Volume 122, Article No. 25 (2017) https://doi.org/10.6028/jres.122.025
Journal of Research of the National Institute of Standards and Technology

# *A Sampling-Agnostic Software Framework for Converting Between Texture Map Representations of Virtual Environments*

**Vinay K Sriram**[1, 2] **and Wesley Griffin**[1]

[1]National Institute of Standards and Technology,
Gaithersburg, MD 20899, USA

[2]Stanford University,
Stanford, CA 94305, USA

vsriram@stanford.edu
wesley.griffin@nist.gov

**Software DOI:** https://doi.org/10.18434/M3P88M

**Software Version:** 1.0

**Key words:** equirectangular; octahedral; paraboloid; projection; sampling; texture map; transformation.

**Accepted:** May 10, 2017

**Published:** May 18, 2017

https://doi.org/10.6028/jres.122.025

## 1. Summary

We have developed a utility to both stitch cube maps into other types of texture maps (equirectangular, dual paraboloid, and octahedral), and stitch those other types back into cube maps. The utility allows for flexibility in the image size of the conversion - the user can specify the desired image width, and the height is computed (cube, paraboloid, and octahedral mappings are square, and spherical maps are generated to have 16:9 aspect ratio). Moreover, the utility is sampling-agnostic, so the user can select whether to use uniform or jittered sampling over the pixels, as well as the number of samples to use per pixel. The rest of this paper discusses the mathematical framework for projecting from cube maps to equirectangular, dual paraboloid, and octahedral environment maps, as well as the mathematical framework for the inverse projections. We also describe two sampling techniques: uniform sampling and correlated multi-jittered sampling. We perform an evaluation of the sampling techniques and a comparative analysis of the different projections using objective image quality assessment metrics.

## 2. Software Specifications

| | |
|---|---|
| **NIST Operating Unit(s)** | ITL, ACMD |
| **Category** | Image Projection and Sampling |
| **Targeted Users** | Virtual Reality Developers |
| **Operating System(s)** | Linux |
| **Programming Language** | C++11 |

| | |
|---|---|
| **Inputs/Outputs** | Cube map images for input and single projected environment map images on output (or vice versa). |
| **Documentation** | https://github.com/usnistgov/cubemap-stitch |
| **Accessibility** | N/A |
| **Disclaimer** | https://www.nist.gov/director/licensing |

## 3. Projecting from Cube Maps

A cube map may be thought of as the set of six images that six 90-degree field-of-vision cameras would capture if placed in orthogonal directions to one another at a fixed distance from the origin observer. The six faces are: up, down, left, right, floor, and ceiling. Figure 1 illustrates how the cubic texture map is assembled from each of these image captures. The cubic texture map can equivalently be thought of as a projection of the unit sphere onto the cube faces when they are arranged correctly in three dimensions. In other words, each point $P$ on one of the faces is assigned the color of some point $Q$ on the surface of the environment's unit sphere such that $P$, $Q$, and the origin are collinear.

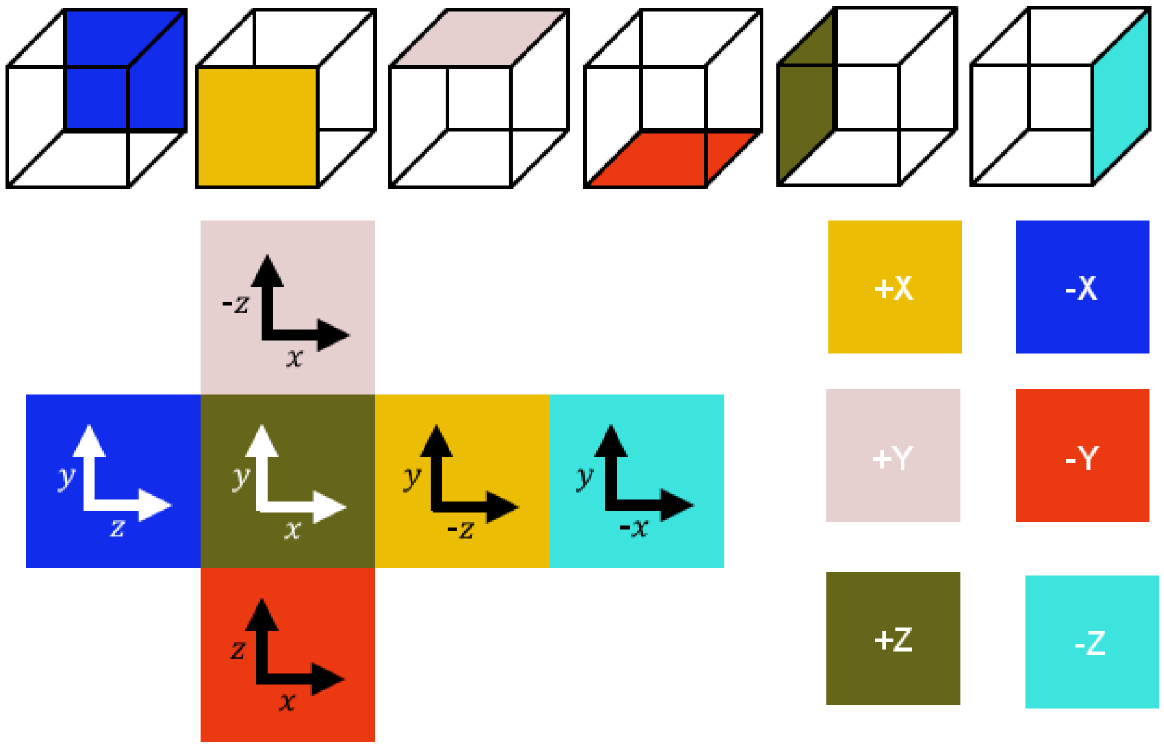

**Fig. 1.** Cube map texture layout and assembly from the six sides of a cube.

When converting a cube map, we seek to populate some other $n$-pixel by $m$-pixel texture map in which each pixel is assigned an $(R, G, B)$ coloring[1]. We determine the color coordinates pixel-by-pixel; consider a single pixel in this map $(i, j) \in \{0, \dots, n-1\} \times \{0, \dots, m-1\}$. We sample a set of points $S = \{P_1, \dots, P_k\}$ from within this pixel using one of the techniques in Sec. 5. For each of these points $P_l$, we compute the

[1] In practice, $R$, $G$, and $B$ each represent an 8-bit data type describing a color intensity; each intensity is given as an integer between 0 and 255 (inclusive).

point on the unit sphere surface corresponding to the pixel. This computation varies depending on the type of texture map we wish to generate, and is addressed in more detail in Sec. 3.1, 3.2, and 3.3. Once the unit vector is obtained, we can apply the algorithm described in Listing 1 to determine the cube face and pixel on the face corresponding to it. We extract the coloring from this pixel: $(R_l, G_l, B_l)$. Once this process is repeated for all points in the set $S$, the pixel of interest $(i, j)$ is simply assigned the following average:

$$(R, G, B)_{(i,j)} = \left(\sum_{l=1}^{k} \frac{R_l}{k}, \sum_{l=1}^{k} \frac{G_l}{k}, \sum_{l=1}^{k} \frac{B_l}{k}\right). \quad (1)$$

Once this process is repeated for all pixels $(i, j)$, our texture map conversion is complete. Note that in some cases, a single texture map image is insufficient. For instance, in the case of dual paraboloid (Sec. 3.2) or octahedral (Sec 3.3) projections, we require two texture maps. In such cases, the unit vector is computed not only given the point $P_l$ but also the index of whichever of the two images the point was sampled from.

```
convertUnitRayToCubeFacePixel(float x, float y, float z, int &i, int &j, int &n)
{
        float xMag = abs(x);
        float yMag = abs(y);
        float zMag = abs(z);

        if ((xMag > yMag && xMag > zMag) || (xMag <= yMag && yMag <= zMag)) {

                if (p.x < 0.0) {
                        n = 2; // Back Face
                        i = (int) (((-z/xMag) + 1.0)*input_width/2);
                        j = (int) (((y/xMag) + 1.0)*input_height/2);
                } else {
                        n = 3; // Front Face
                        i = (int) (((z/xMag) + 1.0)*input_width/2);
                        j = (int) (((y/xMag) + 1.0)*input_height/2);
                }

        } else if (xMag > yMag && xMag <= zMag) {

                if (p.z < 0.0) {
                        n = 0; // Left Face
                        i = (int) (((x/zMag) + 1.0)*input_width/2);
                        j = (int) (((y/zMag) + 1.0)*input_height/2);
                } else {
                        n = 1; // Right Face
                        i = (int) (((-x/zMag) + 1.0)*input_width/2);
                        j = (int) (((y/zMag) + 1.0)*input_height/2);
                }

        } else if(xMag <= yMag && yMag > zMag) {

                if (p.y < 0.0) {
                        n = 5; // Ceiling Face
                        i = (int) (((x/yMag) + 1.0)*input_width/2);
                        j = (int) (((-z/yMag) + 1.0)*input_height/2);
                } else {
                        n = 4; // Floor Face
                        i = (int) (((x/yMag) + 1.0)*input_width/2);
                        j = (int) (((z/yMag) + 1.0)*input_height/2);
                }

        }
}
```

Listing 1: Conversion algorithm for a unit ray to a cube face and pixel.

## 3.1 Equirectangular Projection

Here we address how to determine the unit vector $(x_v, y_v, z_v)$ for the texture map point $P_l = (x, y)$ in an equirectangular projection [1]. Recall that an equirectangular texture map is simply a planar representation of a unit sphere in which latitude is encoded as length and longitude is encoded as height. Therefore, the point on the unit sphere that corresponds to our texture point $(x, y)$ is given in spherical coordinates as:

$$\begin{cases} \theta = x\pi. \\ \varphi = \frac{y\pi}{2}. \end{cases} \tag{2}$$

Figure 2 depicts the spherical coordinates of a unit vector in $\mathbb{R}^3$. Note that a unit vector in three-dimensions can be uniquely determined by two spherical coordinates $(\theta, \phi)$; we can apply the standard transformation between spherical and Euclidean space to obtain the unit vector $(x_v, y_v, z_v)$ from spherical coordinates:

$$\begin{cases} x_v = \cos(\phi)\cos(\theta)\,. \\ y_v = \sin(\phi)\,. \\ z_v = \cos(\phi)\sin(\theta)\,. \end{cases} \tag{3}$$

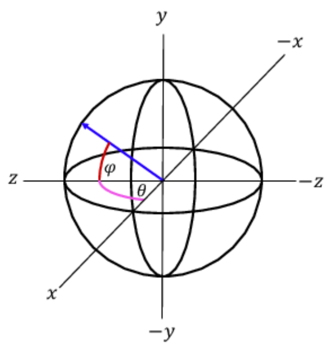

**Fig. 2.** Spherical coordinates of a unit vector.

Figure 3 is the equirectangular texture map projection of the environment depicted in the cubic texture map in Fig. 1, created using the aforementioned conversion algorithm.

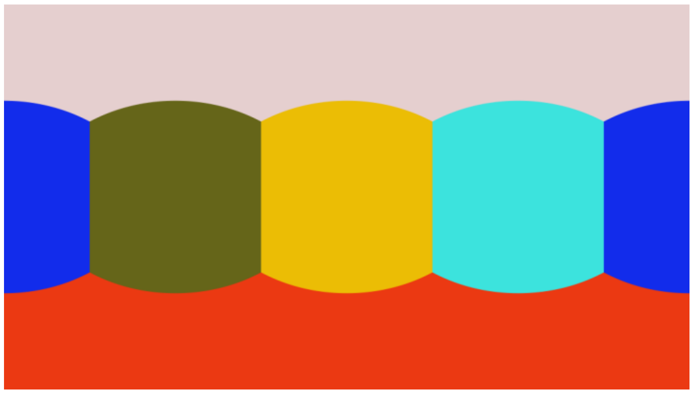

**Fig. 3.** Equirectangular environment map.

## 3.2 Dual Paraboloid Projection

A dual paraboloid mapping [2] consists of two images that represent the environment's unit sphere flattened through two paraboloid reflectors, above and below the x-y plane. In the positive hemisphere image, the points on the image are mapped to unit vectors that have a positive z-coordinate, and in the negative hemisphere image, the points on the image are mapped to unit vectors that have a negative z-coordinate.

As illustrated in Fig. 4, points on the two images are mapped to unit vectors in the following way. If the point $P_l = (x, y)$ is drawn from the positive image, then we determine the point at which the **top** paraboloid intersects the line passing through $(x, y)$ that is orthogonal to the x-y plane. Mathematically, this intersection is:

$$(r_x, r_y, r_z) = \left(x, y, \frac{1}{2} - \frac{1}{2}(x^2 + y^2)\right). \tag{4}$$

If instead the $(x, y)$ is drawn from the negative image, then we determine the point at which the **bottom** paraboloid intersects the line passing through $(x, y)$ that is orthogonal to the x-y plane. Mathematically, this intersection is:

$$(r_x, r_y, r_z) = \left(x, y, \frac{1}{2}(x^2 + y^2) - \frac{1}{2}\right). \tag{5}$$

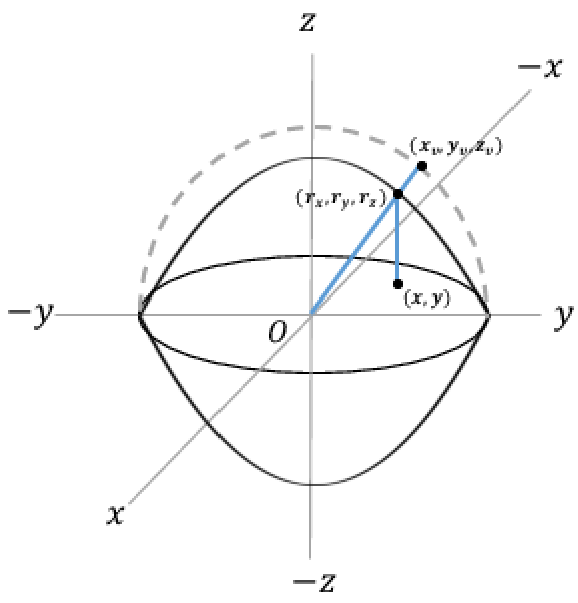

**Fig. 4.** Dual paraboloid mapping.

Now we normalize this point (which can be thought of as a ray from the origin to the paraboloid) to obtain our desired unit vector $(x_v, y_v, z_v)$:

$$\begin{cases} x_v = \frac{r_x}{\sqrt{r_x^2+r_y^2+r_z^2}}. \\ y_v = \frac{r_y}{\sqrt{r_x^2+r_y^2+r_z^2}}. \\ z_v = \frac{r_z}{\sqrt{r_x^2+r_y^2+r_z^2}}. \end{cases} \quad (6)$$

Figures 5 and 6 are the negative-z and positive-z (respectively) paraboloid texture map projections of the environment depicted in the cubic texture map in Fig. 1.

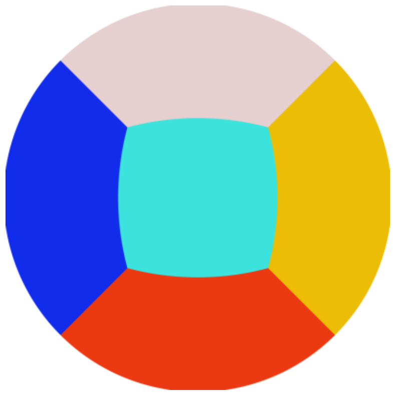

**Fig. 5.** Negative-z dual paraboloid environment map.

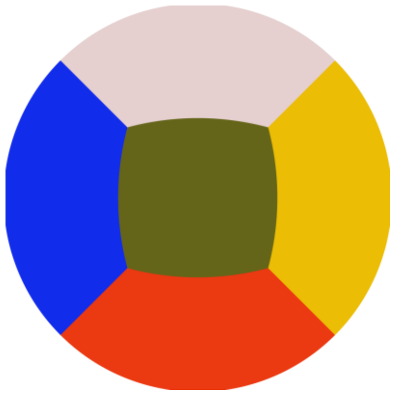

**Fig. 6.** Positive-z dual paraboloid environment map.

## 3.3 Octahedral Projection

An octahedral mapping [3] also consists of two images. These represent the environment's unit sphere flattened through the top and bottom of an octahedral reflector, above and below the x-z plane. In the positive hemisphere image, the points on the image are mapped to unit vectors that have a positive y-coordinate, and in the negative hemisphere image, the points on the image are mapped to unit vectors that have a negative y-coordinate.

As illustrated in Fig. 7, points on the two images are mapped to unit vectors in the following way. If the point $P_l = (x, y)$ is drawn from the positive image, then we determine the point at which the octahedron's faces residing in the $y \geq 0$ half-space intersect the line passing through $(x, z)$ that is orthogonal to the x-z plane:

$$(r_x, r_y, r_z) = \left(\frac{z+x}{2}, \frac{z-x}{2}, 1 - \left(|r_x| + |r_y|\right)\right). \tag{7}$$

If instead the $(x, z)$ is drawn from the negative image, then we determine the point at which the octahedron's faces residing in the $y < 0$ half-space intersect the line passing through $(x, z)$ that is orthogonal to the x-z plane:

$$(r_x, r_y, r_z) = \left(\frac{z-x}{2}, \frac{z+x}{2}, |r_x| + |r_y| - 1\right). \tag{8}$$

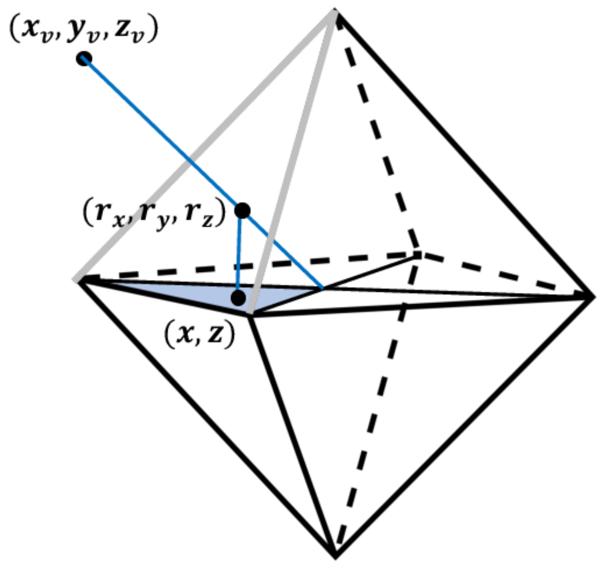

**Fig. 7.** Octahedral projection mapping.

Note that in the above equations, computing $r_z$ is dependent upon first computing $r_x$ and$r_y$. Now we simply normalize this point using Eq. (6) in Sec. 3.2 to obtain our desired unit vector $(x_v, y_v, z_v)$. Figures 8 and 9 are the positive-y and negative-y (respectively) octahedral texture map projection of the environment depicted in the cubic texture map in Fig. 1.

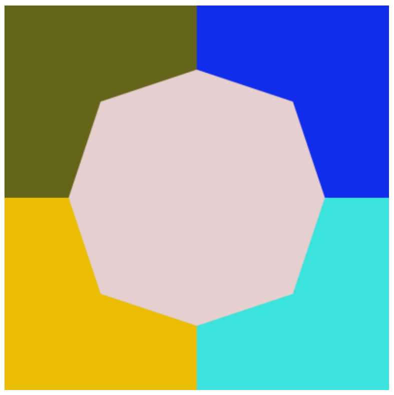

**Fig. 8.** Positive-y octahedral environment map.

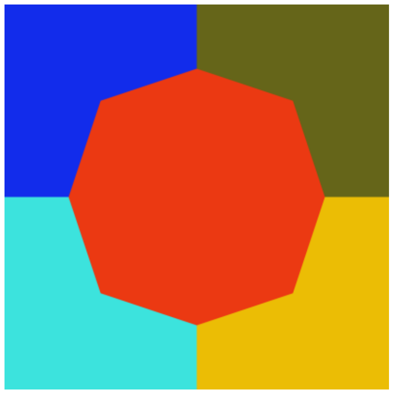

**Fig. 9.** Negatve-y octahedral environment map.

## 4. Projecting to Cube Maps

When converting to a cube map from some other texture map, we seek to populate the six individual $n$-pixel by $n$-pixel face maps. We traverse each of these maps pixel-by-pixel, and assign to each pixel an $(R, G, B)$ coloring. Consider a single pixel in this map $(i, j) \in \{0, \dots, n-1\}^2$ on face $f \in \{1, \dots, 6\}$. We sample a set of points $S = \{P_1, \dots, P_k\}$ from within this pixel, and for each of these points $P_l = (f, p_i, p_j)$ we compute the corresponding point on the unit sphere surface (a unit vector $(x_v, y_v, z_v)$). This computation is described in Listing 2.

```
convertCubeFacePixelToUnitRay(float &x, float &y, float &z, float pi, float pj, int face)
{
        double i = (2 * pi)/output_width - 1;
        double j = (2 * pj)/output_height - 1;
        /* Left */ if (face == 0) {z = -1; x = i; y = j;}
        /* Right */ if (face == 1) {z = 1; x = -i; y = j;}
        /* Back */ if (face == 2) {x = -1; z = -i; y = j;}
        /* Front */ if(face == 3) {x = 1; z = i; y = j;}
        /* Bottom */ if(face == 4) {y = 1; x = i; z = j;}
        /* Top */ if(face == 5) {y = -1; x = i; z = -j;}
        double mag = sqrt((x * x) + (y * y) + (z * z));
        x = x/mag; y = y/mag; z = z/mag;
}
```

Listing 2: Conversion algorithm for a cube face and pixel to a unit ray.

We then determine the point $(x, y)$ (or $(x, z)$ in the octahedral case) on the source texture map plane that corresponds to the unit vector $(x_v, y_v, z_v)$. Note that in paraboloid and octahedral maps, we must also identify on which of the two halves of the source map this pixel resides. How this information is determined is detailed in Secs. 4.1, 4.2, and 4.3. Once the point is located, we round it to obtain the pixel $(i_s, j_s)$ that it resides in:

```
i_s = (int) (((x  + 1.0) * input_width) / 2);
j_s = (int) ((([y or z] + 1.0) * input_height) / 2);
```

Once the above pixel $(i_s, j_s)$ is determined, we extract the coloring from this pixel: $(R_l, G_l, B_l)$. Once this process is repeated for all points in the set $S$, the pixel of interest $(i, j)$ is assigned the color average determined by Eq. (1) in Sec. 3. Once this process is repeated for all pixels $(i, j)$ on all faces $k$, our texture map conversion is complete.

### 4.1 Inverse Equirectangular Projection

In order to determine the point $(x, y)$ on the plane given the unit vector $(x_v, y_v, z_v)$, we apply the inverse of the transformation in Sec. 3.1. We have that the spherical coordinates of the unit vector are given by:

$$\begin{cases} \phi = \arcsin(y_v). \\ \theta = \arctan\left(\frac{z_v}{x_v}\right). \end{cases} \tag{9}$$

Now we scale these coordinates to produce the point $(x, y)$:

$$\begin{cases} x = \frac{\theta}{\pi}. \\ y = \frac{2\phi}{\pi}. \end{cases} \tag{10}$$

### 4.2 Inverse Dual Paraboloid Projection

In order to determine the point $(x, y)$ on the plane given the unit vector $(x_v, y_v, z_v)$, we must first find the point $(r_x, r_y, r_z)$ at which the unit vector intersects one of the two paraboloids. Note that a line in $\mathbb{R}^3$ with slope $(x_v, y_v, z_v)$ is given by the parametric equation $(x, y, z) = t(x_v, y_v, z_v)$.

When $z_v \geq 0$, we are interested in the intersection between this line and the top parabola. The point of intersection can be solved using the equation of the top parabola:

$$z_v t = \frac{1}{2} - \frac{1}{2}((x_v t)^2 + (y_v t)^2). \tag{11}$$

$$t^2\left(\frac{x_v^2+y_v^2}{2}\right) + z_v t - \frac{1}{2} = 0. \tag{12}$$

$$t = \frac{-z_v \pm \sqrt{x_v^2+y_v^2+z_v^2}}{x_v^2+y_v^2}. \tag{13}$$

Note that the unit vector satisfies $x_v^2 + y_v^2 + z_v^2 = 1$. In addition, we can reject the solution resulting in $t < 0$ because we require $z = z_v t \geq 0$ and we have that $z_v \geq 0$ for this case. Therefore, we have:

$$t = \frac{-z_v+1}{(1-z_v^2)} = \frac{-z_v+1}{(1+z_v)(1-z_v)} = \frac{1}{1+z_v}. \tag{14}$$

Thus, the point of intersection is given by:

$$(r_x, r_y, r_z) = \left(\frac{x_v}{1+z_v}, \frac{y_v}{1+z_v}, \frac{z_v}{1+z_v}\right). \tag{15}$$

Thus we have the point $(x, y)$, which is drawn from the positive-z (top) map:

$$\begin{cases} x = \frac{x_v}{1+z_v} \\ y = \frac{y_v}{1+z_v} \end{cases}. \tag{16}$$

When $z_v < 0$, we are interested in the intersection between the line and bottom parabola. The point of intersection can be solved for using the equation of the bottom parabola:

$$z_v t = \frac{1}{2}((x_v t)^2 + (y_v t)^2) - \frac{1}{2}. \tag{17}$$

$$t^2\left(\frac{x_v^2+y_v^2}{2}\right) - z_v t - \frac{1}{2} = 0. \tag{18}$$

$$t = \frac{z_v \pm \sqrt{x_v^2+y_v^2+z_v^2}}{x_v^2+y_v^2}. \tag{19}$$

Note that the unit vector satisfies $x_v^2 + y_v^2 + z_v^2 = 1$. Also, we can reject the solution resulting in $t < 0$ because we require $z = z_v t < 0$ and we have that $z_v < 0$ for this case. Therefore, we have:

$$t = \frac{z_v+1}{(1-z_v^2)} = \frac{z_v+1}{(1+z_v)(1-z_v)} = \frac{1}{1-z_v}. \tag{20}$$

Thus, the point of intersection is given by:

$$(r_x, r_y, r_z) = \left(\frac{x_v}{1-z_v}, \frac{y_v}{1-z_v}, \frac{z_v}{1-z_v}\right). \tag{21}$$

Thus we have the point $(x, y)$, which is drawn from the negative-z (bottom) map:

$$\begin{cases} x = \frac{x_v}{1-z_v} \\ y = \frac{y_v}{1-z_v} \end{cases}. \tag{22}$$

## 4.3 Inverse Octahedral Projection

In order to determine the point $(x, y)$ on the plane given the unit vector $(x_v, y_v, z_v)$, we must first find the point $(r'_x, r'_y, r'_z)$ at which the unit vector intersects the octahedron. Owing to the geometric properties of an octahedron, this point of intersection is given as:

$$\begin{cases} r'_x = \frac{x_v}{|x_v|+|y_v|+|z_v|}. \\ r'_y = \frac{y_v}{|x_v|+|y_v|+|z_v|}. \\ r'_z = \frac{z_v}{|x_v|+|y_v|+|z_v|}. \end{cases} \tag{23}$$

Note that the above transformation is based on defining the octahedral surface with the equation $|x| + |y| + |z| = 1$ in some $\mathbb{R}^3$ coordinate system. While this definition produces an elegant representation of the octahedron, the consequence is that the image plane of the two-dimensional x-z texture map that we seek to populate is rotated relative to the x'-z' plane used in the $\mathbb{R}^3$ coordinate system. Therefore, the point $(x, z)$ is **not** simply equal to $(r'_x, z'_x)$ and we must incorporate the rotation into the transformation from $(r'_x, y'_y, r'_z)$ to $(x, z)$. When $y_v \geq 0$, this transformation is given as follows, where the point $(x, y)$ is drawn from the positive-y (top) source image.

$$\begin{cases} x = r'_x - r'_z. \\ y = r'_x + r'_z. \end{cases} \tag{24}$$

When $y_v < 0$, this transformation is given as follows, where the point $(x, y)$ is drawn from the negative-y (bottom) source image.

$$\begin{cases} x = r'_z - r'_x. \\ y = r'_z + r'_x. \end{cases} \tag{25}$$

## 5. Sampling Techniques

We describe two strategies to sample points $\{P_l, \ldots, P_k\}$ from some pixel $(i, j)$ in the output space of the transformation: uniform sampling and correlated multijittered sampling [4]. For convenience, we assume the number of points we sample, $k$, is a perfect square. The justification for this is seen when examining the algorithms that produce the samplings. Because the ordered pair identifying the pixel represents the coordinates in $\mathbb{R}^2$ of the pixel's top left corner, all points sampled are drawn from the set $X_{(i,j)}$, where $X_{(i,j)} = \{(x, y) \in \mathbb{R}^2 \mid i < x < i + 1 \text{ and } j < y < j + 1\}$. Uniform sampling produces a uniform $\sqrt{k}$ by $\sqrt{k}$ grid of points on the selected pixel. The algorithm for uniform sampling is given in Listing 3.

```
generateUniformSamples(vector<Point2D> &points)
{
        int const n = points.size();
        for(int j = 0; j < n; j++) {
                for(int i = 0; i < n; i++) {
                        points[j*n+i].x = ((double) (i+1))/(n+1);
                        points[j*n+i].y = ((double) (j+1))/(n+1);
                }
        }
}
```

Listing 3: Uniform sampling

Note that uniform sampling is deterministic; it produces the same grid of points on every run. By contrast, correlated multi-jittered sampling produces a relatively uniform set of points but injects some randomness into the sample selection. This type of jittered sampling involves (1) generating a randomized canonical arrangement and then (2) randomly shuffling this arrangement. The canonical arrangement is produced by stratifying the pixel into a $\sqrt{k}$ by $\sqrt{k}$ grid of jitter cells, and then stratifying each of those jitter cells into a $\sqrt{k}$ by $\sqrt{k}$ grid of sub-cells. For each jitter cell $C$, we randomly select a sample in the sub-cell corresponding to $C$'s location in the pixel. The effect is that each jitter cell contains one randomly-placed sample such that every row and every column of sub-cells in the pixel contains exactly one sample. Figure 10 shows the canonical arrangement of samples with a single pixel.

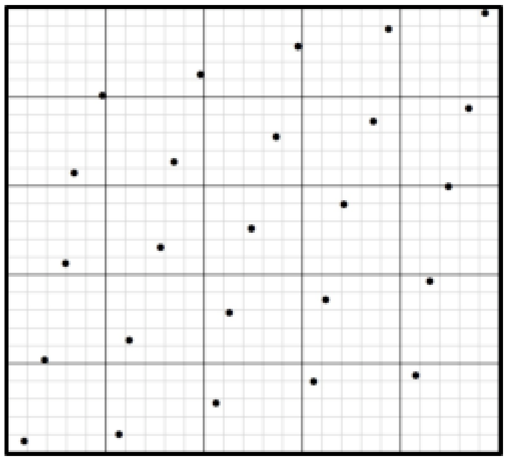

**Fig. 10.** Canonical arrangement of samples within one pixel.

To produce the final arrangement, all that is left is to apply a random shuffle to the x-coordinates of the selected samples, and then apply the same shuffle to the y-coordinates of the selected samples. Figure 11 displays an example of a final shuffled selection of points.

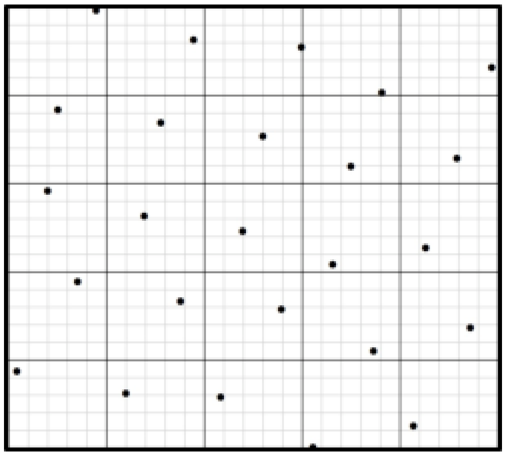

**Fig. 11.** Shuffled arrangement of samples within one pixel.

To evaluate these sampling strategies against one another, we selected a set of test cube maps and transformed them into equirectangular projections (using both uniform and correlated multi-jittered sampling with different numbers of samples). We observed two key points by visually examining the produced images:

(1) Images produced with a higher number of samples per pixel were less aliased than images produced with a lower number of samples per pixel. An example of the quality improvement associated with a higher number of samples is given in Fig. 12. However, visual results indicate that increasing the number of samples past 36 stops producing visually perceptible quality improvements.

(2) Jittered sampling produces images with less aliasing in projections rendered with a low number of samples. However, the benefit of using jittered sampling declines as more samples are used per pixel.

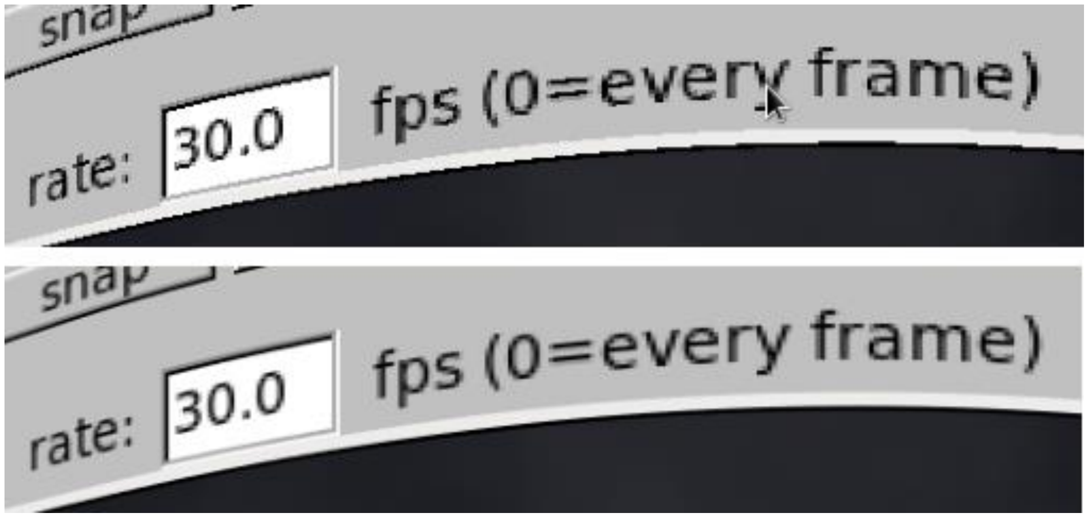

**Fig. 12.** Comparing different samples per pixel. The top image was projected with a single centered sample, while the bottom image was projected with 25 samples per pixel. The 25 samples per pixel produce a noticeable improvement in visual quality.

We also performed an objective comparison of the generated images using the $\Delta E$ color distance metric [7]. A set of 8 test frames (cube maps) were converted to equirectangular projections using both types of sampling algorithms (at different numbers of samples). For each converted frame, the $\Delta E$ value between the projection and a pseudo-ground-truth[2] image was computed. From the resulting data set, we were able to compute, for each number of samples $k \in \{4,9,25,\dots 400\}$, a set of eight values $\{v_1,\dots,v_8\}$. Each $v_i$ represents the difference between the uniform-to-ground-truth $\Delta E$ value and the jittered-to-ground-truth $\Delta E$ value for one frame. For each $k$, a Student's T-Distribution was used to construct a 90% confidence interval based on the eight uniform-jittered differences.

Figure 13 shows these confidence intervals. This graph generally confirms the aforementioned observations (jittered sampling produces images closer to the ground truth than uniform sampling does, but the benefit declines as the number of samples increases). However, we see that quantitative color-difference results are unable to clearly delineate the point at which a higher number of samples ceases to produce visually observable quality benefits. In general, our results in evaluating sampling algorithm are consistent with the results of prior literature [4] [5] [6].

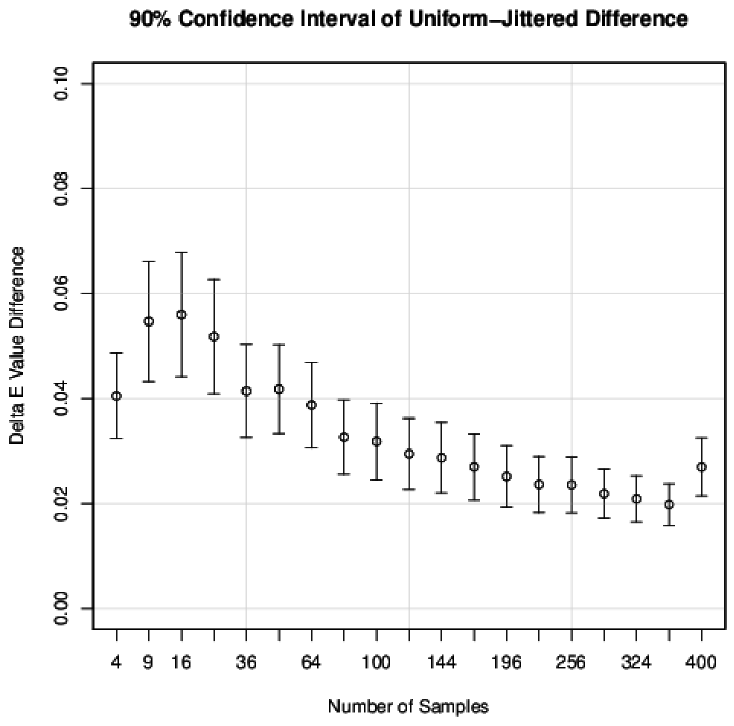

**Fig. 13.** Confidence intervals for difference between uniform sampling $\Delta E$ and multi-jittered sampling $\Delta E$. The error bars are standard error.

## 6. Comparative Analysis of Projections

Equirectangular, dual-paraboloid, and octahedral mappings can be compared with one another using the following procedure. We select a particular cube map, transform it into each of the three other types of texture maps, and then transform each of those texture maps back into a cube map. We then compare each of the three generated cube maps to the original, using both the $\Delta E$ metric as well as the SSIM structural

[2] For each cube map, the pseudo-ground-truth image was an equirectangular projection of that cube map computed using 400 samples. Using such an image as a ground-truth reference is reasonable because 400 samples is well beyond the point at which increasing the number of samples produces quality benefits. The ground-truth here is therefore the best possible conversion.

similarity metric [8]. If one of the three transformations has a lower $\Delta E$ and/or a higher SSIM, then this transformation is better able to preserve the information contained in the source image than the other transformations. The results of this comparison are summarized in Figs. 14 and 15, each of which displays a box plot of 50 test frames. These results indicate that no method of transformation has any statistically significant benefit over the others in preserving the information contained in the source image. Note that for each test frame, the output images were sized such that all three texture maps contained roughly the same total number of pixels. Moreover, the cube maps produced from the conversions were of exactly the same size as the originals.

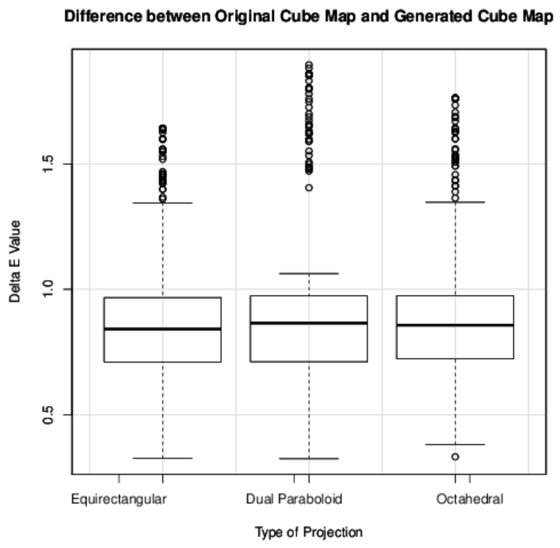

**Fig. 14.** $\Delta E$ measurement between original cube map and projected/inverse projected cube map over 50 test frames.

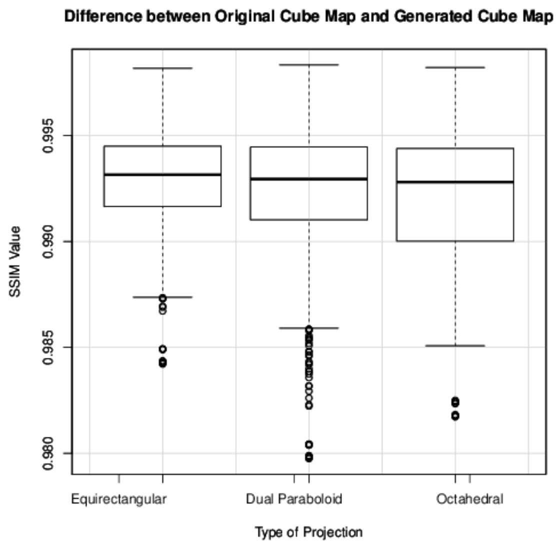

**Fig. 15.** SSIM measurement between original cube map and projected/inverse projected cube map over 50 test frames.

## 7. References

[1] Bourke P (2016) Converting to/from cubemaps. http://paulbourke.net/miscellaneous/cubemaps/; accessed July 2016.
[2] Imagination Technologies Limited (2016) Dual Paraboloid Environment Mapping Whitepaper. (Imagination Technologies Limited, UK).
[3] Engelhardt T and Dachsbacher C (2008) Octahedron environment maps. *Proceedings of the 2008 Vision Modelling and Visualization (VMV) Conference* (Aka GmbH, Konstanz Germany), pp 383-388.
[4] Kensler A (2013) Correlated multi-jittered sampling. (Pixar, Emeryville, CA), Technical memo 13-01.
[5] Kollig T and Keller A (2002 Efficient multidimensional sampling. *Computer Graphics Forum* 21(3):557-563 https://doi.org/10.1111/1467-8659.00706.
[6] Chiu K, Shirley P, and Wang C (1994) Multi-jittered sampling. *Graphics Gems IV*, ed Heckbert P (AP Professional, Boston), pp 370-374.
[7] Sharma G, Wu W, and Dalal EN (2005) The CIEDE2000 color-difference formulate: implementation notes, supplementary test data, and mathematical observations. *Color Research & Application* 30(1):21-30 https://doi.org/10.1002/col.20070.
[8] Wang Z, Bovik AC, Sheikh HR, and Simoncelli EP (2003) Image quality assessment: from error measurement to structural similarity. *IEEE Trans. On Image Processing* 13(4):600-612 https://doi.org/10.1109/TIP.2003.819861.

***About the authors:*** *Vinay Sriram is an undergraduate student in the Computer Science Department at Stanford University, and formerly a summer research intern at NIST.*

*Wesley Griffin is a Computer Scientist in the Applied and Computational Mathematics Division at NIST. His research interests are in scientific visualization and real-time computer graphics.*

*The National Institute of Standards and Technology is an agency of the U.S. Department of Commerce.*