Cuberille Implicit Surface Polygonization for ITK

Release 1.00

Dan Mueller[1]

[1]Philips Healthcare, Best, Netherlands

July 20, 2010

Abstract

This article describes an ITK implementation of the "cuberille" method for poloygonization of implicit surfaces. The method operates by dividing the surface into a number of small cubes called *cuberilles*. Each cuberille is centered at a pixel lying on the iso-surface and then quadrilaterals are generated for each face. The original approach is improved by projecting the vertices of each cuberille onto the implicit surface, smoothing the typical block-like resultant mesh. Source code and examples are provided to demonstrate the method.

Latest version available at the Insight Journal [http://hdl.handle.net/10380/3186]
Distributed under Creative Commons Attribution License

Contents

1 Introduction

The "cuberille" model was proposed over 30 years ago [2], and subsequently utilized for polygonization over 20 years ago [1]. The original method described three sets of orthogonal planes which divided the volume of an implicit function into equal cubes (voxels). The surface was then represented as the set of directed quadrilateral faces of the cubes lying on the iso-surface. A voxel was considered to lie on the iso-surface if the implicit function value was greater than or equal to the isovalue, and the voxel had one more neighbours less than the isovalue. The concept is easily extended to support non-uniform spacing using cuboids.

The cuberille method has a number of advantages, many deriving from its shear simplicity. First and foremost the method is simple to implement. A basic summary of the algorithm is as follows: step over all pixels, for each pixel determine if it lies on the iso-surface, center a cube on the pixel, create quadrilateral faces aligned with the cube (taking care of neighbouring pixels also on the surface). The second advantage is that the method preserves topology with the resultant surface directly representing the underlying *discrete* implicit function. Take for example an implicit function consisting of a single non-zero pixel. Using the cuberille algorithm, this function will be represented as a single cube with six faces. The resultant polygonization is similar to dual contouring methods which produce vertices at pixel corners rather than pixel centers.

Of course the method also has a number of drawbacks, the most notable of which is the block-like nature of the resultant mesh. This issue can be overcome by projecting each vertex onto the iso-surface using a gradient descent method[1]. Some may also perceive the large number of vertices/cells as a disadvantage, however this can be alleviated using a decimation post-processing step. (NOTE: The current implementation does *not* support `itk::QEMesh`. This issue is being worked on.)

[1] See http://www2.imm.dtu.dk/~jab/gallery/polygonization.html .

Distributed under Creative Commons Attribution License

2 Description

The main algorithm for generating quadrilateral faces on the cuberilles is as follows:

```
// Iterate image
foreach pixel in input {
    // Determine if the current pixel is suitable
    if pixel value < isovalue
        continue
    // Compute which faces (if any) have quads
    foreach face in cube {
        faceHasQuad = neighbouring pixel value < isovalue
        if faceHasQuad {
            vertexHasQuad = get vertices from face
        }
    }
    // Process each face
    if at least one face {
        // Create vertices
        foreach vertex in vertexHasQuad {
            if vertex does not already exist {
                add vertex
            }
        }
        // Create faces
        foreach face in faceHasQuad {
            add face
        }
    }
}
```

For each pixel the algorithm needs to store two boolean arrays: the first array flags which faces have quadrilateral cells (`faceHasQuad`), and the second array flags which vertices are in use by the cells (`vertexHasQuad`). Figure 1 depicts the numbering scheme used to index these arrays.

The main drawback of the original cuberille method is the block-like nature of the resultant mesh. This can be overcome by projecting each vertex onto the iso-surface. The following pseudocode describes the gradient descent method used to project each vertex:

```
step = maximum spacing * 0.25
while ( true ) {
    // Compute normal vector
    normal = compute normal vector at vertex

    // Compute whether vertex is close enough to iso-surface value
    if ( abs(pixel value - isovalue) < threshold )      break

    // Step along the normal towards the iso-surface value
    sign = ( pixel value < isovalue ) ? +1.0 : -1.0
    vertex += ( normal * sign * step )
    step *= relaxation factor
    if ( step < threshold )   break
}
```


Distributed under Creative Commons Attribution License

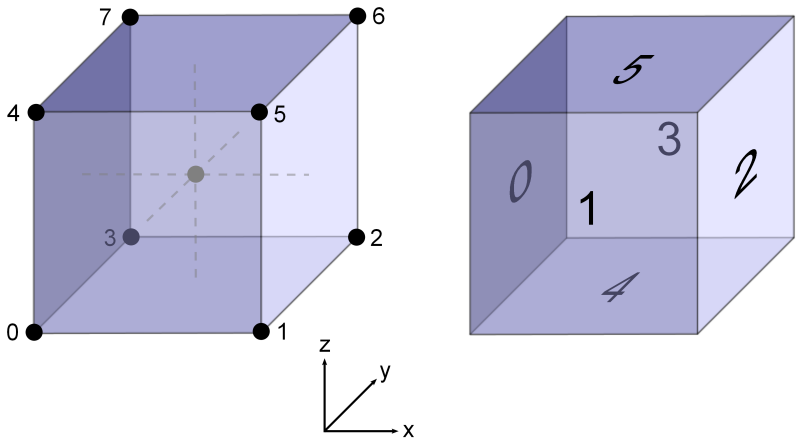


Figure 1: Depiction of numbering scheme used by cuberille algorithm.

3 Implementation

The cuberille filter `itk::CuberilleImageToMeshFilter` is a subclass of `itk::ImageToMeshFilter`. The filter has two required parameters and six optional parameters. The two required parameters are:

Input specifies the input image containing the implicit surface for polygonization.

IsoSurfaceValue specifies the value of the iso-surface for which to generate the mesh. Pixels equal to or less than this value are considered to lie on the surface or inside the resultant mesh.

The optional parameters are divided into two groups. The first group consists of two boolean flags which turn on or off different parts of the algorithm. Figure 2 depicts resultant meshes generated using these parameters.

GenerateTriangleFaces specifies whether triangle or quadrilateral faces should be generated. The default is to generate triangle faces.

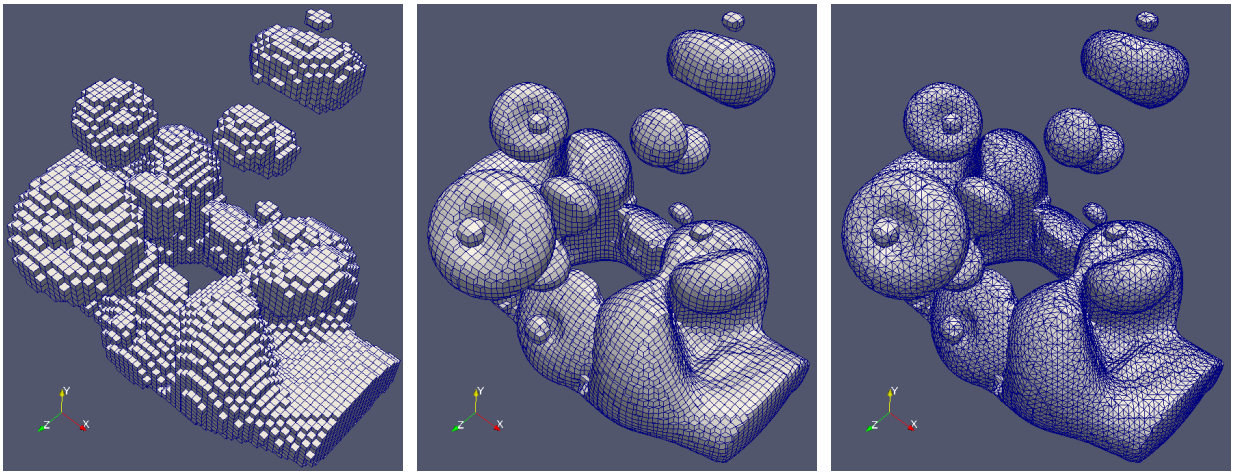
ProjectVerticesToIsoSurface specifies whether the vertices should be projected onto the iso-surface. If the projection is disabled, the resultant mesh exhibits the traditional blocky features. Projection takes roughly half of the algorithm time. The default is to project the vertices.

The remaining four optional parameters are all related to the vertex projection step. The default values seem to work well for a variety of inputs, so typical usage will not require these parameters to be set.

ProjectVertexSurfaceDistanceThreshold specifies the threshold for the "distance" from iso-surface during vertex projection. Note that the distance is actually measured in pixel value units (not space). The smaller this value, the closer the vertices will be to the iso-surface. Small values result in longer convergence time (i.e. slower). Values are clamped to the range [0.0, max pixel value]. The default value is 0.5.

ProjectVertexStepLength specifies the threshold for the step length during vertex projection. The smaller this value, the more likely the vertices will end up closer to the surface. Small values cause the

Distributed under Creative Commons Attribution License



(a) GenerateTriangleFaces=false, ProjectVerticesToIsoSurface=false

(b) GenerateTriangleFaces=false, ProjectVerticesToIsoSurface=true

(c) GenerateTriangleFaces=true, ProjectVerticesToIsoSurface=true

Figure 2: Neghip data set (http://www.volvis.org).

projection to take longer to converge. Values are clamped to the range [0.0, large]. The default value is max spacing $\times$ 0.25 (expressed in physical space).

ProjectVertexStepLengthRelaxationFactor specifies the step length relaxation factor during vertex projection. The step length is multiplied by this factor each iteration to allow convergence. Values are clamped to the range [0.0, 1.0]. The default value is 0.95.

ProjectVertexMaximumNumberOfSteps specifies the maximum number of steps used during vertex projection. The default value is 50.

Distributed under Creative Commons Attribution License

Dataset	Size	Vertices	Cells	Time (s)
Fuel	64 64 64	5,281	10,568	0.08
Neghip	64 64 64	15,146	30,272	0.20
Engine	258 258 112	319,918	640,112	3.37
Bunny	512 512 361	1,022,508	2,045,068	24.09

Table 1: Results for various datasets. Timings performed on Intel Core 2 Duo, 2 × 2.53GHz, 3.45 GB RAM, Windows XP SP3.

4 Examples

The cuberille algorithm is quite easy to use:

```
// Typedefs
const unsigned int Dimension = 3;
typedef unsigned char PixelType;
typedef itk::Image< PixelType, Dimension > ImageType;
typedef itk::Mesh< PixelType, Dimension > MeshType;
typedef itk::LinearInterpolateImageFunction< ImageType > InterpolatorType;
typedef itk::CuberilleImageToMeshFilter< ImageType, MeshType, InterpolatorType > CuberilleType;

// Create output mesh
MeshType::Pointer outputMesh = NULL;

// Create cuberille mesh filter
CuberilleType::Pointer cuberille = CuberilleType::New();
cuberille->SetInput( input );
cuberille->SetIsoSurfaceValue( IsoSurfaceValue );
cuberille->Update();
outputMesh = cuberille->GetOutput();
```

As shown in Figure 3, the use of a B-Spline interpolator can improve the resultant mesh quality (though this is at the cost of performance):

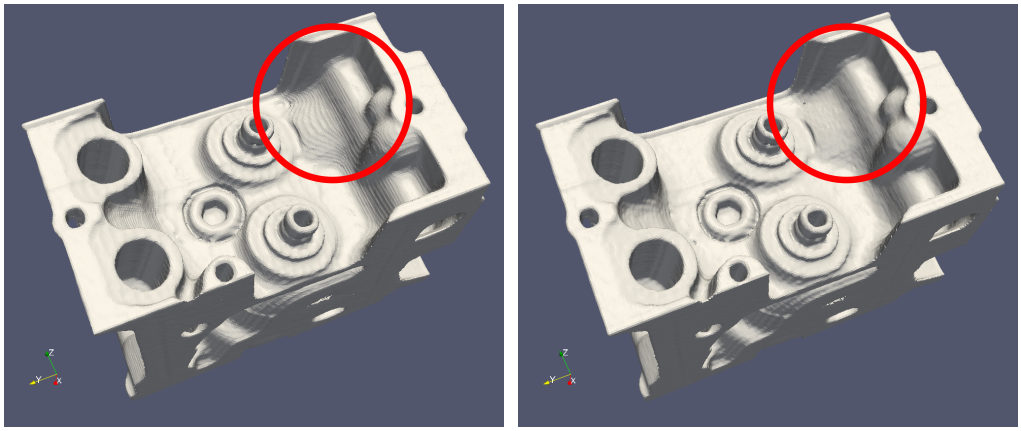
```
typedef itk::BSplineInterpolateImageFunction< ImageType,       float, float > InterpolatorType;
InterpolatorType::Pointer interpolator = InterpolatorType::New();
interpolator->SetSplineOrder( 3 );
cuberille->SetInterpolator( interpolator );
```

A number of figures have been included to show the output of the filter, see Figure 4, Figure 5, and Figure 6. Table 1 lists some simple results concerning the algorithm.

5 Conclusion

This article described the implementation of the cuberille polygonization algorithm for ITK. The algorithm is simple, relatively fast, and produces visually appealing results (when vertex projection is enabled). Currently the algorithm does not support `itk::QEMesh`, however, with the help of the Insight developers, hopefully this issue will be resolved shortly.

Distributed under Creative Commons Attribution License



(a) Linear interpolator (3.41 s) (b) B-Spline interpolator (10.95 s)

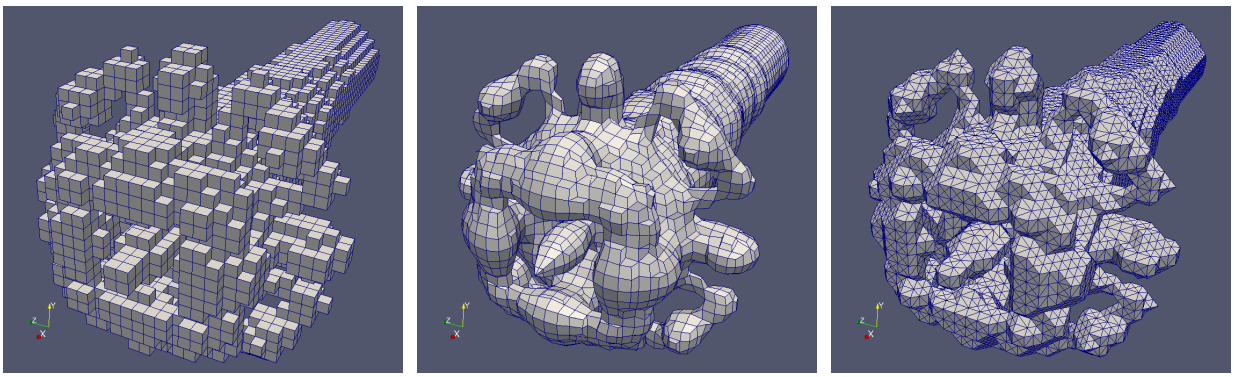
Figure 3: Comparison of linear and B-Spline interpolators. Timings performed on Intel Core 2 Duo, 2 × 2.53GHz, 3.45 GB RAM, Windows XP SP3.

References

[1] D. Gordon and J. K. Udupa. Fast surface tracking in three-dimensional binary images. *Computer Vision, Graphics and Image Processing*, 45:196–214, 1989. 1

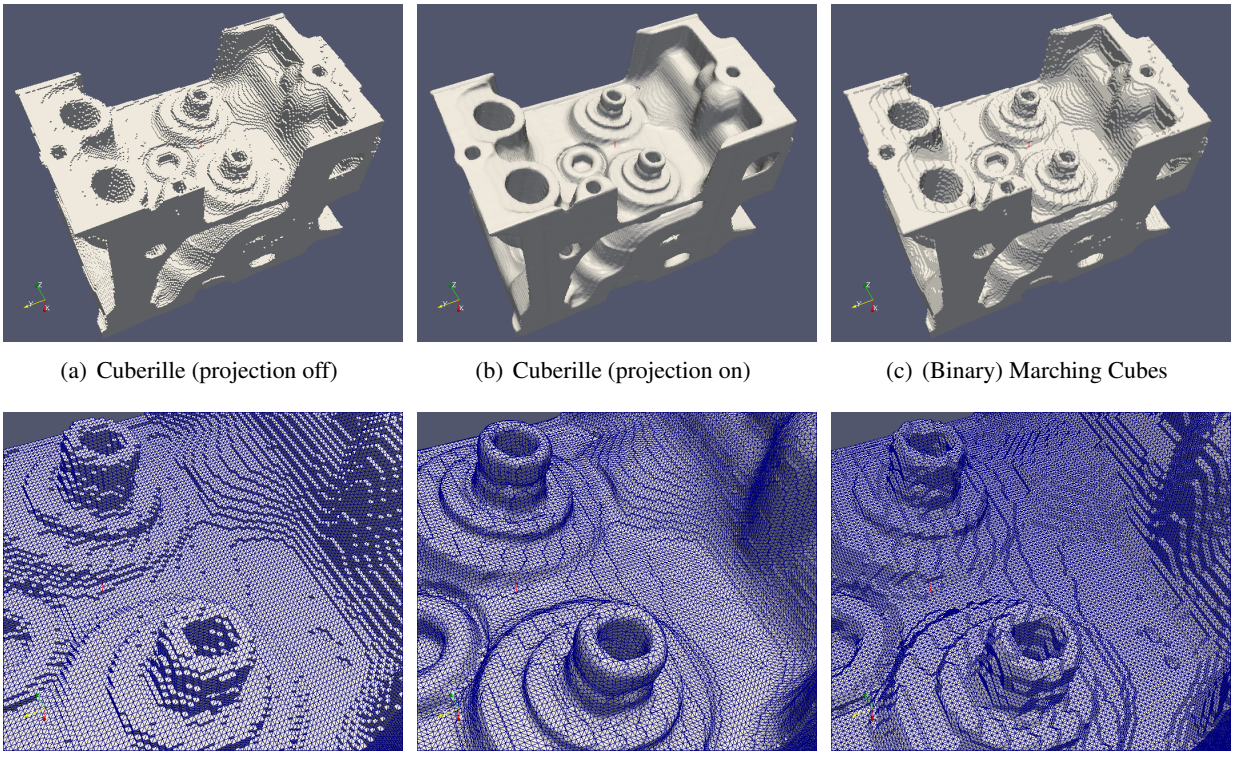
[2] G. Herman and H. Liu. Three-dimensional display of human organs from computed tomograms. *Computer Graphics and Images Processing*, 9:1–21, 1979. 1

Distributed under Creative Commons Attribution License



(a) Cuberille (projection off) (b) Cuberille (projection on) (c) (Binary) Marching Cubes

Figure 4: Fuel data set (http://www.volvis.org).

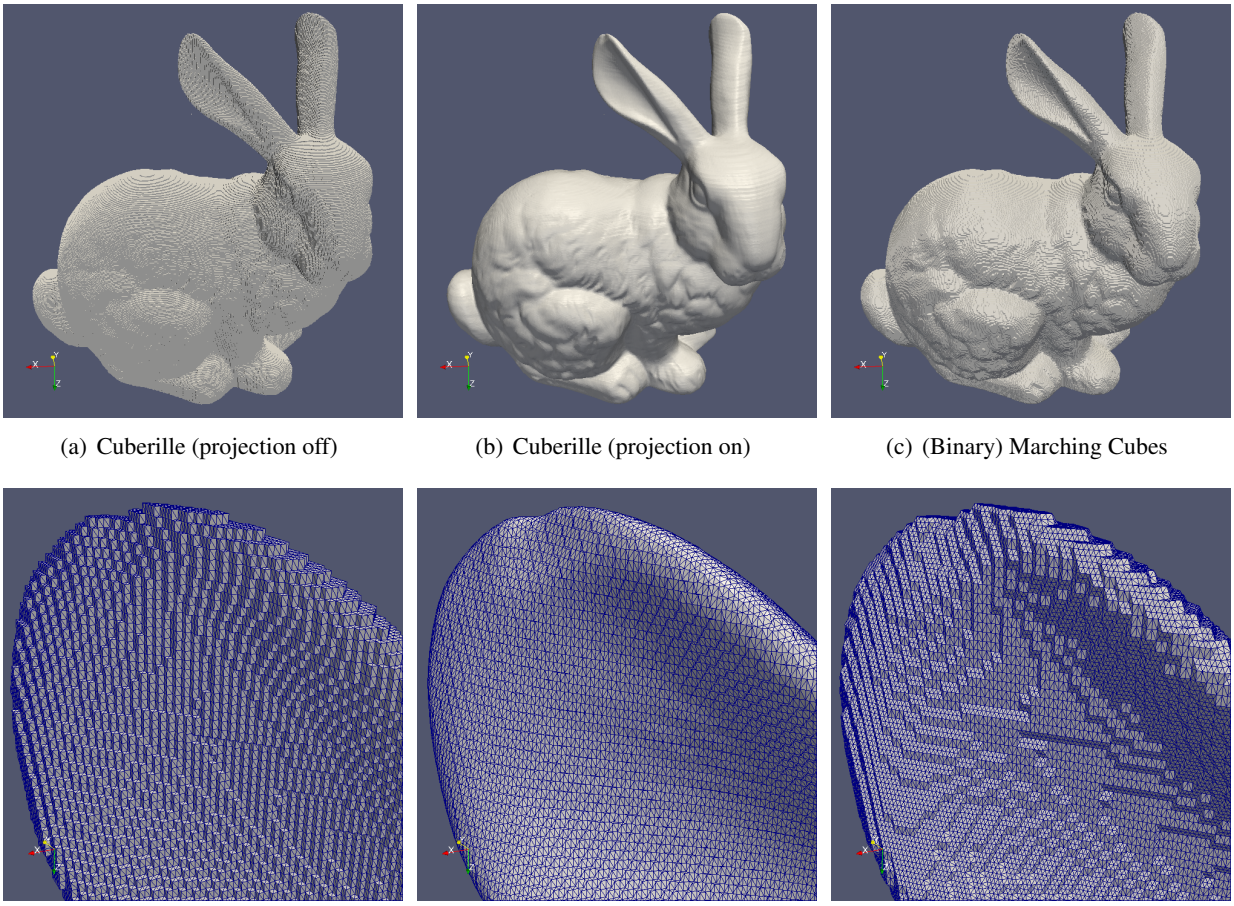


(a) Cuberille (projection off) (b) Cuberille (projection on) (c) (Binary) Marching Cubes

(d) Cuberille (projection off) (e) Cuberille (projection on) (f) (Binary) Marching Cubes

Figure 5: Engine data set (http://www.volvis.org).

Distributed under Creative Commons Attribution License



(a) Cuberille (projection off) (b) Cuberille (projection on) (c) (Binary) Marching Cubes

(d) Cuberille (projection off) (e) Cuberille (projection on) (f) (Binary) Marching Cubes

Figure 6: Bunny data set (http://graphics.stanford.edu/data/voldata/).

Distributed under Creative Commons Attribution License