

Spherical Scale Mapping

Reiter Stephan

Abstract

This paper presents a practical approach to the interpolation between arbitrary convex polygonal models. In a preprocessing step a model's bounding sphere serves as the base for the creation of a cubic scale map. During the rendering process this map is applied to a sphere using vertex texturing. Interpolation between two different models may then be implemented as the interpolation of the associated scale maps.

1. OVERVIEW

The main motivation behind the development of this technique was the search for a common solution to the problem of interpolation between arbitrary polygonal models. Arbitrary in the sense of having different vertex and triangle-counts which do not allow the employment of typical interpolation between paired vertices of the two models in question.

The basic idea is that models have at least one thing in common: they can be enclosed by a bounding sphere. The technique builds upon this fact and uses a cubemap, which is computed in a preprocessing step. Unlike traditional displacement mapping techniques, which use vertex texturing to offset vertices along their normals [1], the created cubemap is used to scale vertices in such a way that the base sphere resembles the original model. Blending between two scale maps then effectively means interpolating between two models. Furthermore an algorithm is proposed which allows the recreation of vertex normals for lighting-calculations.

2. PREPROCESSING THE MODEL

The preprocessing step covers the creation of the scale map: First the bounding sphere of the model is determined. Around this sphere a virtual axis-aligned box - representing the cubemap - is created: It shares the origin with the sphere and its faces have an edge-length of twice the bounding sphere radius.

At the center of each cubemap texel a ray is spawned towards the bounding sphere's center and the intersection-point with the model is determined. A scale-factor in the range of [0.0, 1.0] can then be computed - 0.0 meaning, the vertex lies in the bounding sphere's origin; 1.0 leaving the vertex on the surface of the bounding sphere.

$$\lambda = \frac{\overrightarrow{\text{mesh_intersection} - \text{bounding_origin}}}{\text{bounding_radius}} \quad (1)$$

```

For each cubemap-texel {
  Compute texel-position and the direction towards the bounding sphere's origin.
  Spawn a ray and determine closest intersection point with the model.
  float fDist = Length of vector from bounding sphere origin to intersection point.
  float fLambda = fDist / fBoundingSphereRadius; // 0.0 <= fLambda <= 1.0
  Store fLambda * 255 in the cubemap.
}

```

List. 1. Basic algorithm for the preprocessing step.

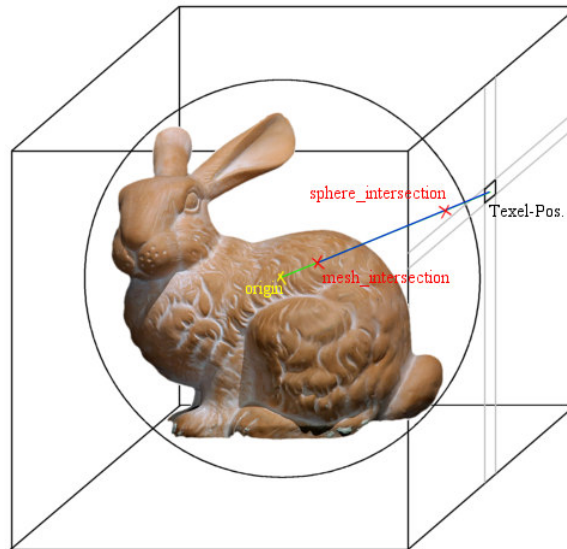


Fig. 1. The scene-setup for the preprocessing step. [2]

3. RENDERING WITH THE SCALE MAP

To reconstruct the original model from the computed scale map a polygonal sphere is rendered employing a vertex shader. The suggested type of sphere is a geosphere [3] [4], which provided better quality than a polar sphere due to a better distribution of vertices.

During the rendering process the vertex shader performs a texture-lookup into the cubemap using the vertex' normal (normal retrieved from the base-sphere) and scales the vertex' position. After this step the vertex can be transformed into worldspace.

```
float32 fGetScale( const vector3 &i_vNormal, const float32 i_fInterpolation ) {
    vector4 vScaleA, vScaleB;
    SampleTexture( vScaleA, 2, i_vNormal.x, i_vNormal.y, i_vNormal.z );
    SampleTexture( vScaleB, 3, i_vNormal.x, i_vNormal.y, i_vNormal.z );
    return fLerp( vScaleA.r, vScaleB.r, i_fInterpolation );
}

void Execute( const shaderreg *i_pInput, vector4 &o_vPosition, shaderreg *o_pOutput ) {
    const vector3 &vPos = i_pInput[0]; // Get position of vertex.
    const float32 fInterpolation = fGetFloat( 0 ); // Get interpolation-delta.

    // Determine vertex scale from the cubemap: Here we simply use the vertex position as the
    // vertex normal, due to the nature of the base-sphere (origin at (0,0,0)).
    const vector3 vScaledPos = vPos * fGetScale( vPos, fInterpolation );

    // Transform the final vertex position.
    const vector4 vFinalPos = vector4( vScaledPos.x, vScaledPos.y, vScaledPos.z, 1 );
    o_vPosition = vFinalPos * matGetMatrix( m3dsc_wvpmatrix );
}
```

List. 2. Muli3D [5] vertex-shader code: interpolates between two cubemap-samples and scales the vertex in the local coordinate system.

3.1. Reconstructing vertex normals

Rendering the geometry itself didn't pose much of a problem, yet the reconstruction of the original model isn't finished: per vertex normal vectors are not recalculated from the cubemap.

One solution to this problem can be implemented in the preprocessing step: When determining the scale-factor for a particular vertex we could go along and also store the normal vector in the cubemap. This would mean an additional usage of three color-channels, which would lead to a RGBA-cubemap texture (Normal scaled to [0,255] range and stored in

the RGB channels; λ scaled to [0,255] and stored in the alpha-channel), effectively resulting in an increase of the size of a factor of four.

With the advent of consumer-level hardware with support for VS-model 3.0 the reconstruction of vertex-normals can be moved to a shader with the cubemap only storing scale-factors and limited to a single-channel texture.

The basic idea behind the algorithm is the lookup of neighbour vertex positions – a normal per three vertices is calculated using the cross-product; all normals are averaged and renormalized to find the normal of the vertex in question.

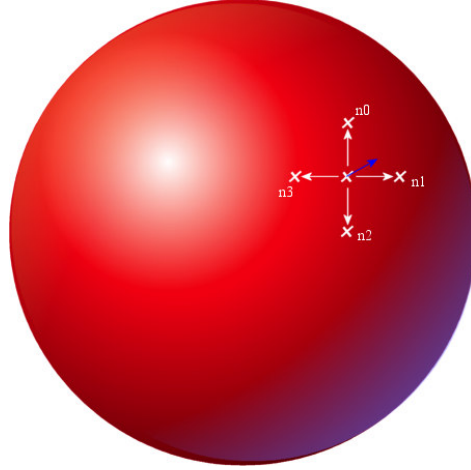


Fig. 2. Reconstruction of a vertex normal vector (Figure ignores vertex-scaling).

How do we determine the scaled positions of the neighbour vertices? First we have to generate the correct points on the surface of the base-sphere. We do this by determining the spherical coordinates for the current vertex and creating new points by offsetting these coordinates and converting back to the Cartesian system. The four created points $n_0 - n_3$ correspond to the four directions North, East, South and West.

$$\theta = \arctan\left(\frac{z}{x}\right), \phi = \arccos(y)$$

$$n_0 = \begin{pmatrix} \cos(\theta) * \sin(\phi - \epsilon) \\ \cos(\phi - \epsilon) \\ \sin(\theta) * \sin(\phi - \epsilon) \end{pmatrix}, n_1 = \begin{pmatrix} \cos(\theta + \epsilon) * \sin(\phi) \\ \cos(\phi) \\ \sin(\theta + \epsilon) * \sin(\phi) \end{pmatrix} \quad (3)$$

$$n_2 = \begin{pmatrix} \cos(\theta) * \sin(\phi + \epsilon) \\ \cos(\phi + \epsilon) \\ \sin(\theta) * \sin(\phi + \epsilon) \end{pmatrix}, n_3 = \begin{pmatrix} \cos(\theta - \epsilon) * \sin(\phi) \\ \cos(\phi) \\ \sin(\theta - \epsilon) * \sin(\phi) \end{pmatrix}$$

After having scaled these four points in the vertex shader we form vectors from the current vertex position to each neighbour position and calculate four separate normals by applying the cross-product to two adjacent vectors. We sum these four normals and renormalize the result to get the final vertex normal vector.

$$\overrightarrow{normal} = \left(\sum_{i=1}^4 (\overrightarrow{scaled_n_i} - \overrightarrow{center}) \times (\overrightarrow{scaled_n_{(i+1) \bmod 4}} - \overrightarrow{center}) \right)_0 \quad (4)$$

How can the offset ε be determined? The ideal solution would be to sample a new scale-value for each neighbour-vertex.

$$\varepsilon = \arctan(\tan(\varphi) + \delta) - \varphi \mid \delta = \frac{2}{\text{cubeedge}} \wedge \varphi \in]-\frac{\pi}{4}; \frac{\pi}{4}[\quad (2)$$

Although this function can be implemented in a vertex shader it is relatively complex. A better and easier solution is to choose a fixed ε for all vertices: $\varepsilon = \frac{4}{180}\pi$ provided good results in the sample application for a scale-map edglength of 128 pixels and with linear filtering enabled.

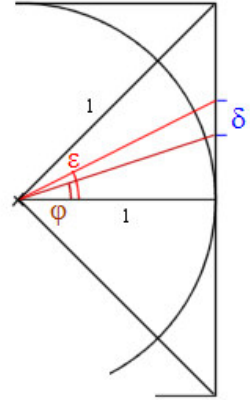


Fig. 3. Choosing ε so, that a new texel is sampled for each vertex.

4. RESULTS

Figure 4 shows an implementation of the proposed technique using a software rasterizer: A hybrid approach of image-based lighting [6] and standard phong per-pixel lighting [7] has been implemented. The base-sphere consists of about 2500 triangles; the cubemap has been exported from Milkshape [8] using a custom-plugin that takes care of the preprocessing step and directly outputs the grayscale cubemap (8-bits per channel). The software rasterizer is capable of rendering the scene at an average of 14 frames per second on a Pentium IV 2,8 GHz – resolution 320x240. An implementation of the technique on next-gen graphics hardware should run considerably faster; therefore the current software-only implementation should be seen as proof-of-concept.

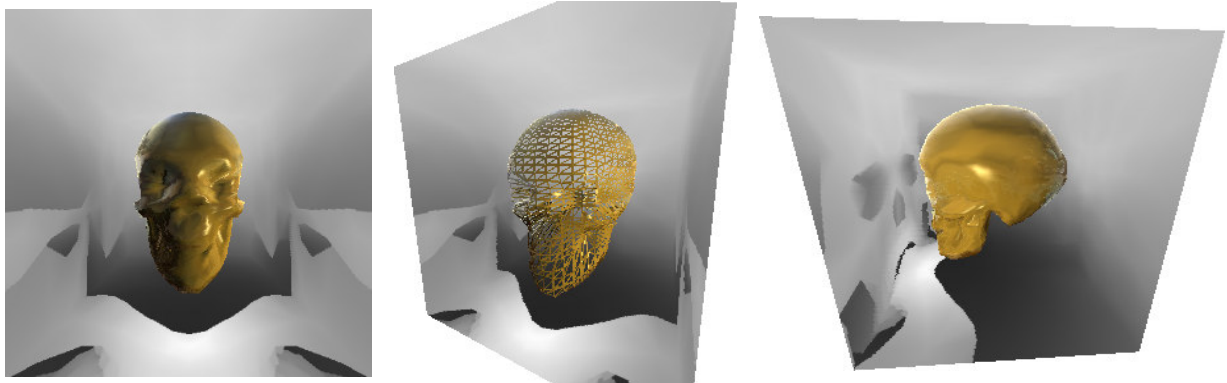


Fig. 4. Screenshots showing the reconstruction of a polygonal model from a scale map.



Fig. 5. Skull rendered with reconstructed normals as color.

5. LIMITATIONS

While the proposed technique builds a solid base for interpolation between models, it comes with limitations concerning the shape of the source models: The employment of a sphere as base-primitive and the idea of offsetting its vertices along their normals limits this approach to convex models.

In the preprocessing step the original model's surface is scanned by casting rays through the bounding sphere's origin and calculating intersection points with the model's triangles. Although this algorithm works for convex models, because it would detect a single intersection point per ray, it fails when being applied to concave models. Figure 6 illustrates the problem of multiple intersection points.

Because we can only select one of these points for the calculation of the scale factor the overhang of the concave shape could not be captured. The reconstructed model would therefore not resemble the original shape.

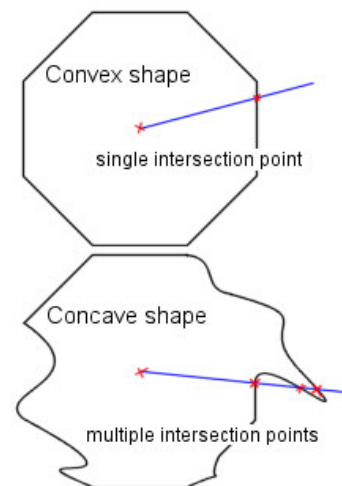


Fig. 6. Detection of surface-ray intersection points.

Another limitation associated with the presented technique is related to the representable range of surface slopes: Spheres have to be approximated with triangles, which adds a discrete nature to the base-primitive. Because vertices can only be moved along their normal, steep geometric slopes cannot be recreated from the cubemap-data if the base-sphere is not tessellated well enough.

6. CONCLUSIONS

This paper presents a technique for model-encoding into cubic scale-maps and the reconstruction of the model including normals. The proposed algorithm greatly simplifies the interpolation of closed, convex polygonal models with different amounts of triangles and vertices through blending between two scale-maps. The technique relies on capabilities available on next-gen graphics cards: programmable vertex units with support for texture sampling (vertex-shader model 3.0). The proof-of-concept implementation was created using a software rasterizer presenting the different aspects of the algorithms.

REFERENCES

- [1] Mitchell, Jason. „Direct3D Shader Models“. Presentation at GDC 2004.
- [2] Stanford University Computer Graphics Laboratory. „The Stanford Bunny“.
- [3] Woo et al. „OpenGL Programming Guide, 2nd Edition“. pp. 57 – 62. Addison-Wesley, 1997.
- [4] Bourke, Paul. <http://astronomy.swin.edu.au/~pbourke/modelling/sphere/>
- [5] Reiter, Stephan. „Muli3D“. API version 0.63. <http://muli3d.sourceforge.net/>
- [6] BJORKE, Kevin. „Image-Based Lighting“. In *GPU Gems*, Addison-Wesley, 2004.

[7] Phong, Bui Toung. „Illumination for Computer Generated Pictures". Communications of the ACM, 1975.

[8] Cirgan, Mete. „Milkshape 3D“. <http://www.milkshape3d.com/>