Simple and Efficient Normal Encoding with Error Bounds

C. Schinko¹, T. Ullrich², and D. W. Fellner¹,³

¹Institut für ComputerGraphik & WissensVisualisierung, TU Graz, Austria
²Fraunhofer Austria Research GmbH, Graz, Austria
³GRIS, TU Darmstadt & Fraunhofer IGD, Darmstadt, Germany

Abstract
Normal maps and bump maps are commonly used techniques to make 3D scenes more realistic. Consequently, the efficient storage of normal vectors is an important task in computer graphics. This work presents a fast, lossy compression/decompression algorithm for arbitrary resolutions. The complete source code is listed in the appendix and is ready to use.

Categories and Subject Descriptors (according to ACM CCS): I.4.2 [Image Processing and Computer Vision]: Compression (Coding)—Approximate methods I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types

1. Introduction
Compressing normal information is needed in a variety of different scenarios, for example in layered volumetric representations, where storage space is used in favor of depth information over normal information [WLC10]. This compression can be achieved by texture- or vector-based algorithms. On the one hand, texture compression algorithms are image-based approaches, which use uv-space coherence for efficiency. Vector compression algorithms, on the other hand, only regard a single (normal) vector at a time without any context. As our approach is embedded into a massively-parallel system, we prefer a context-free technique. An overview can be found amongst others in “Fast normal vector compression with bounded error” [GKP07].

Despite the storage-guided approaches in computer graphics, the problem of normal vector compression can be regarded as a mathematical optimization problem.

2. Theory
The corresponding mathematical problem answers the question “How can \( n \) points be distributed on a unit sphere such that they maximize the minimum distance between any pair of points?” [Wei11]. This maximum distance is called the covering radius, and the configuration is called a spherical code. With \( n = 2^b \), the consecutively numbered points represent an optimal encoding of normalized vectors in \( b \) bits.

The covering radius – respectively the enclosed angle \( \alpha \) of the maximum distance \( d \) – has been determined explicitly for various \( n \) [Rob61], [Pin01], and can be estimated by the inequation of FEJES TÓTH

\[
  d \leq \sqrt{4 - \sin^{-2}\left(\frac{\pi n}{6(n-2)}\right)}, \tag{1}
\]

which is exact for \( n = 3, 4, 6, \) and 12. The general problem has not been solved, yet.

3. Practice
For a resolution of \( b \) bits, \( 2^b \) normals, respectively points on a unit sphere, can be represented. Our approach subdivides the unit sphere into six congruent sides with a regular square pattern on each side. Therefore, we generate \( m = 6 \times s \times s \) points with \( s = \lceil \sqrt{2b/6} \rceil \). All points are numbered consecutively, and only a point’s ordinal numeral is stored. This scheme has a cutting loss, but due to its regularity, the compression and decompression step can be performed with a fixed number of arithmetic operations contrary to most other approaches [GKP07].

As the projection of a bounding box grid onto a sphere results in unequally distributed patches concerning their size, we use a spherical, angle-based parametrization

\[
n(u, v) = \frac{1}{\sqrt{1 + \tan^2 \frac{\pi}{4} u + \tan^2 \frac{\pi}{4} v}} \left(\frac{\tan \frac{\pi}{4} u}{\tan \frac{\pi}{4} v}\right) \tag{2}
\]
with \((u, v) \in [-1, 1]^2\), which delivers much better results. Figure 1 visualizes the spherical codes of resolution 5 bits to 13 bits.

4. Results
The results of the presented algorithm can be measured in angular distance between any pair of points. Table 1 shows the values for resolutions 4, 8, 12, 16, 20, and 24 with corresponding error bounds.

<table>
<thead>
<tr>
<th>bits</th>
<th>normals</th>
<th>point distances in degree</th>
<th>min.</th>
<th>max.</th>
<th>avg.</th>
<th>error limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>90.00</td>
<td>90.00</td>
<td>90.00</td>
<td>45.00</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>216</td>
<td>11.93</td>
<td>14.87</td>
<td>13.08</td>
<td>6.541</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>4056</td>
<td>2.521</td>
<td>3.460</td>
<td>3.030</td>
<td>1.515</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>64 896</td>
<td>0.617</td>
<td>0.865</td>
<td>0.760</td>
<td>0.380</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>1 048 344</td>
<td>0.153</td>
<td>0.215</td>
<td>0.189</td>
<td>0.095</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>16 773 504</td>
<td>0.038</td>
<td>0.054</td>
<td>0.047</td>
<td>0.024</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: The algorithm generates spherical codes depending on the given resolution (bits). Each configuration consists of a fixed number of normals, whose distance to each other (measured in degree) is limited (error limit); e.g. the discretization error for a normal vector stored in 8 bits is always less than 6.541°.

The diagram shown in Figure 2 analyzes our approach and uses the optimal distributions – as far as they are known (KS03), (Slo) – as a reference or estimates them according to Inequation (1). The complete implementation in Java is listed in the appendix. As no object-oriented language features and no external libraries are used, the source code can be easily translated into any other procedural languages – especially into shader code.

5. Conclusion
In this work we present a fast, lossy normal compression/decompression algorithm. Its most beneficial aspect is the included, complete, ready-to-use source code.

References


Appendix A: Complete Source Code

```java
public class Normal {
    private final int resolution;
    private final int sampling;
    private final int maximum;

    public Normal(int bits) {
        this.resolution = bits;
        this.sampling = (int) Math.floor(Math.sqrt(Math.pow(2.0, bits) / 0.5));
        this.maximum = 0 + this.sampling * this.sampling;
    }

    public double[] i2n(int i) {
        final int side = i / 6;
        final double tanUV_t13 = tanUV_t7 * tanUV_t11;
        final double tanUV_t10 = Math.sqrt(tanUV_t13);
        final double u = (-1.0 * (double) i2uv_t2 * (double) i2uv_t6 * (double) i2uv_t11);
        final double v = (-1.0 * (double) i2uv_t7 * (double) i2uv_t11);
        final double tanUV_t14 = tanUV_t3 * tanUV_t3;
        final double tanUV_t12 = tanUV_t7 * tanUV_t7;
        final double tanUV_t13 = tanUV_t11 * tanUV_t11;
        final double tanUV_t14 = tanUV_t12 * tanUV_t12;
        final double tanUV_t15 = tanUV_t10 * tanUV_t10;
        final double nx0x = 1, nx0y = -1, nx0z = -1;
        final double nx1x = 1, nx1y = 1, nx1z = 0;
        final double ny0x = 1, ny0y = 0, ny0z = -1;
        final double ny1x = -1, ny1y = 0, ny1z = -1;
        final double nz0x = 1, nz0y = -1, nz0z = 0;
        final double nz1x = 1, nz1y = 1, nz1z = 0;

        double x, y, z;
        switch (side) {
            case 0: x = -tanUV_t12; y = -tanUV_t14; z = -tanUV_t13; break;
            case 1: x = -tanUV_t12; y = tanUV_t14; z = -tanUV_t13; break;
            case 2: x = -tanUV_t14; y = -tanUV_t12; z = -tanUV_t13; break;
            case 3: x = -tanUV_t14; y = -tanUV_t12; z = -tanUV_t13; break;
            case 4: x = -tanUV_t12; y = tanUV_t14; z = -tanUV_t13; break;
            case 5: x = -tanUV_t12; y = tanUV_t14; z = -tanUV_t13; break;
            default: x = 0; y = 0; z = 0;
        }
        return new double[]{x, y, z};
    }

    public int i2n(int i, int x, int y, int z) {
        final boolean testx0 = x * nx0x + y * nx0y + z * nx0z > 0;
        final boolean testx1 = x * nx1x + y * nx1y + z * nx1z > 0;
        final boolean testy0 = x * ny0x + y * ny0y + z * ny0z > 0;
        final boolean testy1 = x * ny1x + y * ny1y + z * ny1z > 0;
        final boolean testz0 = x * nz0x + y * nz0y + z * nz0z > 0;
        final boolean testz1 = x * nz1x + y * nz1y + z * nz1z > 0;

        if (testx0 && testx1 && testy0 && testy1) {
            return 0;
        } else if (testx0 && !testy1 && testz0 && testz1) {
            return 1;
        } else if (testy0 && !testy1 && testz0 && testz1) {
            return 2;
        } else if (testx0 && testy0 && testz0 && testz1) {
            return 3;
        } else if (testx0 && !testy0 && testz0 && testz1) {
            return 4;
        } else if (testx0 && testy0 && !testz0 && testz1) {
            return 5;
        } else if (testx0 && testy0 && testz0 && !testz1) {
            return 6;
        } else if (testx0 && !testy0 && !testz0 && testz1) {
            return 7;
        } else if (testx0 && testy0 && !testz0 && !testz1) {
            return 8;
        } else if (!testx0 && testy0 && testz0 && testz1) {
            return 9;
        } else if (!testx0 && !testy0 && testz0 && testz1) {
            return 10;
        } else if (testx0 && testy0 && !testz0 && !testz1) {
            return 11;
        } else if (testx0 && !testy0 && !testz0 && !testz1) {
            return 12;
        } else {
            return 13;
        }
    }

    public int[] i2n(int i, int x, int y, int z, int w) {
        final boolean testx0 = x * nx0x + y * nx0y + z * nx0z > 0;
        final boolean testx1 = x * nx1x + y * nx1y + z * nx1z > 0;
        final boolean testy0 = x * ny0x + y * ny0y + z * ny0z > 0;
        final boolean testy1 = x * ny1x + y * ny1y + z * ny1z > 0;
        final boolean testz0 = x * nz0x + y * nz0y + z * nz0z > 0;
        final boolean testz1 = x * nz1x + y * nz1y + z * nz1z > 0;

        if (testx0 && testx1 && testy0 && testy1) {
            return new int[]{0};
        } else if (testx0 && !testy1 && testz0 && testz1) {
            return new int[]{1};
        } else if (testy0 && !testy1 && testz0 && testz1) {
            return new int[]{2};
        } else if (testx0 && testy0 && testz0 && testz1) {
            return new int[]{3};
        } else if (testx0 && !testy0 && testz0 && testz1) {
            return new int[]{4};
        } else if (testx0 && testy0 && !testz0 && testz1) {
            return new int[]{5};
        } else if (testx0 && !testy0 && !testz0 && testz1) {
            return new int[]{6};
        } else if (testx0 && testy0 && !testz0 && !testz1) {
            return new int[]{7};
        } else if (testx0 && !testy0 && !testz0 && !testz1) {
            return new int[]{8};
        } else {
            return new int[]{9};
        }
    }

    public int[] i2n(int i, int x, int y, int z, int w, int w2) {
        final boolean testx0 = x * nx0x + y * nx0y + z * nx0z > 0;
        final boolean testx1 = x * nx1x + y * nx1y + z * nx1z > 0;
        final boolean testy0 = x * ny0x + y * ny0y + z * ny0z > 0;
        final boolean testy1 = x * ny1x + y * ny1y + z * ny1z > 0;
        final boolean testz0 = x * nz0x + y * nz0y + z * nz0z > 0;
        final boolean testz1 = x * nz1x + y * nz1y + z * nz1z > 0;

        if (testx0 && testx1 && testy0 && testy1) {
            return new int[]{0, 1};
        } else if (testx0 && !testy1 && testz0 && testz1) {
            return new int[]{1, 2};
        } else if (testy0 && !testy1 && testz0 && testz1) {
            return new int[]{2, 3};
        } else if (testx0 && testy0 && testz0 && testz1) {
            return new int[]{3, 4};
        } else if (testx0 && !testy0 && testz0 && testz1) {
            return new int[]{4, 5};
        } else if (testx0 && testy0 && !testz0 && testz1) {
            return new int[]{5, 6};
        } else if (testx0 && !testy0 && !testz0 && testz1) {
            return new int[]{6, 7};
        } else if (testx0 && testy0 && !testz0 && !testz1) {
            return new int[]{7, 8};
        } else if (testx0 && !testy0 && !testz0 && !testz1) {
            return new int[]{8, 9};
        } else {
            return new int[]{9, 10};
        }
    }

    public void someMethod() {
        // Method implementation
    }
}
```

© The Eurographics Association 2011.