# CONTINUOUS CUBE MAPPING

*Cindy M. Grimm Bill Niebruegge*

Department of Computer Science and Engineering
Washington University in St. Louis
One Brookings Drive
St. Louis, MO 63130
United States
cmg@cse.wustl.edu and niebruegge.1@osu.edu

**ABSTRACT**

Existing environment mapping techniques include spherical mapping and cube mapping. These techniques have inherent flaws that cause sampling issues and aliasing. Continuous cube mapping is offered as an alternative environment mapping approach that effectively folds the cube onto the sphere, providing a better parameterization of cube mapping. We provide a hardware implementation.

## 1. INTRODUCTION

Environment mapping is the process of surrounding a scene with a large, textured shape and using it to determine the color of each the scene's objects. The two most prevalant approaches are spherical mapping and cube mapping. Each of these methods have inherent flaws, including sampling issues and aliasing. Continuous cube mapping is introduced to offer an improved parameterization of the surrounding texture while only requiring a minimal increase in computation time.

## 2. ENVIRONMENT MAPPING

Environment mapping is used to create the illusion of a surrounding environment or as a cheap alternative to area light sources in a scene. In environment mapping, a scene is surrounded by a large, textured shape. For each fragment of an object, a reflected ray, based on the view direction and surface normal, is first calculated - (Figure 1). The reflected ray is a function of the incident ray and the surface normal, R = $2(N.I)N - I$. This reflected ray is then used to index the texture applied to the surrounding shape.

Two common approaches for environment mapping include spherical mapping and cube mapping. The difference between the two approaches is how textures are stored in memory and accessed. In spherical mapping, one square texture is mapped onto a sphere. In cube mapping, there are six separate textures that represent the six faces of the cube. Before sampling the textures in cube mapping, we must first determine which texture to sample based on which face the reflected vector intersects.

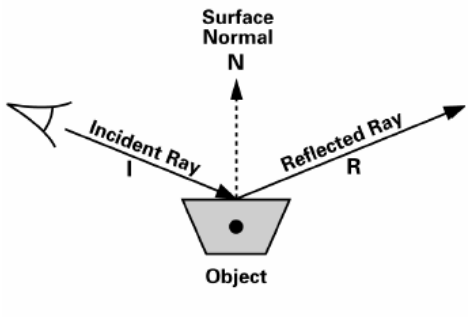


**Fig. 1**. Environment Mapping Rays [1]

### 2.1. Spherical Mapping

In spherical mapping, the position on the sphere is defined in terms of latitude and longitude. Latitude is referenced in terms of the angle $\phi$ and longitude is referenced in terms of the angle $\theta$. Given the reflected vector, $\theta$ and $\phi$ are defined as:

$$\theta = \arctan(\frac{Ry}{Rx}) \tag{1}$$

$$\phi = \arccos(\frac{Rz}{|R|}) \tag{2}$$

The corresponding $uv$ texture coordinates are then defined as:

$$u = \frac{\theta}{2\pi} \tag{3}$$

$$v = 0.5 + \frac{\phi}{\pi} \tag{4}$$

Spherical mapping's flaws are due to the fact that the technique maps a square texture onto a spherical shape, causing very different sampling rates at the poles and the equator. This is most apparent at the poles of the sphere as shown in Figure 2.

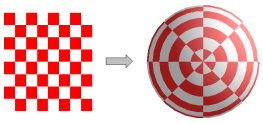

**Fig. 2**. Spherical Mapping: Pole View

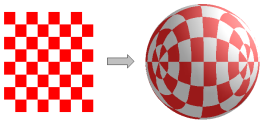

**Fig. 3**. Spherical Mapping: Equator View

### 2.2. Cube Mapping

In cube mapping, the environment is stored as the six faces of a cube. Each of the faces is a texture with $uv$ coordinates ranging from 0 to 1. The first step is to determine which of the faces the reflected ray intersects. The $uv$ values to be sampled are determined by finding the point of intersection with the face within this range.

The cube shape is a more straightforward shape than the sphere since the surrounding sides are flat, making creating the necessary textures easier. Think of standing in one position and taking six photographs each at an orthogonal 90 degree view from the others. This would provide the necessary six textures (Figure 5).

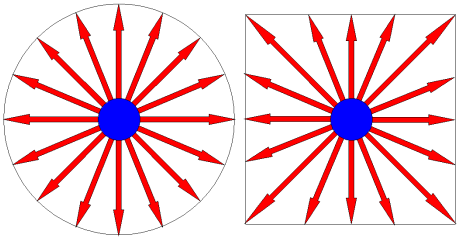

**Fig. 4**. Spherical Sampling vs. Cube Sampling

While cube mapping offers a more intuitive representation of the surroundings, it still introduces issues at the edges of the cube. With spherical mapping, the sampling is uniform while in cube mapping it differs between the center of the cube face and the edge (Figure 4). This difference in sampling results in artifacts. Likewise, the abrupt edges of the cube result in discontinuities with textures that span across an edge. These issues will not be as apparent when texturing more complex shapes (Figure 6).

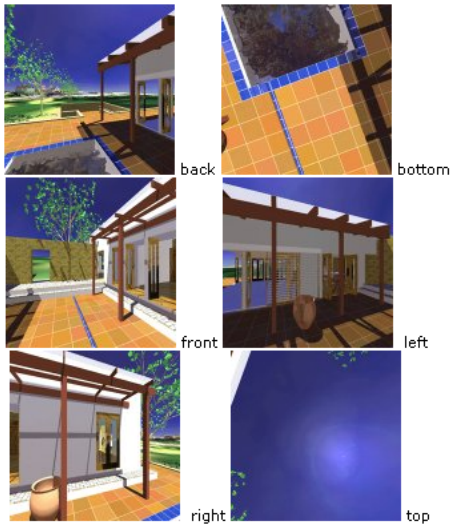


**Fig. 5**. Cube Mapping Textures [2]

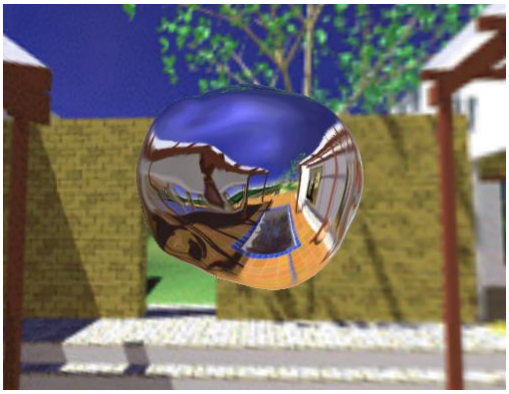

**Fig. 6**. Cube Mapping Example [2]

### 2.3. Parabolic Mapping

This environment mapping approach uses two parabolic maps, one for each hemisphere of the sphere. This produces a more even sampling but results in a fair amount of wasted area in the texture map. The mapping functions, however, are simple - a combination of multiplications and additions [6].

## 3. CONTINUOUS CUBE MAPPING

Continuous cube mapping combines the fundamental concepts of spherical and cube mapping while removing their inaccuracies. The basic idea is to fold the cube onto the

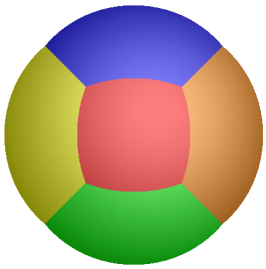

**Fig. 7**. Sphere divided into six faces

sphere. The result is the sphere divided into six faces, each of which is bounded by a great circle, resulting in a better parameterization than the cube (Figure 7). As in cube mapping, the reflected vector will intersect one of the six faces of the sphere, resulting in using the texture applied to that face. This is a continuous mapping, meaning the parameterization of one face can be extended into any adjacent face. This makes this method a better choice for capturing and manipulating spherical data.

One difficulty is that the math for converting the reflected vector to the angles $\theta$ and $\phi$ is different for each face of the sphere. However, since the mapping is symmetrical on all six faces, we can simplify the math for only one face and rotate all other faces' processing to it. This means we can use the same equations for all faces, by simply rotating the vector into the correct axes first. This rotation is handled by rotating the axes in 90 degree combinations and then processing with the base $\theta$ and $\phi$. For this implementation, we chose the positive-y face as the base face.

The steps of the algorithm are defined as:

1. Identify face based on the reflected vector ($Rx$, $Ry$, $Rz$).
2. Rotate sphere so the reflected vector is in the positive-y face.
3. Use sphere latitude/longitude equations to convert the reflected vector to $\theta$ and $\phi$.
4. Convert the angles $\theta$ and $\phi$ to square texture coordinates $u$ and $v$ so that the great arc boundaries map to the unit square.

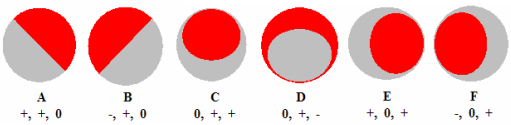


**Fig. 8**. Great arcs of the sphere

Steps 3 and 4 can be combined and simplified if some preprocessing is performed.

To determine which face the reflected vector is intersecting, we use the fact that the sphere can be divided by six great circles (Figure 8). Each face is then defined by a combination of the resulting great arcs. To determine which face is being intersected, we take the dot product of the reflected vector with each of the six normals and check combinations of the results. Figure 9 shows how to use the sign of the resulting dot products to determine which face is being intersected. For example, if it is intersecting the positive-y face, the dot products with normals A, B, C, and D will all be positive.

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| Positive-y | + | + | + | + | N/A | N/A |
| Negative-y | - | - | - | - | N/A | N/A |
| Positive-z | N/A | N/A | + | - | + | + |
| Negative-z | N/A | N/A | - | + | - | - |
| Positive-x | + | - | N/A | N/A | + | - |
| Negative-x | - | + | N/A | N/A | - | + |

**Fig. 9**. Face determination table

Now we can simplify the math for the positive-y case only. This involves combining steps 3 and 4, combining the projection from the sphere with the mapping to the unit square. These steps simplify to the following equations for the $u$ and $v$ coordinates.

$$u = 0.5 - \frac{\arctan(\frac{Rx}{Ry\sqrt{2}})}{2\arcsin(\frac{1}{\sqrt{3}})} \tag{5}$$

$$v = \frac{1 + \frac{\arcsin(Rz)}{\arcsin(\frac{1}{\sqrt{2+\frac{Rx^2}{Ry^2}}})}}{2} \tag{6}$$

## 4. EXAMPLES

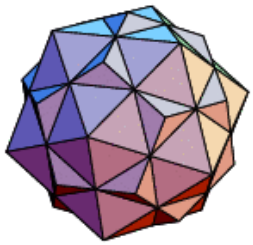

**Fig. 10**. Icosahedron and Dodecahedron Shape [4]

The following example images compare the different environment mapping methods. An icosahedron and a dodecahedron (Figure 10) are used together to define the vertices that will be mapped using the different methods. The vertices of the icosahedron are represented as red crosses and

the vertices of the dodecahedron are represented as blue crosses.

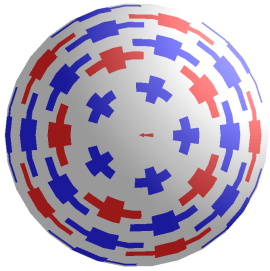

**Fig. 11**. Spherical Mapping

### 4.1. Spherical Mapping

For spherical mapping, the equations in section 2.1 were used to transform the vertices to $uv$ coordinates, resulting in Figure 11. In this case the vertices of the two polygons are distorted and the vertex at the pole of the sphere is pinched together, almost erasing it completely.

### 4.2. Cube Mapping

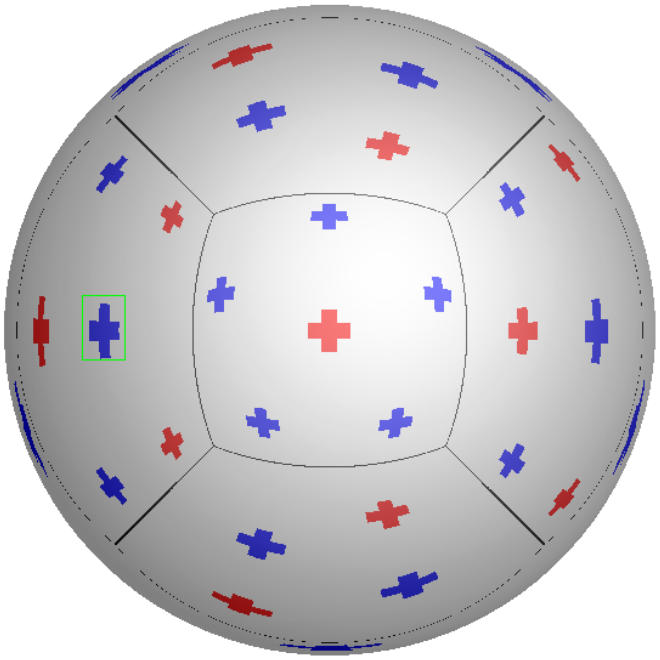

**Fig. 12**. Cube Mapping

For the cube mapping image, each vertex was treated as a vector from the origin. It was then determined which cube face this vector intersected and the corresponding $uv$ coordinates were computed based on where the vector intersected the face. The resulting image, Figure 12, shows the sampling issue with cube mapping. The crosses vary in size and shape depending on where they are in the cube face.

### 4.3. Continuous Cube Mapping

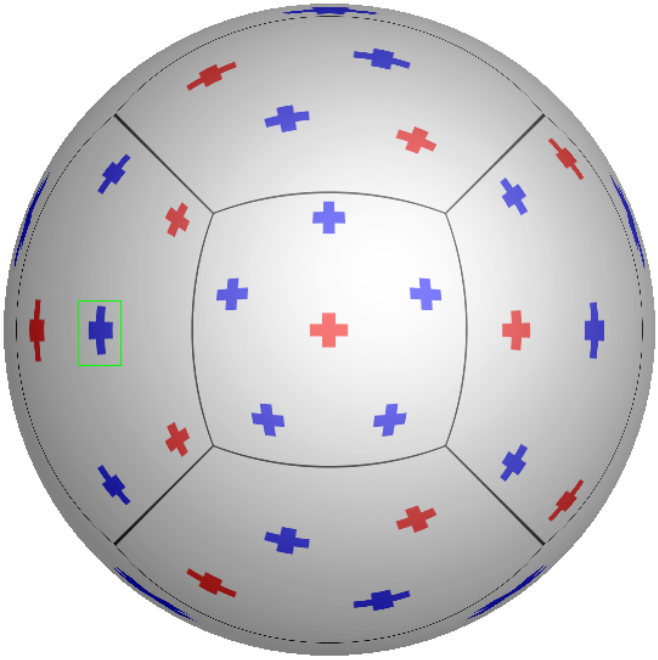

**Fig. 13**. Continuous Cube Mapping

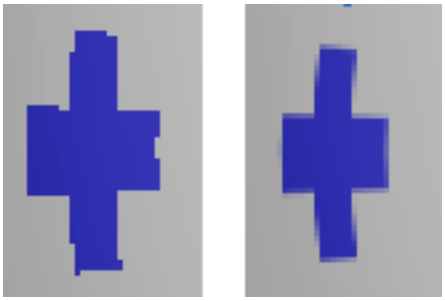

**Fig. 14**. Cube Mapping vs. Continuous Cube Mapping Cross

For continuous cube mapping, each vertex was again treated as a vector from the origin. It was then determined which face this vector intersected using the methods described in Section 3. The math in Appendix B was then used to compute the corresponding $uv$ coordinates. The resulting image, Figure 13, shows a more correctly spaced version of the cube mapping example with more uniformly shaped crosses (a comparison of the middle faces is the best example). Figure 14 compares crosses from each method.

The cube mapping cross has aliasing, while the continuous cube mapping cross is smoother.

## 5. REFERENCES


[1] Randima Fernando and Mark J. Kilgard, 2003
*The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics, pages 169 - 197*
Addison-Wesley

[2] NVIDIA Cube Map OpenGL Tutorial
http://developer.nvidia.com/object/cube_map_ogl_tutorial.html

[3] Parameterization Using Manifolds
Cindy Grimm
*International Journal of Shape Modeling*, Volume 10, Number 1, Pages 51-80
Editor: B Falcidieno
World Scientific, June 2004

[4] Dodecahedron – from Wolfram MathWorld
http://mathworld.wolfram.com/Dodecahedron.html

[5] Models of Light Reflection For Computer Synthesized Pictures
James F. Blinn
*Computer Graphics (Proceedings of SIGGRAPH 77)*
Jul 1977, 192-198

[6] Wolfgang Heidrich and Hans-Peter Seidel, 1998
*View-independent Environment Maps*
1998 SIGGRAPH / Eurographics Workshop on Graphics
Aug 1998, pages 39 - 46


## 6. APPENDICES

The appendices of this paper provide the Cg fragment shader code and the math to convert texture coordinates in continuous cube mapping. The math for this process was first presented in [3].

Appendix A gives the Cg fragment shader code. This takes a reflected vector and uses the $uv$ coordinate equations described in Section 3 to perform texture lookups. The $uv$ coordinate equations from Section 3 are condensed versions of the math presented in Appendx B. The math in Appendix B is reversible. This process reversed is provided in Appendix C.

## A. CG FRAGMENT SHADER

```
float4 Rotate_R(float3 R)
{
    // makes the program think we're only computing the
    // positive-y texture
    float x, y, z, texIndex;
    float L = sqrt(2.0) / 2.0;

    // create the normals to check against
    float3 N1 = float3(L, L, 0.0);
    float3 N2 = float3(-L, L, 0.0);
    float3 N3 = float3(0.0, L, L);
    float3 N4 = float3(0.0, L, -L);
    float3 N5 = float3(L, 0.0, L);
    float3 N6 = float3(-L, 0.0, L);

    R = normalize(R);

    bool check1 = ceil(dot(N1, R));
    bool check2 = ceil(dot(N2, R));
    bool check3 = ceil(dot(N3, R));
    bool check4 = ceil(dot(N4, R));
    bool check5 = ceil(dot(N5, R));
    bool check6 = ceil(dot(N6, R));

    // positive y
    if (check1 && check2 && check3 && check4)
    {
        // do nothing
        x = R.x;
        y = R.y;
        z = R.z;
        texIndex = 0.0;
    }
    // negative y
    else if (!check1 && !check2 && !check3 && !check4)
    {
        // rotate around x, 180 degrees
        x = R.x;
        y = -1.0*R.y;
        z = -1.0*R.z;
        texIndex = 1.0;
    }
    // positive z
    else if (check5 && check6 && check3 && !check4)
    {
        // rotate around x, 90 degrees
        x = R.x;
        y = R.z;
        z = -1.0*R.y;
        texIndex = 2.0;
    }
    // negative z
    else if (!check5 && !check6 && !check3 && check4)
    {
        // rotate around x, -90 degrees
        x = R.x;
        y = -1.0*R.z;
        z = R.y;
        texIndex = 3.0;
    }
    // negative x
    else if (!check5 && check6 && !check1 && check2)
    {
        // rotate around z, -90 degrees
        x = R.y;
        y = -1.0*R.x;
        z = R.z;
        texIndex = 4.0;
    }
    // positive x
    else if (check5 && !check6 && check1 && !check2)
    {
        // rotate around z, 90 degrees
        x = -1.0*R.y;
        y = R.x;
        z = R.z;
        texIndex = 5.0;
    }
    else
    {
        // do nothing
        x = 0.0;
        y = 0.0;
        z = 0.0;
        texIndex = 6.0;
    }

    return float4(x, y, z, texIndex);
}

void frag_program(  float3 R           : TEXCOORD1,

                out float4 oColor      : COLOR,

            uniform sampler2D decalX,
            uniform sampler2D decalNegX,
            uniform sampler2D decalY,
            uniform sampler2D decalNegY,
            uniform sampler2D decal2,
            uniform sampler2D decalNegZ )
{
    float4 temp = Rotate_R(R);
    float3 rotatedR = temp;
    float  texIndex = temp[3];

    float x = rotatedR.x;
    float y = rotatedR.y;
    float z = rotatedR.z;

    float S1 = atan2((x / y), sqrt(2.0));
    float S2 = 2.0 * asin(1.0 / sqrt(3.0));
```

```
    float S = 0.5 - S1/S2;
    float T = ( 1.0 + (asin(z) /
       ( asin(1.0 / (sqrt (2.0 + (x/y)*(x/y) ) ) ) ) ) ) / 2.0;

    // Fetch relected environment color
    if (texIndex == 0.0)
    {
       oColor = tex2D(decalY, float2(S, T));
    }
    else if (texIndex == 1.0)
    {
       oColor = tex2D(decalNegY, float2(S, T));
    }
    else if (texIndex == 2.0)
    {
       oColor = tex2D(decalZ, float2(S, T));
    }
    else if (texIndex == 3.0)
    {
       oColor = tex2D(decalNegZ, float2(S, T));
    }
    else if (texIndex == 4.0)
    {
       oColor = tex2D(decalNegX, float2(S, T));
    }
    else if (texIndex == 5.0)
    {
       oColor = tex2D(decalX, float2(S, T));
    }
    else if (texIndex == 6.0)
    {
       oColor = float4(0.0, 0.0, 0.0, 1.0);
    }
}
```

## B. MATH TO CONVERT FROM RX,RY,RZ TO SQUARE TEXTURE COORDINATES

```
dPi = 3.0 * PI / 4.0

switch(face}
case 0:
    dS = atan2(Ry, Rx)
    dT = asin(Rz)
    dS = dS / PI
    dT = (dT + dPi / 2.0) / dPi
    break
case 1:
    dS = atan2(Ry, Rx)
    dT = asin(Rz)
    dS = 1.0 + dS / PI
    dT = (dT + dPi / 2.0) / dPi
    break
case 2:
    dS = atan2(Rx, Rz)
    dT = asin(Ry)
    dS = dS / PI
    dT = (dT + dPi / 2.0) / dPi
    break
case 3:
    dS = atan2(Rx, Rz)
    dT = asin(Ry)
    dS = 1.0 + dS / PI
    dT = (dT + dPi / 2.0) / dPi
    break
case 4:
    dS = atan2(Rz, Ry)
    dT = asin(Rx)
    dS = dS / PI
    dT = (dT + dPi / 2.0) / dPi
    break
case 5:
    dS = atan2(Rz, Ry)
    dT = asin(Rx)
    dS = 1.0 + dS / PI
    dT = (dT + dPi / 2.0) / dPi
    break

if (dS < -0.5)
    dS = dS + 2.0
else if (dS > 1.5)
    dS = dS - 2.0


c_dv1 = asin( -sqrt(1/3) ) * 4 / (3 * PI) + 0.5
c_dv2 = asin( sqrt(1/3) ) * 4 / (3 * PI) + 0.5


dSP = (0.75 + 1.0) * PI
dYdX = tan(PI * dS)
dA = dYdX / cos(dSP)
dTP = atan(1 / dA)
dPerc = (dTP + dPi / 2) / dPi

u = (dPerc - c_dv2) / (c_dv1 - c_dv2)

dZ = sin(dSP) * cos(dTP)
dAng2 = asain(dZ)
dTMe = (dAng2 + dPi / 2) / dPi

v = (dT - dTMe) / (1 - 2 * dTMe)
```

## C. MATH TO CONVERT FROM SQUARE TEXTURE COORDINATES TO RX, RY, RZ

```
dPi = 3.0 * PI / 4.0


c_dv1 = asin( -sqrt(1/3) ) * 4 / (3 * PI) + 0.5
c_dv2 = asin( sqrt(1/3) ) * 4 / (3 * PI) + 0.5

dPercV = c_dv2 * (1.0 - u) + c_dv1 * u

dSP = (0.75 + 1.0) * PI
dTP = (dPercV * dPi - dPi / 2)

dY = cos(dSP) * cos(dTP)
dX = sin(dTP)
dZ = sin(dSP) * cos(dTP)
dAng1 = atan2(dY, dX)
dAng2 = asin(dZ)
dSMe = dAng1 / PI
dTMe = (dAng2 + dPi / 2) / dPi


switch(face}
case 0:
    dS = dU * PI
    dT = dV * dPi - dPi / 2

    Rx = cos(dS) * dCT
    Ry = sin(dS) * dCT
    Rz = sin(dT)
    break
case 1:
    dS = (dU + 1) * PI
    dT = dV * dPi - dPi / 2

    Rx = cos(dS) * dCT
    Ry = sin(dS) * dCT
    Rz = sin(dT)
    break
case 2:
    dS = dU * PI
    dT = dV * dPi - dPi / 2

    Rx = sin(dS) * dCT
    Ry = sin(dT)
    Rz = cos(dS) * dCT
    break
case 3:
    dS = (dU + 1) * PI
    dT = dV * dPi - dPi / 2

    Rx = sin(dS) * dCT
    Ry = sin(dT)
    Rz = cos(dS) * dCT
    break
case 4:
    dS = dU * PI
    dT = dV * dPi - dPi / 2

    Rx = sin(dT)
    Ry = cos(dS) * dCT
    Rz = sin(dS) * dCT
    break
case 5:
    dS = (dU + 1) * PI
    dT = dV * dPi - dPi / 2

    Rx = sin(dT)
    Ry = cos(dS) * dCT
    Rz = sin(dS) * dCT
    break
```