EG UK Theory and Practice of Computer Graphics (2011)
Hamish Carr, Ian Grimstead (Editors)

Simple and Efficient Normal Encoding with Error Bounds

C. Schinko[1], T. Ullrich[2], and D. W. Fellner[1,3]
[1]Institut für ComputerGraphik & WissensVisualisierung, TU Graz, Austria
[2]Fraunhofer Austria Research GmbH, Graz, Austria
[3]GRIS, 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 α 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 $s = \left\lfloor \sqrt{2^b/6} \right\rfloor$. 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

$$n(u,v) = \frac{1}{\sqrt{1 + \tan^2\frac{\pi}{4}u + \tan^2\frac{\pi}{4}v}} \begin{pmatrix} \tan\frac{\pi}{4}u \\ \tan\frac{\pi}{4}v \\ 1 \end{pmatrix} \tag{2}$$

DOI: 10.2312/LocalChapterEvents/TPCG/TPCG11/063-065

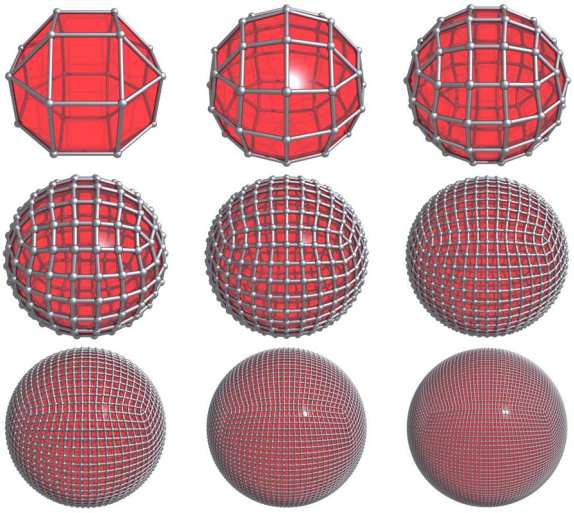

Figure 1: *Within a fixed resolution the compression scheme can represent only a limted number of normals / points on a unit sphere. These configurations are visualized as tesselations of 24 vertices (5 bits resolution, top-left) up to 7 776 vertices (13 bits resolution, bottom-right). This visualization shows the high degree of regularity and the almost equidistant spacing at all levels of resolution.*

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.

bits	normals	point distances in degree			
		min.	max.	avg.	error limit
4	6	90.00	90.00	90.00	45.00
8	216	11.93	14.87	13.08	6.541
12	4 056	2.521	3.460	3.030	1.515
16	64 896	0.617	0.865	0.760	0.380
20	1 048 344	0.153	0.215	0.189	0.095
24	16 773 504	0.038	0.054	0.047	0.024

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 the are known

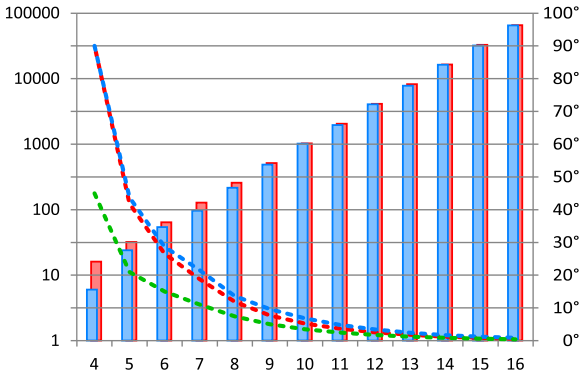

Figure 2: *The diagram shows the number of normals (left axis) and the angular errors (right axis) for each level of resolution. Red bars represent the maximum number of normals, whereas the blue ones correspond to the number of generated normals. The difference results from cutting loss. The red line shows the angular quality of the optimal normal distribution [KS03], [Slo] in contrast to our results (blue line). The corresponding discretization error is visualized in green.*

[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

[GKP07] GRIFFITH E. J., KOUTEK M., POST F. H.: Fast normal vector compression with bounded error. *Proceedings of the fifth Eurographics symposium on Geometry processing* (2007), 263–272. 1

[KS03] KATANFOROUSH A., SHAHSHAHANI M.: Distributing Points on a Sphere. *Experimental Mathematics 12* (2003), 199–208. 2

[Pin01] PINTER J. D.: Globally Optimized Spherical Point Arrangements: Model Variants and Illustrative Results. *Annals of Operations Research 104* (2001), 213–230. 1

[Rob61] ROBINSON R. M.: Arrangement of 24 Points on a Sphere. *Math. Annalen 144* (1961), 17–48. 1

[Slo] SLOANE N. J. A.: Spherical Codes. online: http://www2.research.att.com/∼njas/packings/. 2

[Wei11] WEISSTEIN E. W.: Spherical code. online: http://mathworld.wolfram.com/SphericalCode.html, 2011. 1

[WLC10] WANG C. C. L., LEUNG Y.-S., CHEN Y.: Solid modeling of polyhedral objects by layered depth-normal images on the gpu. *Computer Aided Design 42* (2010), 535–544. 1

Appendix A: Complete Source Code

```
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, this.resolution) / 6));
        this.maximum    = 6 * this.sampling * this.sampling;
    }

    public double[] i2n(int i) {
        final int    i2uv_t1   = i / 6;
        final int    i2uv_t2   = i2uv_t1 / this.sampling;
        final double i2uv_t3   = 1.0 / this.sampling;
        final int    i2uv_t7   = i2uv_t1 % this.sampling;
        final int    i2uv_t11  = i % 6;
        final double u = -0.10e1 + 0.20e1 * (double) i2uv_t2 * (double) i2uv_t3 + (double) i2uv_t3;
        final double v = -0.10e1 + 0.20e1 * (double) i2uv_t7 * (double) i2uv_t3 + (double) i2uv_t3;
        final int    w = i2uv_t11;

        final double tanUV_t3  = Math.tan(Math.PI * u / 0.4e1);
        final double tanUV_t4  = tanUV_t3 * tanUV_t3;
        final double tanUV_t7  = Math.tan(Math.PI * v / 0.4e1);
        final double tanUV_t8  = tanUV_t7 * tanUV_t7;
        final double tanUV_t10 = Math.sqrt(0.1e1 + tanUV_t4 + tanUV_t8);
        final double tanUV_t11 = 0.1e1 / tanUV_t10;
        final double tanUV_t12 = tanUV_t3 * tanUV_t11;
        final double tanUV_t13 = tanUV_t7 * tanUV_t11;
        final double tanUV_t14 = 0.10e1 * tanUV_t11;

        double x, y, z;
        switch (w) {
            case 0: x = -tanUV_t12; y = -tanUV_t13; z =  tanUV_t14; break;
            case 1: x = -tanUV_t12; y = -tanUV_t13; z = -tanUV_t14; 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 n2i(double x, double y, double z) {
        final double nx0x =  0, nx0y =  1, nx0z = -1;
        final double nx1x =  0, nx1y = -1, nx1z = -1;
        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;

        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;

        final int side;
        if      ( testx0 &&  testx1 &&  testy0 &&  testy1) { side = 1; }
        else if (!testx0 && !testx1 && !testy0 && !testy1) { side = 0; }
        else if ( testy0 &&  testy1 &&  testz0 &&  testz1) { side = 2; }
        else if (!testy0 &&  testy1 && !testz0 &&  testz1) { side = 3; }
        else if (!testx0 &&  testx1 &&  testz0 && !testz1) { side = 5; }
        else { side = 4; }

        double scale;
        switch (side) {
            case 0: scale =  1 / z;  break;
            case 1: scale = -1 / z;  break;
            case 2: scale =  1 / x;  break;
            case 3: scale = -1 / x;  break;
            case 4: scale =  1 / y;  break;
            case 5: scale = -1 / y;  break;
            default: scale = 0;
        }
        final double sx = scale * x, sy = scale * y, sz = scale * z;
        final double u, v;
        switch (side) {
            case 0: case 1:
                u = (-4 / Math.PI) * Math.atan(sx);
                v = (-4 / Math.PI) * Math.atan(sy);
                break;
            case 2: case 3:
                u = (-4 / Math.PI) * Math.atan(sy);
                v = (-4 / Math.PI) * Math.atan(sz);
                break;
            case 4: case 5:
                u = (-4 / Math.PI) * Math.atan(sx);
                v = (-4 / Math.PI) * Math.atan(sz);
                break;
            default: u = 0; v = 0;
        }

        final int p = Math.max(0, Math.min(sampling-1,
                (int) Math.round(0.5 * (this.sampling - 1 + u * this.sampling))));
        final int q = Math.max(0, Math.min(sampling-1,
                (int) Math.round(0.5 * (this.sampling - 1 + v * this.sampling))));
        final int r = (p * this.sampling + q) * 6 + side;
        return r;
    }
}
```