the internal properties of the camera itself. In other words, how the camera converts a 3D direction into a 2D image. It does not depend on where the camera is in space (that is handled by T), instead it depends on the physical design of the camera and how it forms images.

A common form of K is:

$$K = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

Here:

- f_x, f_y are the focal lengths of the camera in pixel units, which control how "zoomed in" the image appears,
- c_x, c_y represents the principal point, which is usually close to the centre of the image,

- the final row ensures the use of homogeneous coordinates.

Intuitively, K describes how rays of light entering the camera are mapped onto the pixel grid of the image sensor.

PS: The symbol $\sim$ means "equal up to scaling" ($X = aY$ where a is some non-zero constant), because in image space we work in homogeneous coordinates - so two vectors that differ by a constant factor represent the same 2D point.

If the estimated camera pose is correct, then the projected 3D points will align with the feature points detected in the image. SLAM continuously refines its estimates so that this alignment error is minimised. Once the camera motion is known, virtual objects can be placed into the scene using the same transformation model. Because they follow the same geometry as real-world points, they remain fixed in space as the camera moves, creating the illusion of physical presence.

Even more interestingly, depth is not measured directly. Instead, it is inferred from motion. As the camera moves, the same point appears in different image positions. The amount of shift depends on distance - closer points move more across the image, while distant points move less. By combining many such observations, the system reconstructs the 3D structure of the scene from nothing more than a moving 2D camera. In effect, the system is solving an inverse problem: from a sequence of 2D images, it reconstructs both the 3D structure of the world and its own motion within it.

3 Now for our object

Now that we have gotten over the most difficult part, we need to actually be able to transform the 3D object in relation to our camera movement. To transform a three-dimensional object, we must use matrices with four rows:

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

We use homogeneous four-dimensional coordinates (adding an extra coordinate) for 3D objects because, while rotation and scaling can be represented using 3×3 matrices, translation cannot. This is because standard matrix transformations are linear, meaning that they must preserve the origin, so the point

(0, 0, 0) always remains fixed. Translation, however, moves every point in space equally and therefore does not preserve the origin, so it is not a linear transformation.

To include translation within a matrix framework, we embed 3D points into a higher-dimensional space by adding an extra coordinate, forming homogeneous coordinates. In this 4D representation, translation becomes representable as a matrix multiplication, just like rotation and scaling. After applying the transformation in this higher-dimensional space, we can project back into 3D to recover the translated object. Our transformation matrices are now:

$$\text{Translation} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}, \text{Scaling} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\text{Rotation}_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \text{Rotation}_y = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\text{Rotation}_z = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Now that we have both a 3D map of the environment and the camera's pose, we can finally place a virtual object into the scene. Suppose we define a virtual object using points in 3D space, for example, a point $P = (x, y, z, 1)^T$ in world coordinates. To display this point on the screen, we apply the same transformation used for real-world points:

$$\mathbf{p} \sim KTP$$

This projects the virtual point into the image, giving its position (p) on the screen, and as the camera moves, the pose matrix (T) updates continuously. Since the same transformation is applied to the virtual object, its projected position changes in exactly the same way as real-world points. This is what creates the illusion that the object is fixed in space. Even though it does not physically exist, it appears anchored to the environment because it follows the same geometric rules as everything else in the scene.

To place the object on a surface, such as the floor, we can use the estimated 3D map. For example, if a plane has been detected, we can define the position of the object relative to that plane. This ensures that the object sits correctly on the surface rather than floating or intersecting with it, which means that, finally, perspective naturally emerges from the projection process. Objects further away appear smaller because their projected coordinates depend on depth, while objects closer to the camera appear larger.

4 Always go further!

What first got me interested in computer vision was actually aim-bots and mouse-tracking. I was just curious about how they worked, and it led me here, so with this little space left, I want to just encourage anyone reading to just explore something basic, and maybe you'll find something interesting that you start seeing popping up everywhere (I just really like taking pictures of Pokémon on Pokémon Go). Overall, I hope you learned something new here, and if you did find this intriguing, please check out the bibliography, which covers everything here in much more detail and explains how to actually code AR systems!

5 Bibliography

- Optical Flow | Structure from Motion | Object Tracking
- Unity AR Augmented Reality Tutorials
- Math for Game Developers: Why do we use 4x4 Matrices in 3D Graphics?
- Essence of linear algebra
- The Math behind (most) 3D games - Perspective Projection