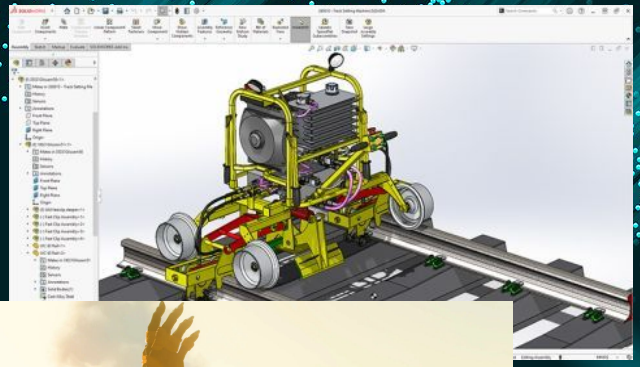
The Math Behind 3D Computer Graphics

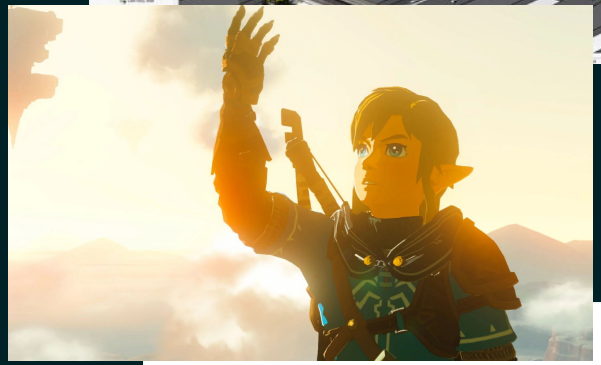
Tyler Pringle

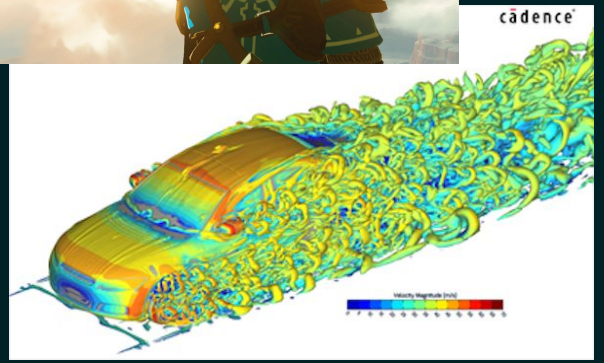
Why care about 3D graphics?

Beyond just being cool to look at, 3D computer graphics have a number of applications:

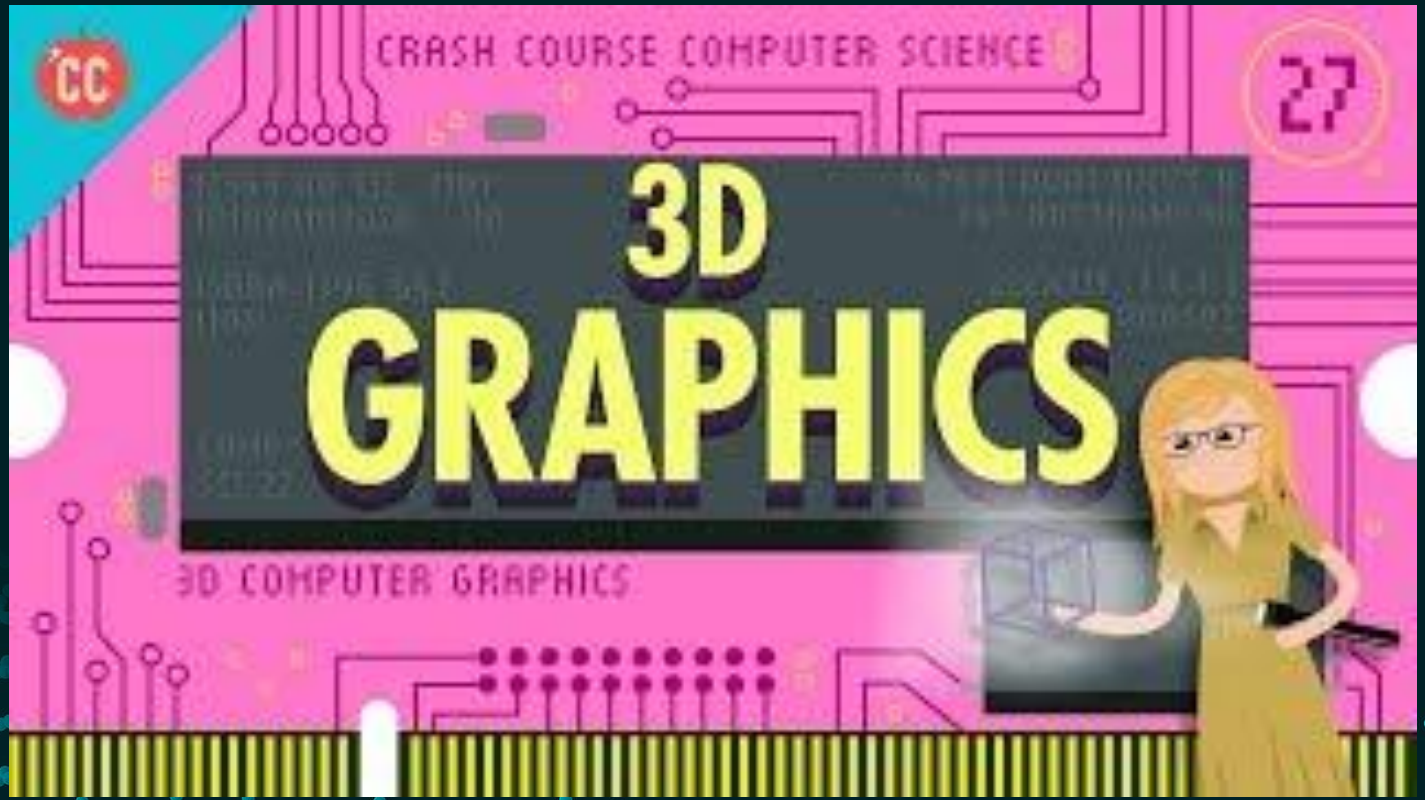
- Animation and visual effects
- Computer-aided design (CAD)
- 3D printing
- Video games
- Virtual and augmented reality
- Physics simulations





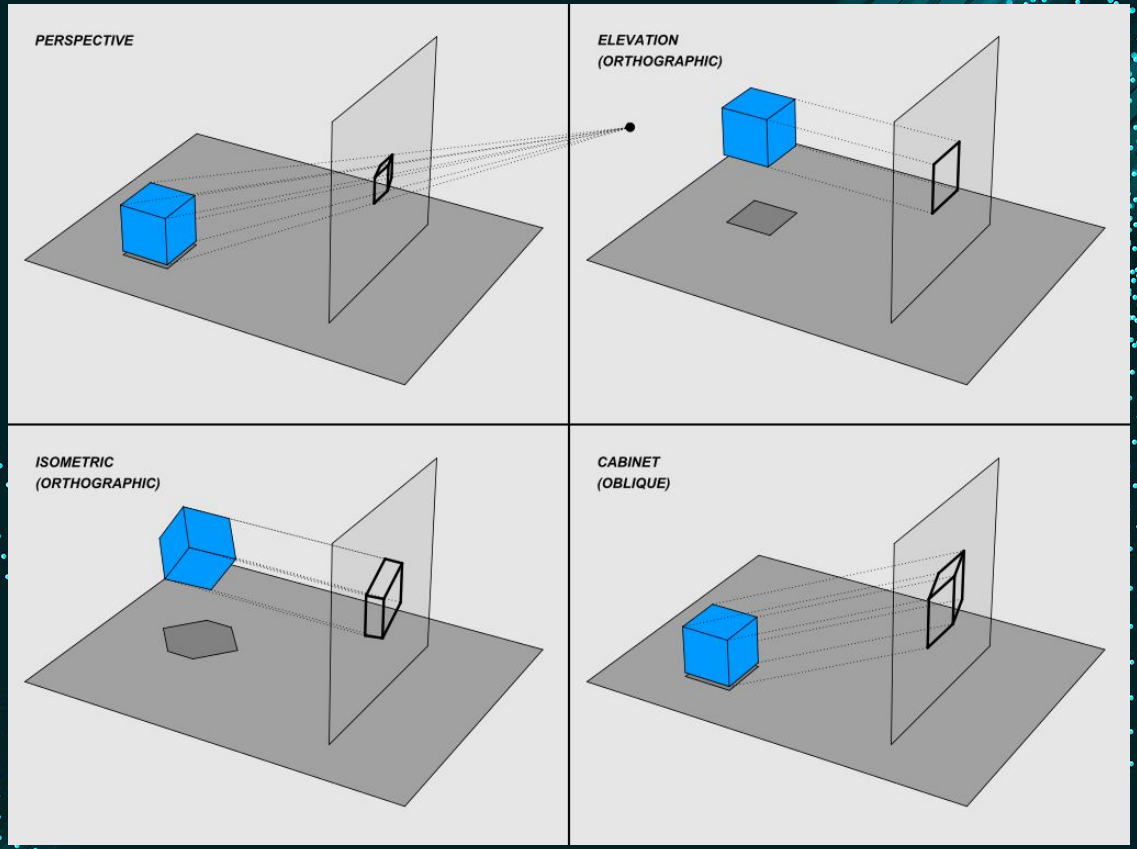


A Crash Course on 3D Graphics



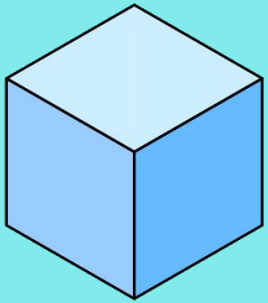
Projections

The fundamental principle behind 3D graphics

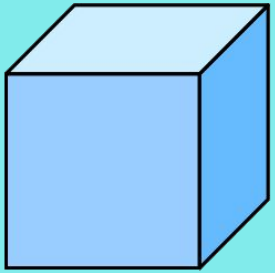


Projection Planes

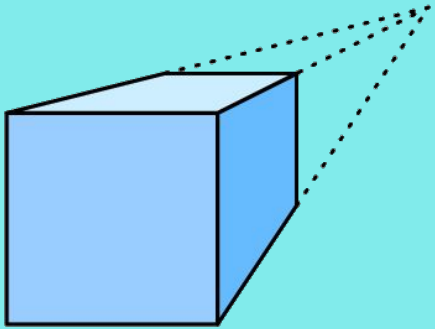
Three Main Types of Projections



ORTHOGRAPHIC



OBLIQUE



PERSPECTIVE

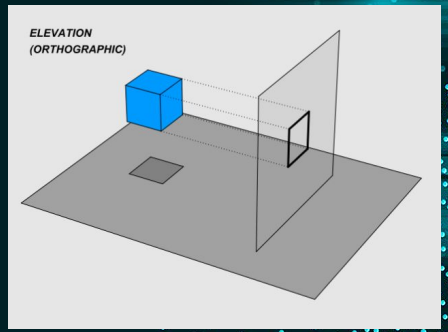
PARALLEL

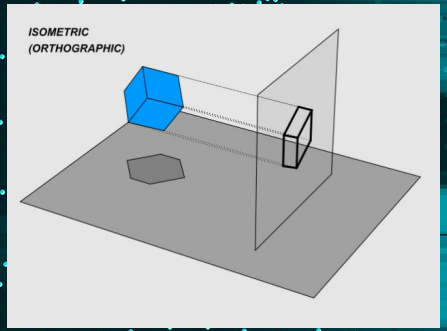
Orthographic Projections

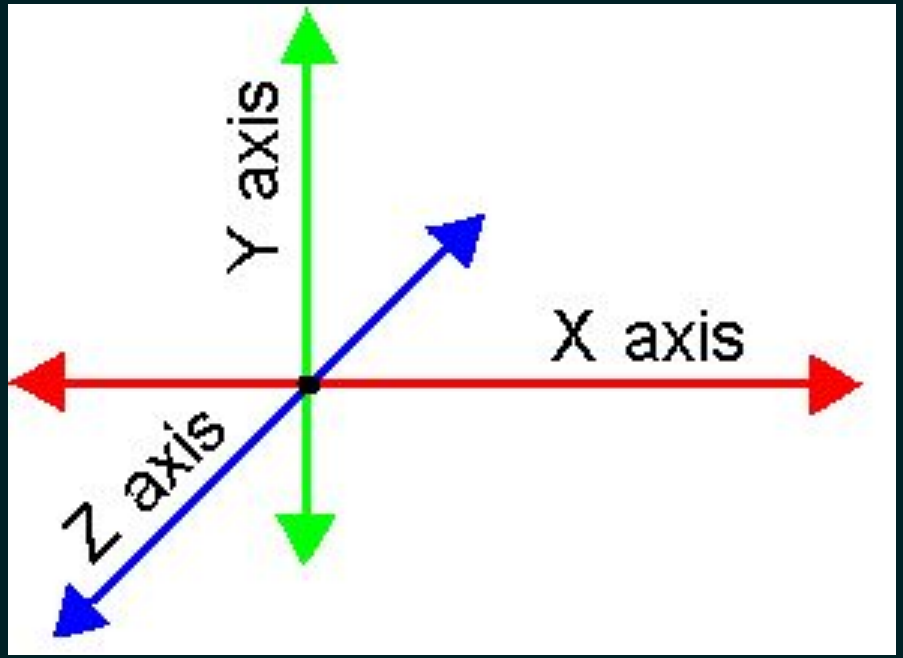
As a parallel projection, lines that are parallel in reality are still parallel in the projection.

In an orthographic projection, the projection lines are orthogonal to the projection plane.

A limitation of this kind of projection (and of parallel projections in general) is that objects do not appear larger as they extend closer to the camera, which can make the true depth and height of objects hard to visualize.







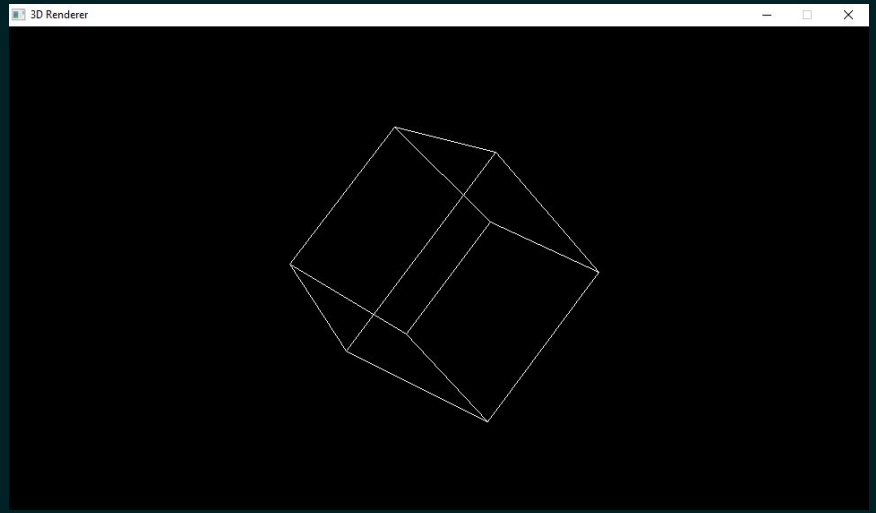
The axis orientation we'll be using

Perspective Projections

In this kind of projection, certain lines that are parallel in reality converge towards a single point (a vanishing point) in the image. Which lines appear to converge depends on the exact perspective of the viewer/camera.

There are two types of perspective projection: one that requires a more involved mathematical definition, and another, simpler type called weak perspective projection.

Creating a 3D renderer with these projections



based off of https://www.youtube.com/watch?v=nvWDgBGcAIM

```
// cube
std::vector<Point3D> points{ Point3D{-1.0f, -1.0f, -1.0f}, Point3D{-1.0f, -1.0f, 1.0f},
                             Point3D{1.0f, -1.0f, -1.0f}, Point3D{-1.0f, 1.0f, -1.0f},
                             Point3D{-1.0f, 1.0f, 1.0f}, Point3D{1.0f, -1.0f, 1.0f},
                             Point3D{1.0f, 1.0f, -1.0f}, Point3D{1.0f, 1.0f, 1.0f} };

std::vector<Edge> edges{ Edge{0, 1}, Edge{0, 2}, Edge{0, 3},
                         Edge{2, 5}, Edge{3, 6}, Edge{3, 4},
                         Edge{4, 7}, Edge{6, 7}, Edge{7, 5},
                         Edge{5, 1}, Edge{4, 1}, Edge{2, 6} };
```

Defining the vertices and edges between vertices of a 3D shape

```cpp
int WinMain(int argc, char** argv) {
    // create window and renderer
    SDL_Window* window;
    SDL_Renderer* renderer;
    window = SDL_CreateWindow("3D Renderer", SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED, 960, 540, 0);
    renderer = SDL_CreateRenderer(window, -1, SDL_RENDERER_ACCELERATED);

    bool running = true;

    // create a 3D renderer

    Renderer3D renderer3D1(window, renderer, points, edges);

    while (running) {
        if (SDL_QuitRequested()) {
            running = false;
            break;
        }
        renderer3D1.render();
    }

    return 0;
}
```

Setting up 3D renderer

```cpp
void Renderer3D::render() {
    auto time1 = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration(0);

    SDL_SetRenderDrawColor(renderer, 0, 0, 0, SDL_ALPHA_OPAQUE);
    SDL_RenderClear(renderer);
    SDL_SetRenderDrawColor(renderer, 255, 255, 255, SDL_ALPHA_OPAQUE);

    rotation += 1 * deltaTime;

    // rotates about all axes
    for (auto& edge : edges) {
        Point3D rotatedStartPoint = rotateX(rotateY(rotateZ(points[edge.start])));
        Point3D rotatedEndPoint = rotateX(rotateY(rotateZ(points[edge.end])));
        Point2D start = projection(rotatedStartPoint);
        Point2D end = projection(rotatedEndPoint);
        SDL_RenderDrawLine(renderer, start.x, start.y, end.x, end.y);
    }
```

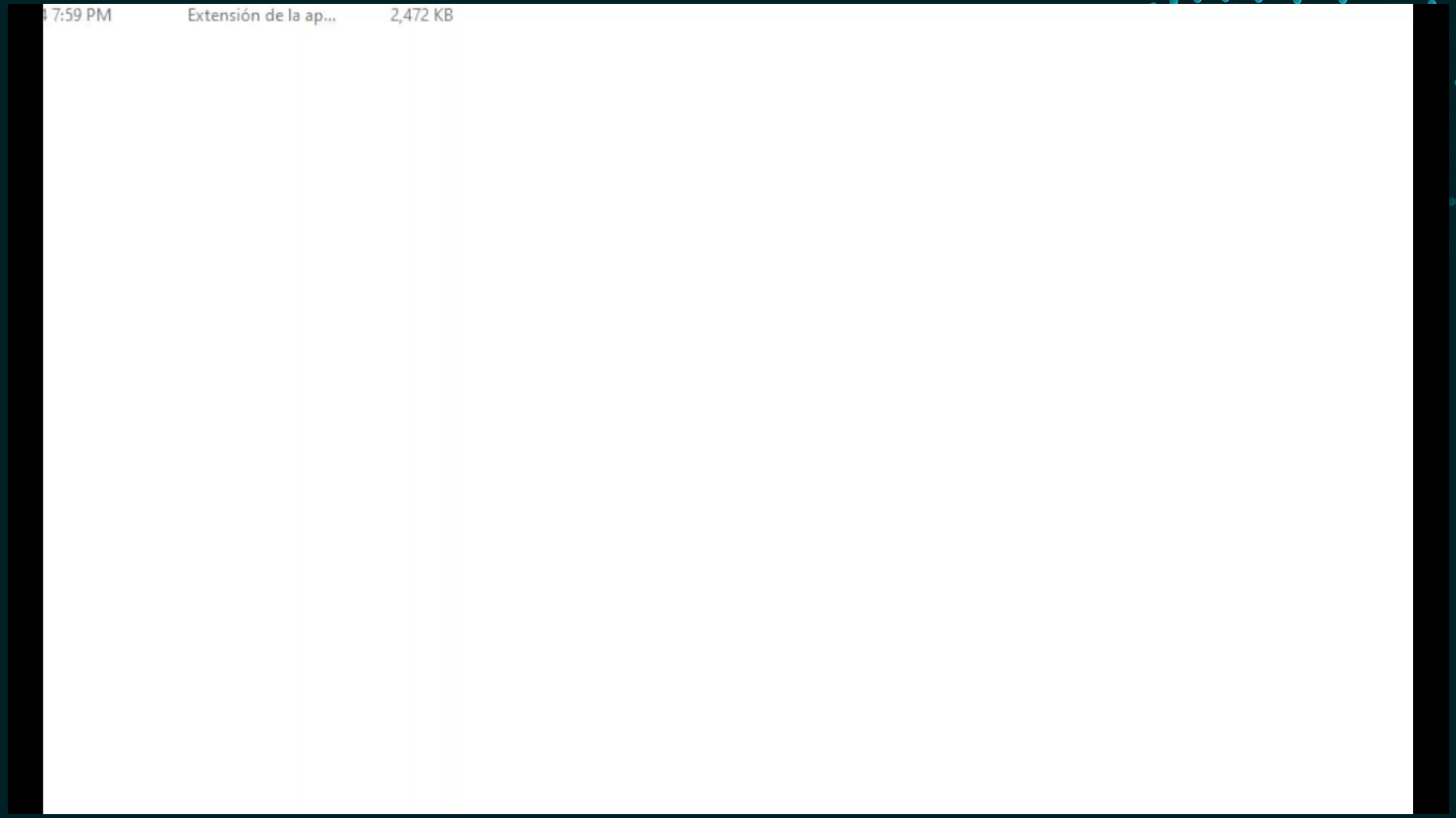
Main part of the render function

```
Point3D Renderer3D::rotateX(Point3D point) {
    Point3D returnPoint;
    returnPoint.x = point.x;
    returnPoint.y = cos(rotation) * point.y - sin(rotation) * point.z;
    returnPoint.z = sin(rotation) * point.y + cos(rotation) * point.z;
    return returnPoint;
}
```

Function that rotates a point about the x-axis

```
// orthographic projection
Point2D Renderer3D::projection(Point3D point) {
    return Point2D{ cameraViewpointX + (point.x * 100), cameraViewpointY + (point.y * 100) };
}
```

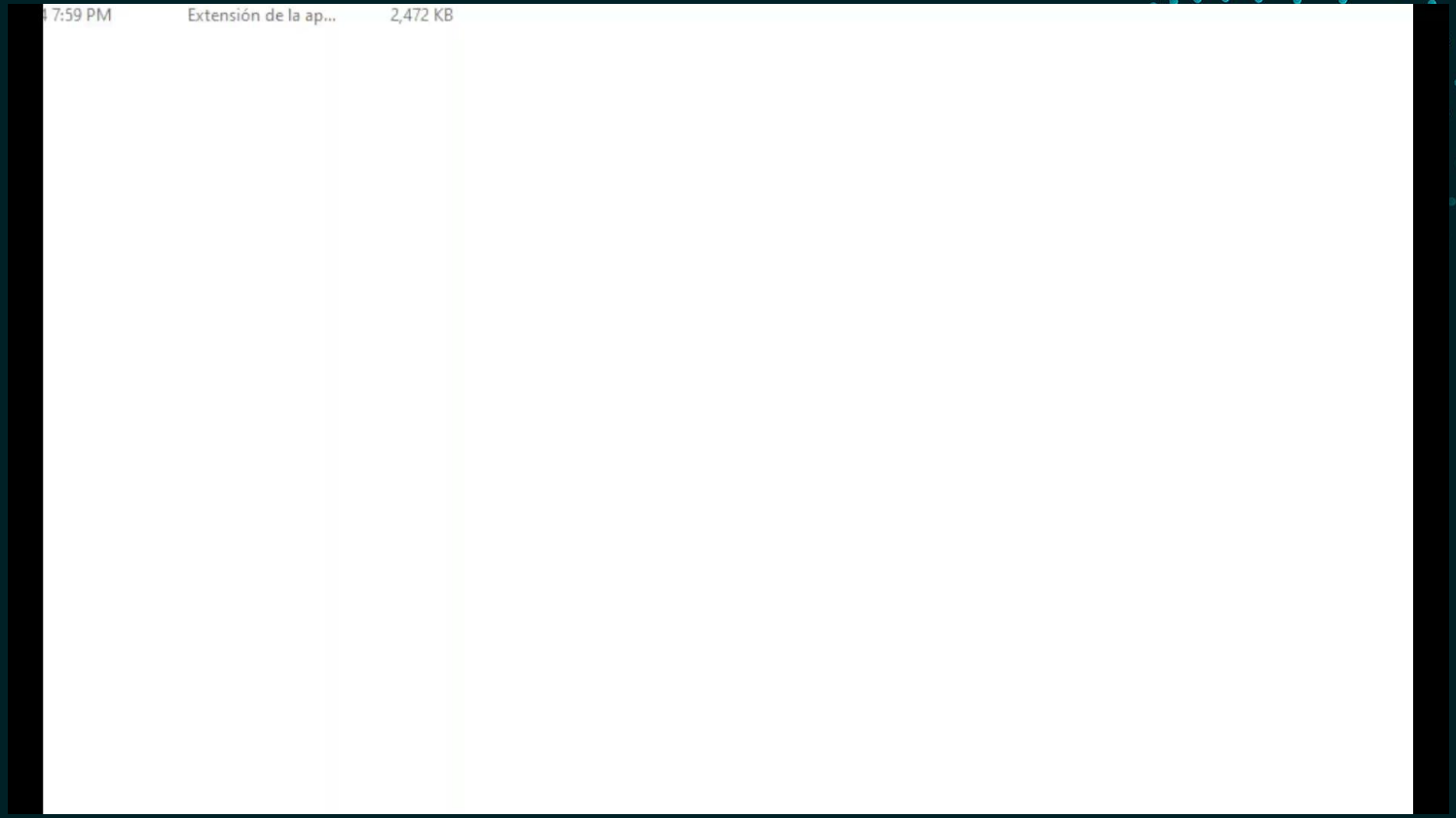
Code for orthographic projection of a 3D point



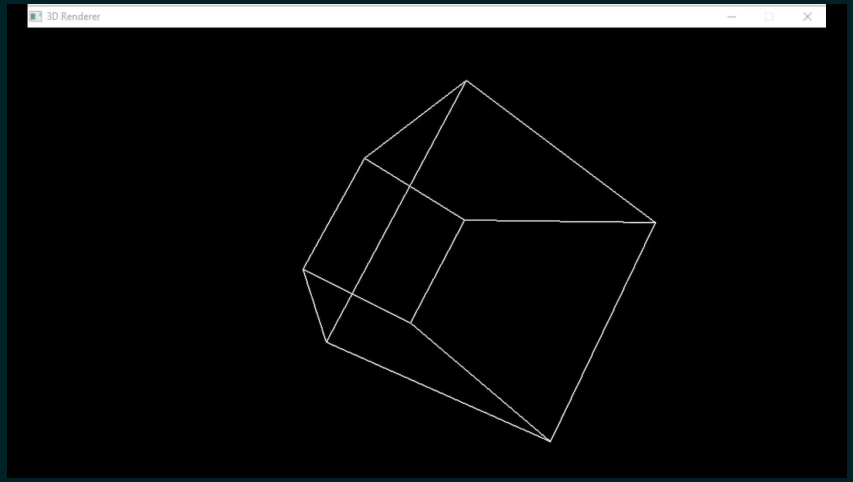
Wireframe render with orthographic projection

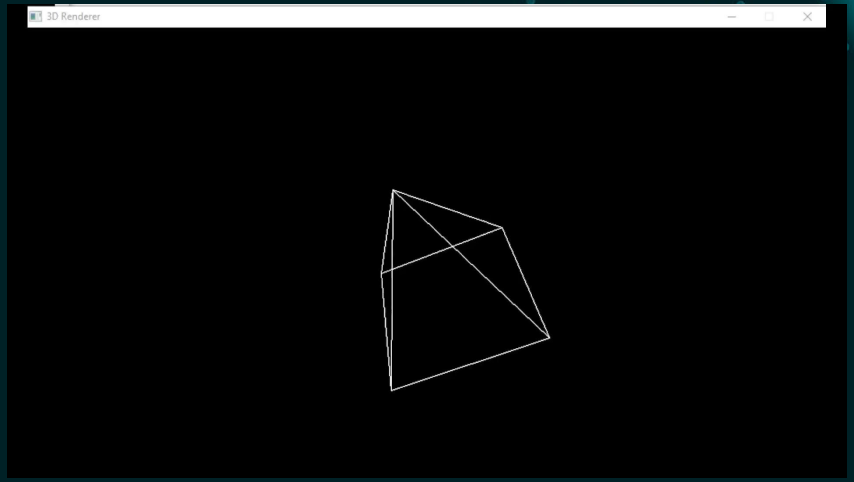
```
// weak perspective projection
Point2D Renderer3D::projection(Point3D point) {
    return Point2D{ cameraViewpointX + ((focalLength * point.x) / (focalLength - point.z)) * 100,
                    cameraViewpointY + ((focalLength * point.y) / (focalLength - point.z)) * 100 };
}
```

Code for weak perspective projection of a 3D point



Wireframe render with weak perspective projection





Wireframe renders of other shapes (with weak perspective projection)

How else is math applied to 3D graphics?

- Use of homogeneous coordinates (and projective geometry in general)
- Matrix multiplication for transforming points and lines
- Modeling curves and surfaces with splines
- Lighting, shading, and illumination using trigonometry and vector algebra

Sources

- https://www.cs.trinity.edu/~jhowland/class.files.cs357.html/blender/blender-stuff/m3d.pdf
- https://www.nxp.com/docs/en/application-note/AN4132.pdf
- https://www.youtube.com/watch?v=U0_ONQQ5ZNM
- https://www.youtube.com/watch?v=TEAtmCYYKZA (Crash Course)
- https://www.youtube.com/watch?v=nvWDgBGcAIM (simple 3D renderer tutorial)
- https://en.wikipedia.org/wiki/Homogeneous_coordinates
- https://en.wikipedia.org/wiki/3D_projection
- https://en.wikipedia.org/wiki/Perspective_(graphical)