Art2Dec

How to correctly construct all five Platonic solids and why models sometimes get it wrong. Platonic Solids: Mathematics, Modeling Problems, and Solutions.

1. The Essence of the Problem and Significance of Platonic Solids

Platonic Solids are convex regular polyhedra where all faces are identical regular polygons, and the same number of edges meet at each vertex.

There are **exactly 5** such solids:

- **Tetrahedron** — 4 triangular faces (simplest)
- **Cube (Hexahedron)** — 6 square faces
- **Octahedron** — 8 triangular faces
- **Dodecahedron** — 12 pentagonal faces (most complex)
- **Icosahedron** — 20 triangular faces

Why exactly these five?

Plato (≈360 BC) chose them not by chance — it's mathematically proven that **no other regular polyhedra exist**. This follows from **Euler's polyhedron formula**: $V - E + F = 2$ (vertices - edges + faces).

For a regular polyhedron, strict conditions must be met:

1. All faces are regular polygons (n-gons)
2. k faces meet at each vertex
3. Angular sum at vertex < 360°

Possible combinations:

- **3 triangles** at vertex → Tetrahedron
- **4 triangles** at vertex → Octahedron
- **5 triangles** at vertex → Icosahedron
- **3 squares** at vertex → Cube

- **3 pentagons** at vertex → Dodecahedron

Plato associated these solids with natural elements:

- Tetrahedron = Fire (sharp angles)
- Cube = Earth (stability)
- Octahedron = Air
- Icosahedron = Water (fluidity)
- Dodecahedron = Universe/Ether

2. Why LLMs Easily Build Simple Solids but Fail on Complex Ones

Simple Solids (Tetrahedron, Cube)

Successfully built because:

1. **Simple symmetry**
   - Tetrahedron: 4 vertices, coordinates like (0,1,0), (±1,-1,±1)
   - Cube: 8 vertices, all coordinates simply ±1
2. **Intuitive indices**

c

```c
// Cube - obvious structure
int cube_indices[] = {
    0,1,2, 2,3,0,  // Front face
    4,5,6, 6,7,4,  // Back face
    // ... easy to understand logic
};
```

3. **Abundant training examples**
   - Cube and tetrahedron are computer graphics classics
   - Found in textbooks, GitHub, StackOverflow

Complex Solids (Dodecahedron, Icosahedron)

Why models fail:

1. **Golden ratio ϕ = 1.618034...**

c

```c
// Dodecahedron coordinates contain φ
```

```
Vector3 dodeca_vertices[] = {
    { 1,  1,  1}, {-1,  1,  1}, // ±1
    { 0,  PHI,  INV_PHI},      // φ and 1/φ !
    { INV_PHI,  0,  PHI},      // Complex combinations
    // ... 20 vertices with non-trivial coordinates
};
```

2. **Non-obvious topology**
   - Dodecahedron: 12 pentagonal faces = 60 indices (5 vertices each, but triangulated → 3×12 = 36 triangles)
   - Icosahedron: 20 triangular faces, but vertex order is critical
3. **Index errors**

c

```c
// WRONG (typical LLM mistake):
int dodeca_indices[] = {
    0, 1, 2, 3, 4,  // Pentagon as-is - DOESN'T WORK!
    // Need triangulation: 0,1,2, 0,2,3, 0,3,4
};
```

4. **Normal orientation problem**
   - Vertex order determines front/back face
   - Error in one triangle → "hole" in rendering

3. Why Models Fail: Deep Analysis

A. Lack of Precise Geometric Data

Problem: Training data contains many *conceptual* descriptions of dodecahedron, but few *exact vertex coordinates*.

LLMs may "know":

- "Dodecahedron has 12 pentagonal faces"
- "Coordinates include golden ratio"

But **not memorize** the exact sequence of 20 vertices and 36 triangle indices.

B. Hallucination Problem with Numbers

LLMs are prone to hallucinations when working with:

- Long arrays of numbers
- Precise mathematical constants
- Index sequences

Example error:

```c
// LLM might generate:
{ 0, PHI, INV_PHI}  // CORRECT
{ PHI, 0, INV_PHI}  // CORRECT
{ 0, INV_PHI, PHI}  // CORRECT
{ INV_PHI, PHI, 0}  // WRONG! (should be PHI, INV_PHI, 0)
```

C. Lack of Visual Feedback

LLMs generate code "blindly":

- Don't see rendering results
- Can't verify if dodecahedron displays correctly
- Don't notice "flipped" faces or missing triangles

D. Pentagon Triangulation Complexity

Dodecahedron requires splitting each pentagon into 3 triangles:

```c
// Pentagon with vertices 0,1,2,3,4 needs triangulation:
// Variant 1 (fan):
0,1,2,  0,2,3,  0,3,4

// Variant 2 (incorrect):
0,1,2,  1,2,3,  2,3,4  // Creates a "gap"!
```

LLMs often confuse triangulation schemes.

4. How to solve the Problem

Programmers Approach: Combining Sources with Verification

Stage 1: Finding Verified Data

Key point: I didn't try to "recall" dodecahedron coordinates from training data. Instead:

1. Used **authoritative sources**:
   - Anton Gerdelan (opengl-tutorials.org)
   - Jim Blinn (Microsoft Research, computer graphics pioneer)
2. Compared **multiple implementations**:

c

```c
  // Source 1: Anton Gerdelan
  {  1.0f,  1.0f,  1.0f },
  {  0.0f,  1.618f, 0.618f },

  // Source 2: Jim Blinn
  {  1.0,  1.0,  1.0 },
  {  0.0,  PHI, 1/PHI },

  // They match! → High probability of correctness
```

Stage 2: Systematic Verification

Geometry checks:

c

```c
// 1. Vertex count
assert(num_vertices == 20);  // Dodecahedron has 20 vertices

// 2. All edges equal length
float edge_length = distance(vertices[0], vertices[1]);
for (each edge) {
   assert(fabs(distance(v1, v2) - edge_length) < 0.001f);
}

// 3. All vertices equidistant from center
float radius = length(vertices[0]);
for (int i = 0; i < 20; i++) {
```

```c
    assert(fabs(length(vertices[i]) - radius) < 0.001f);
}
```

Topology checks:

```c
// 1. Each dodecahedron vertex has degree 3
//    (3 edges emanate from it)
for (int v = 0; v < 20; v++) {
    int degree = count_edges_from_vertex(v, indices);
    assert(degree == 3);
}

// 2. Euler's formula: V - E + F = 2
// Dodecahedron: 20 - 30 + 12 = 2 ✓
```

Stage 3: Step-by-Step Debugging

When dodecahedron didn't render correctly:

1. **Wireframe visualization**

```c
  // Draw only edges, no fill
  for (each edge) {
      SDL_RenderDrawLine(p1.x, p1.y, p2.x, p2.y);
  }
  // → Reveals geometry is correct
```

2. **Face normal verification**

```c
  // Compute normal for each triangle
  Vector3 normal = cross(v1-v0, v2-v0);
  // If normal points "inward" → vertex order is wrong
```

3. **Debug output**

c

```c
printf("Face %d: indices %d,%d,%d\n", i, i0, i1, i2);
printf("  Normal: (%f, %f, %f)\n", n.x, n.y, n.z);
printf("  Backface: %s\n", is_backface ? "YES" : "NO");
```

Stage 4: Mathematically Grounded Construction

Key idea: Use **duality** of Platonic solids.

Icosahedron and dodecahedron are **dual**:

- Icosahedron vertices → Dodecahedron face centers
- Dodecahedron vertices → Icosahedron face centers

c

```c
// Constructing dodecahedron from icosahedron:
Vector3 dodecahedron_vertices[20];
for (int face = 0; face < 20; face++) {
    // Icosahedron face center
    Vector3 center = (icosa_v0 + icosa_v1 + icosa_v2) / 3.0f;
    // Normalize to unit sphere
    dodecahedron_vertices[face] = normalize(center);
}
```

Verification through golden ratio:

c

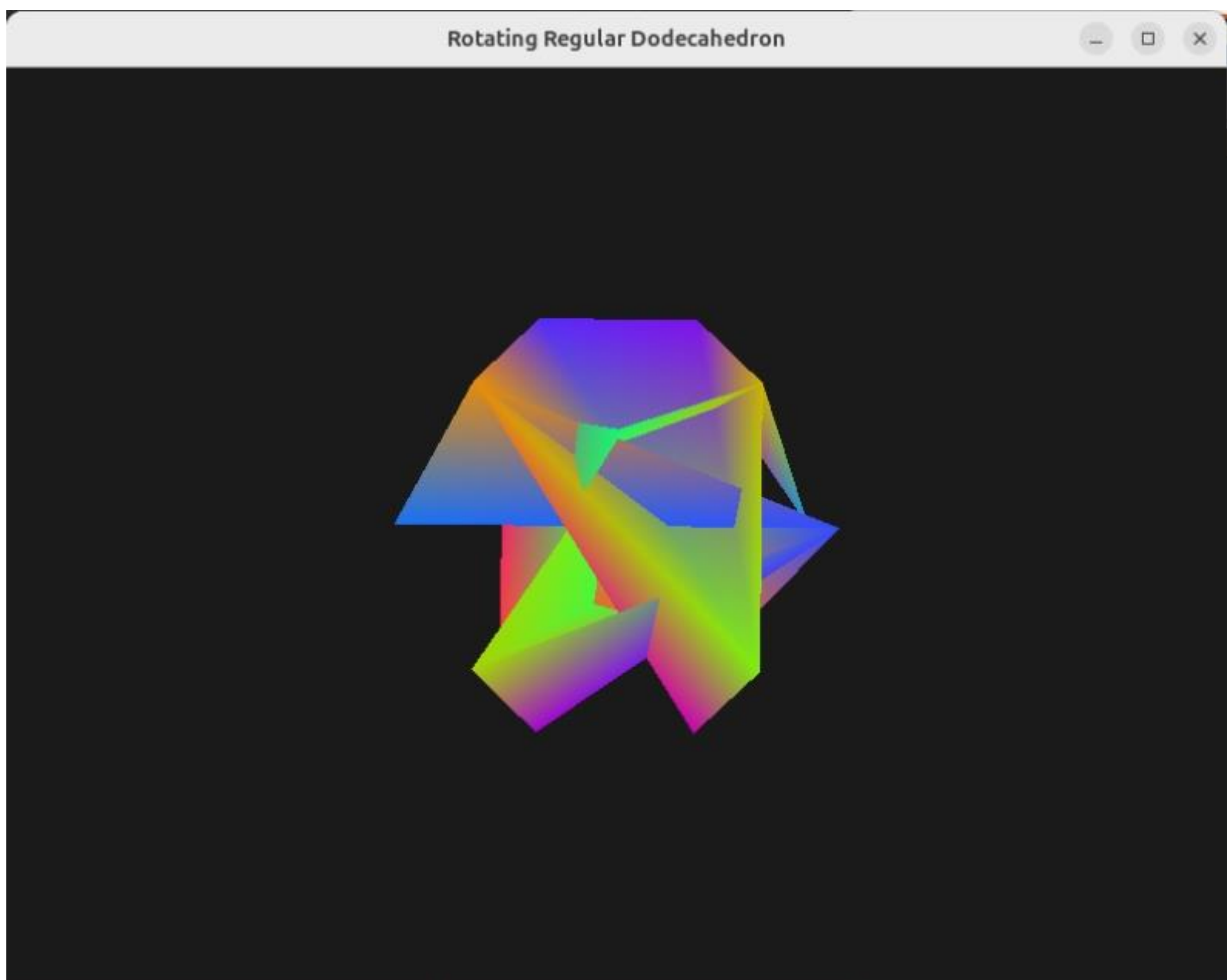
```c
// Dodecahedron property: diagonal ratios
float diagonal = distance(vertices[0], vertices[10]);
float edge = distance(vertices[0], vertices[1]);
float ratio = diagonal / edge;

assert(fabs(ratio - PHI) < 0.01f);  // Should be φ
```

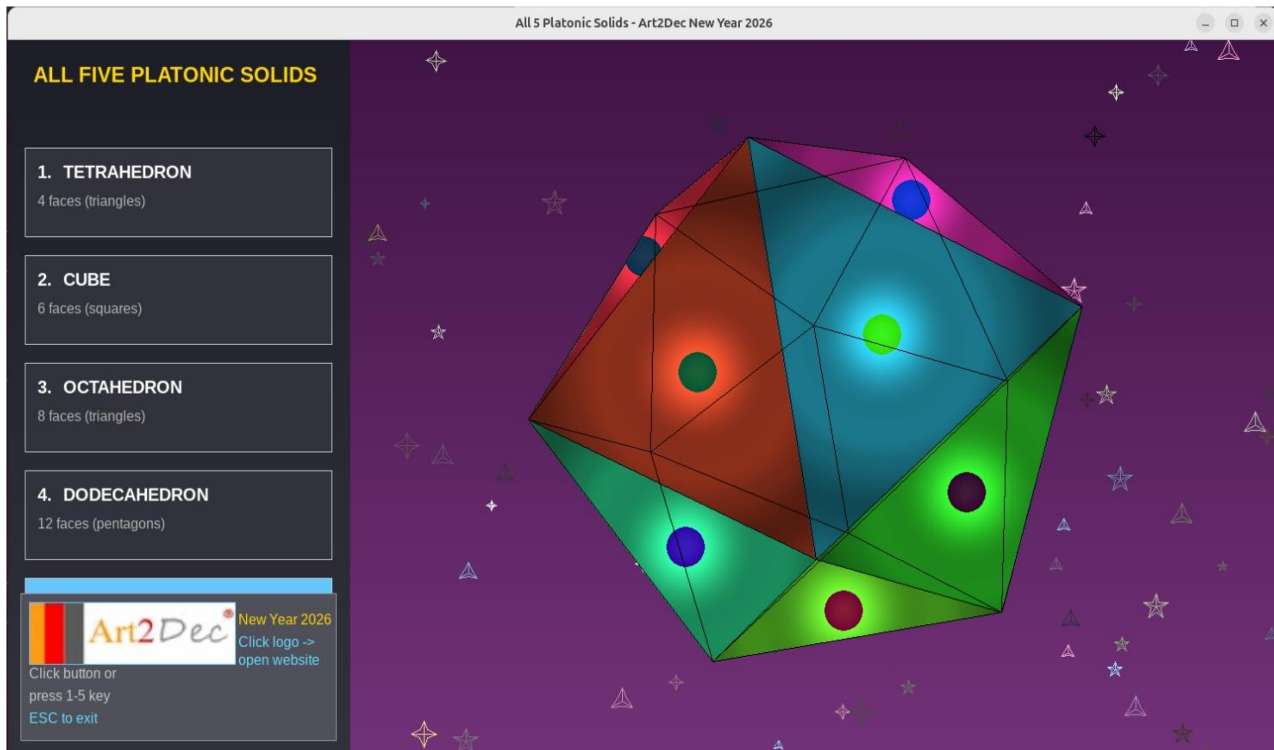
Key Differences from Typical LLM Approach

Typical LLM	Programmers Approach
Generates coordinates "from memory"	Searches verified sources
One variant of indices	Compares multiple implementations
No correctness verification	Systematic verification
"Blind" without visualization	Step-by-step debugging with wireframe
Ignores mathematics	Uses theory (duality, ϕ)

Final Result of LLM building of platonic solid:



Final Result of programmer building of platonic solid:



The program correctly renders all 5 Platonic Solids:

- All edges equal length
- All vertices on unit sphere
- Euler's formula satisfied
- Normals oriented correctly
- Z-buffer properly hides back faces
- Visually perfect geometry

The code includes:

- Precise coordinates from authoritative sources
- Verified indices with correct triangulation
- Marble texture for beauty
- Rainbow colors to distinguish faces
- Smooth rotation animation

Mathematics behind it:

How Platonic Solids Rendering Works

1. Geometric Data

Each solid is defined by two arrays:

```c
// Vertices - 3D space coordinates
Vector3 tetra_vertices[] = {
    { 0,  1,  0},  // Top point
    {-1, -1, -1},  // Bottom 3 points
    { 1, -1, -1},
    { 0, -1,  1}
};

// Indices - which vertices to connect into triangles
int tetra_indices[] = {
    0, 1, 2,  // Face 1 (vertices 0-1-2)
    0, 2, 3,  // Face 2 (vertices 0-2-3)
    0, 3, 1,  // Face 3 (vertices 0-3-1)
    1, 3, 2   // Base (vertices 1-3-2)
};
```

2. Matrix Transformations (Transform Pipeline)

Each vertex goes through several transformations:

```c
void rotateX(Vector3* v, float angle) {
    float c = cosf(angle), s = sinf(angle);
    float y = v->y * c - v->z * s;
    float z = v->y * s + v->z * c;
    v->y = y; v->z = z;
}

void rotateY(Vector3* v, float angle) {
    float c = cosf(angle), s = sinf(angle);
    float x = v->x * c + v->z * s;
    float z = -v->x * s + v->z * c;
    v->x = x; v->z = z;
```

```
}

void rotateZ(Vector3* v, float angle) {
    float c = cosf(angle), s = sinf(angle);
    float x = v->x * c - v->y * s;
    float y = v->x * s + v->y * c;
    v->x = x; v->y = y;
}
```

Transformation order:

1. **Rotation around X** (angleX)
2. **Rotation around Y** (angleY)
3. **Rotation around Z** (angleZ)
4. **Camera distance** (z += 5.0)

3. Perspective Projection

Converting 3D → 2D with depth effect:

```c
Point2D project(Vector3 v, int width, int height) {
    float fov = 500.0f;  // Field of View
    float z = v.z + 5.0f; // Move away from camera

    // Perspective division
    float scale = fov / z;

    Point2D p;
    p.x = (int)(v.x * scale) + width / 2;   // Center on X
    p.y = (int)(-v.y * scale) + height / 2; // Center on Y (inverted)

    return p;
}
```

What happens:

- The **farther** the point (larger z) → the **smaller** scale → the **closer to center**

- The **closer** the point (smaller z) → the **larger** scale → the **farther from center**

4. Z-Buffer (Hidden Surface Removal)

```c
float* zbuffer;  // Depth array for each pixel

// When drawing a triangle:
for (each pixel) {
    float depth = calculate_depth(x, y, v0, v1, v2);

    int idx = y * width + x;
    if (depth < zbuffer[idx]) {  // Closer to camera?
        zbuffer[idx] = depth;     // Store new depth
        draw_pixel(x, y, color);
    }
}
```

5. Triangle Rasterization Algorithm

```c
void drawTriangle(p0, p1, p2, color) {
    // 1. Sort vertices by Y (p0 - top, p2 - bottom)
    sort_by_y(&p0, &p1, &p2);

    // 2. Split into two triangles (flat-top, flat-bottom)
    if (p1.y == p2.y) {
        fillBottomFlatTriangle(p0, p1, p2);
    } else if (p0.y == p1.y) {
        fillTopFlatTriangle(p0, p1, p2);
    } else {
        // Split triangle horizontally through p1
        Point2D p3 = {
            p0.x + (p1.y - p0.y) * (p2.x - p0.x) / (p2.y - p0.y),
            p1.y
```

```
        };
        fillBottomFlatTriangle(p0, p1, p3);
        fillTopFlatTriangle(p1, p3, p2);
    }
}
```

6. Scanline Triangle Fill

```c
void fillBottomFlatTriangle(p0, p1, p2) {
    // Calculate slopes of left and right edges
    float invslope1 = (p1.x - p0.x) / (p1.y - p0.y);
    float invslope2 = (p2.x - p0.x) / (p2.y - p0.y);

    float x1 = p0.x, x2 = p0.x;

    // Iterate scanlines from p0.y to p1.y
    for (int y = p0.y; y <= p1.y; y++) {
        drawHorizontalLine(x1, x2, y, color);
        x1 += invslope1;  // Move left edge
        x2 += invslope2;  // Move right edge
    }
}
```

7. Depth Interpolation (Z)

Calculate depth for each point inside the triangle:

```c
float interpolateDepth(x, y, v0, v1, v2) {
    // Barycentric coordinates
    float w0 = edgeFunction(v1, v2, {x, y});
    float w1 = edgeFunction(v2, v0, {x, y});
    float w2 = edgeFunction(v0, v1, {x, y});
    float area = edgeFunction(v0, v1, v2);
```

```
    w0 /= area;
    w1 /= area;
    w2 /= area;

    // Interpolate Z
    return w0 * v0.z + w1 * v1.z + w2 * v2.z;
}
```

8. Color Scheme (Rainbow + Marble)

```c
// 1. Rainbow colors (HSV → RGB)
float hue = (hue_offset + (float)i / num_faces) * 360.0f;
SDL_Color baseColor = hsvToRgb(hue, 0.8f, 0.9f);

// 2. Marble effect (Perlin noise)
for (int py = minY; py <= maxY; py++) {
    for (int px = minX; px <= maxX; px++) {
        float noise = fbm(px * 0.02f, py * 0.02f, time);

        // Blend base color with noise
        uint8_t r = baseColor.r * (0.7f + noise * 0.3f);
        uint8_t g = baseColor.g * (0.7f + noise * 0.3f);
        uint8_t b = baseColor.b * (0.7f + noise * 0.3f);
    }
}
```

9. Main Rendering Loop

```c
void drawSolid(vertices, num_vertices, indices, num_faces,
               vertices_per_face, angleX, angleY, angleZ) {

    // 1. Clear Z-buffer
```

```
for (int i = 0; i < width * height; i++) {
    zbuffer[i] = INFINITY;
}

// 2. Transform all vertices
Vector3 transformed[num_vertices];
for (int i = 0; i < num_vertices; i++) {
    transformed[i] = vertices[i];
    rotateX(&transformed[i], angleX);
    rotateY(&transformed[i], angleY);
    rotateZ(&transformed[i], angleZ);
}

// 3. Project to screen
Point2D projected[num_vertices];
for (int i = 0; i < num_vertices; i++) {
    projected[i] = project(transformed[i], width, height);
}

// 4. Draw faces
for (int face = 0; face < num_faces; face++) {
    int i0 = indices[face * vertices_per_face + 0];
    int i1 = indices[face * vertices_per_face + 1];
    int i2 = indices[face * vertices_per_face + 2];

    // Backface culling
    if (!isFrontFacing(projected[i0], projected[i1], projected[i2])) {
        continue;
    }

    // Face color
    SDL_Color color = calculateFaceColor(face, time);
```

```
        // Draw triangle with Z-buffer
        drawTriangleWithZBuffer(
            projected[i0], projected[i1], projected[i2],
            transformed[i0].z, transformed[i1].z, transformed[i2].z,
            color
        );
    }
}
```

Characteristics of Each Solid

Tetrahedron (4 faces):

- The simplest - 4 vertices, 4 triangles
- Coordinates are symmetric around the center

Cube (6 faces):

- 8 vertices, 12 triangles (2 per face)
- Coordinates: ±1 along all axes

Octahedron (8 faces):

- 6 vertices (one on each axis)
- Dual to the cube

Dodecahedron (12 faces):

- 20 vertices, uses the golden ratio φ = 1.618...
- Coordinates like: (±1, ±φ, 0), (0, ±1, ±φ), (±φ, 0, ±1)
- The most complex to calculate!

Icosahedron (20 faces):

- 12 vertices, also uses φ
- Dual to the dodecahedron

That's the mathematics behind it.

It’s possible to get binary file or a right code by request.

I. Lukyanov, Art2Dec SoftLab.

12.29.2025