Polygonal-Functional Hybrids for Computer Animation and Games

D. Kravtsov, O. Fryazinov, V. Adzhiev, A. Pasko, P. Comninos

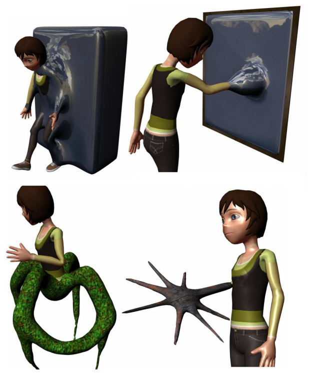
1.1 Introduction

The modern world of computer graphics is mostly dominated by polygonal models. Due to their scalability and ease of rendering such models have various applications in a wide range of fields. Unfortunately some shape modelling and animation problems can hardly be overcome using polygonal models alone. For example, dramatic changes of the shape (involving changes of topology) or metamorphosis between different shapes can not be performed easily. On the other hand Function Representation (FRep) [Pasko et al. 95] allows us to overcome some of the problems and simplify the process of the major model modification. We propose to use a hybrid model, where we combine together both polygonal and FRep models. Hence we can take advantages of different model representations performing model evaluation entirely on the GPU. Our approach allows us to:

- produce animations involving dramatic changes of the shape (e.g. metamorphosis, viscoelastic behaviour, character modifications etc) in short times (Fig. 1.1a)

- interactively create complex shapes with changing topology (Fig. 1.1b) and specified LOD (Fig. 1.2)

- integrate existing animated polygonal models and FRep models within a single model


(a)

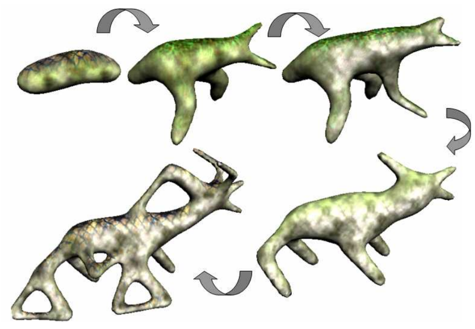

(b)

Figure 1.1. (a) Mimicked viscoelastic behaviour and hybrid characters (Model "Andy" is courtesy of John Doublestein) (b) Iterations of character growth controlled by the user

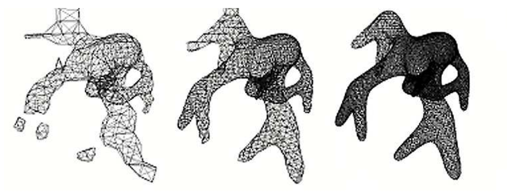


Figure 1.2. Variation of polygonization resolution

1.2 Background

1.2.1 Implicit Surfaces and Function Representation (FRep)

FRep [Pasko et al. 95] incorporates implicit surfaces and more generic types of procedural objects. Any point in space can be classified to find out if it belongs to FRep object. FRep object can be defined by a scalar function and an inequality:

$$f(\mathbf{p}) : R^3 \to R$$

$$\begin{cases} f(\mathbf{p}) > T, \mathbf{p} \text{ is inside the object} \\ f(\mathbf{p}) = T, \mathbf{p} \text{ is on the object's boundary} \\ f(\mathbf{p}) < T, \mathbf{p} \text{ is outside the object} \end{cases} \tag{1.1}$$

where $\mathbf{p}$ is an arbitrary point in 3D space and T is a threshold value (or isovalue). The subset $\{\mathbf{p} \in R^3 : f(\mathbf{p}) \geq T\}$ is called a solid object and the subset $\{\mathbf{p} \in R^3 : f(\mathbf{p}) = T\}$ is called an iso-surface (see fig. 1.3). Function $f(\mathbf{p})$ can be a signed distance field or an arbitrary scalar field.

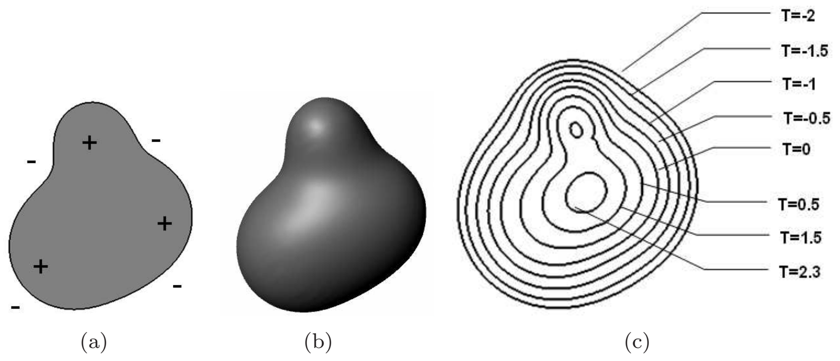


Figure 1.3. Scalar field (defining function): (a) The sign of a scalar field (b) The extracted implicit surface (*T=0*) (c) Different iso-surfaces for different values of T

The first derivative of the function can be used to compute the gradient and the normal on the object's surface:

$$\nabla f(\mathbf{p}) = (\frac{\partial f(\mathbf{p})}{\partial x}, \frac{\partial f(\mathbf{p})}{\partial y}, \frac{\partial f(\mathbf{p})}{\partial z}); \nabla f, \mathbf{p} \in R^3 \tag{1.2}$$

$$\vec{n}(\mathbf{p}) = -\frac{\nabla f(\mathbf{p})}{\|\nabla f(\mathbf{p})\|}; \vec{n}(\mathbf{p}), \mathbf{p} \in R^3 \tag{1.3}$$

Unfortunately, only a small subset of these models is well-known to a wider audience. One of the most popular types of implicit surfaces are blobs [Blinn 82] (also known as metaballs or soft objects). Individual blobby objects defined by positions of their centers and radii can be smoothly blended with each other. Superposition of these simple primitives provides an opportunity to build more complex shapes with changing topology, which is usually hard to achieve with purely polygonal models.

Implicit objects are also known for easy definition of metamorphosis sequences (also known as morphing). One only needs to interpolate between values of two signed distance fields to retrieve an intermediate object (fig. 1.4):

$$f(\mathbf{p}, t) = f_\alpha(t) \cdot f_1(\mathbf{p}) + f_\beta(t) \cdot f_2(\mathbf{p}); \mathbf{p} \in R^3, t \in R$$

where $f_\alpha(t)$ and $f_\beta(t)$ are continuous scalar functions. Parameter t is usually defined on the $[0; 1]$ interval and interpolating functions $f_\alpha(t)$, $f_\beta(t)$

are chosen to satisfy following constraints:

$$f(\mathbf{p},0) = f_1(\mathbf{p}); f(\mathbf{p},1) = f_2(\mathbf{p})$$

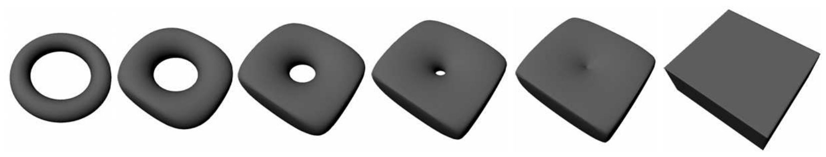


Figure 1.4. Metamorphosis sequence

Another known advantage of implicit surfaces is an easy implementation of Constructive Solid Geometry (CSG) operations [Ricci 73]. Arbitrary objects can be combined together to produce shapes of high complexity. R-functions-based CSG operations preserving C^1-continuity were applied in FReps [Pasko et al. 95]:

$$S_1 \cup S_2 = f_1(\mathbf{p}) + f_2(\mathbf{p}) + \sqrt{f_1^2(\mathbf{p}) + f_2^2(\mathbf{p})}$$
$$S_1 \cap S_2 = f_1(\mathbf{p}) + f_2(\mathbf{p}) - \sqrt{f_1^2(\mathbf{p}) + f_2^2(\mathbf{p})}$$

Preserving C^1-continuity of the resulting function may be especially important to overcome rendering artifacts- when estimating gradient of the scalar field produced by the object. Another important operation available in FRep is *blending union.* This operation allows to perform smooth blending between two objects controlling the shape of the resulting object:

$$f_b(f_1, f_2) = f_1 + f_2 + \sqrt{f_1^2 + f_2^2} + \frac{a_0}{1 + \left(\frac{f_1}{a_1}\right)^2 + \left(\frac{f_2}{a_2}\right)^2}$$

where a_1 controls the contribution of the first object, a_2 controls the contribution of the second object and a_0 controls the overall "shift" from the resulting object. Blending set-theoretic operations allow us to dramatically change the resulting shape controlling the influence of each of the initial shapes being blended, as well as controlling the overall offset from the resulting shape (see fig. 1.5).

This is a small subset of FRep features that we will use to show a number of interesting applications later.

1.2.2 Convolution surfaces

Aforementioned metaballs can be considered a subset of so called *convolution surfaces* [Bloomenthal and Shoemake 91]. These surfaces are defined by a lower-dimensional skeleton and a function defining surface profile:

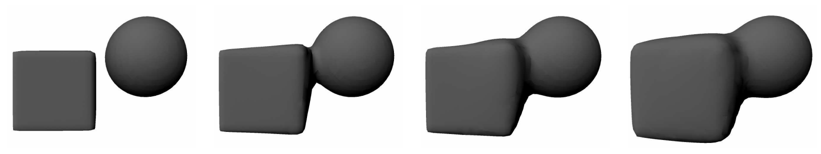

(a) (b) (c) (d)

Figure 1.5. Changing blending parameters

$$f(\mathbf{p}) = \int\limits_{R^3} g(\mathbf{r})h(\mathbf{p}-\mathbf{r})d\mathbf{r} = g \otimes h$$

where $g(\mathbf{r})$ defines the geometry of the primitive (i.e. the skeleton function), $h(\mathbf{p})$ is a kernel function (similar to various potential functions used for metaballs). $g(\mathbf{r})$ equals to "1", if point r belongs to the skeleton and equals to "0" everywhere else. Resulting convolution surface $f(\mathbf{p}) = T$ also depends on the threshold value T.

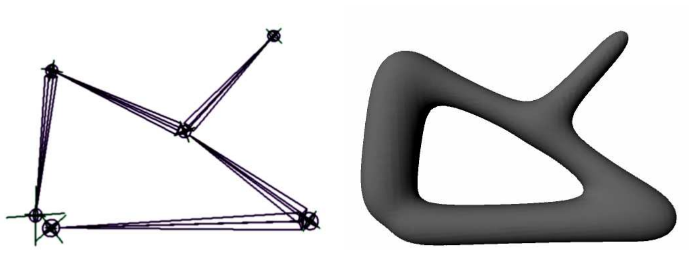

(a) (b)

Figure 1.6. Convolutions surfaces: (a) underlying skeleton (b) produced convolution surface

Convolution surfaces exhibit an important superposition property:

$$(g_1 + g_2) \otimes h = (g_1 \otimes h) + (g_2 \otimes h) \tag{1.4}$$

This means that the field produced by two independent skeletons is the same as the field produced by the combination of these skeletons. I.e. fields produced by different skeletal elements blend together resulting in a smooth surface. Convolution surfaces can be defined by points, line segments, arcs and triangles. Analytical solutions were obtained for a number of kernel functions [McCormack and Sherstyuk 98]. We will refer to Cauchy kernel:

$$h(d) = \frac{1}{(1+s^2d^2)^2}; d > 0 \tag{1.5}$$

where d specifies the Euclidean distance from a point of interest in space and s is a scalar value controlling the radius of the convolution surface. Let us write an equation for a convolution surface produced by a line segment. Given a line segment

$$\mathbf{r}(t) = \mathbf{b} + t\mathbf{a}; 0 \leq t \leq l$$

where $\mathbf{b}$ is the segment base (position vector), $\mathbf{a}$ is the segment axis (direction vector) and l is the segment length. For an arbitrary point $\mathbf{p} \in R^3$ the squared distance between $\mathbf{r}(t)$ and $\mathbf{p}$ is then defined as:

$$d^2(t) = |\mathbf{p} - \mathbf{b}|^2 + t^2 - 2t((\mathbf{p} - \mathbf{b}) \cdot \mathbf{a})$$

A field function for an arbitrary point $\mathbf{p}$ is then defined as:

$$f(\mathbf{p}) = \int_0^l \frac{dt}{(1+s^2d^2(t))^2} =$$

$$= \frac{x}{2m^2\left(m^2+s^2x^2\right)} + \frac{l-x}{2m^2n^2} + \frac{1}{2sm^3}\left(\arctan\left[\frac{sx}{m}\right] + \arctan\left[\frac{s(l-x)}{m}\right]\right)$$

where x is the coordinate on the segment's axis,

$$x = (\mathbf{p} - \mathbf{b}) \cdot \mathbf{a}$$

$$m^2 = 1 + s^2(q^2 - x^2)$$

$$n^2 = 1 + s^2(q^2 + l^2 - 2lx)$$

According to equation 1.4 the field produced by N line segments is defined as follows

$$F(\mathbf{p}) = \sum_{i=1}^{N} f_i(\mathbf{p})$$

where $f_i(\mathbf{p})$ is the field produced by the i-th line segment. An improved version of Cauchy kernel can be used to vary the radius along the line segment [Jin et al. 01].

A few other kernels can be used as well. But polynomial kernels require windowing (i.e. limiting function values within particular intervals), resulting in less smooth convolution surfaces. Besides, evaluation procedure requires more branching instructions, which is often undesirable when performing computations on the GPU. Some other kernels with infinite support are either more computationally expensive or provide less control over the resulting shape. In this article we will only refer to convolution surfaces produced by the line segments using Cauchy kernel.

1.2.3 Rendering FRep models

Even though FReps have a lot of advantages, visualizing them is not as straightforward as visualizing polygonal models. It is often desirable to convert FRep object to a polygonal mesh for efficient rendering. One of the well-known methods used for the extraction of isosurfaces from a scalar field is called Marching Cubes [Lorensen and Cline 87]. There are a number of other methods solving the same problem, but Marching Cubes is still popular due to its high speed and ease of implementation. Texturing of isosurfaces requires additional attention as well. Traditional UV-unwrapping is not suitable for complex dynamic models. Known parameterization methods can be applied to calculate UV coordinates of the extracted surface in real-time (for instance, spherical or cubic projection often used for rendering of liquid substances). Triplanar texturing [Geiss 07] provides a better way of texturing of complex functional objects. Another option could be the usage of procedural solid textures implemented in a shader [Ebert et al. 02]. Though pure procedural textures are not always well suited for rendering of arbitrary objects.

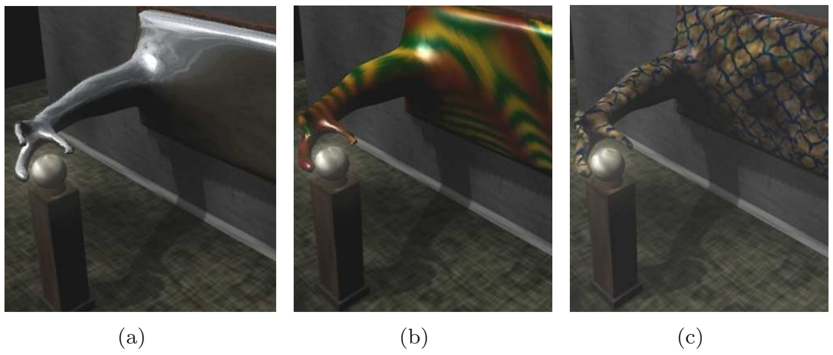


Figure 1.7. Texturing: (a) cubemap (b) procedural shader (c) triplanar texturing

1.3 Working with FRep models using GPU

Modern GPUs allow to perform evaluation and rendering of certain types of FRep models entirely on the GPU in real-time. Three main steps need to be performed:

1. Sample distance function values in the volume and save the results to a temporary buffer
2. Extract isosurface and its normals from discretized data set

3. Render extracted isosurface

Sampling of distance functions can be performed in a vertex shader [Uralsky 06] or a fragment shader. In the latter case, the volume can be sliced with a set of 2D planes or directly rendered to a volume texture. Isosurface extraction from discretized data set can be performed with the help of geometry shaders [Geiss 07, Tatarchuk et al. 07]. All of the above steps can be performed on any DirectX 10 compatible hardware. The code accompanying this article is based on NVIDIA CUDA SDK [NVIDIA 09], which allows performing generic computations on the GPU without the necessity to overcome limitations of the existing graphics APIs. Moreover, CUDA SDK already includes an illustrative example of Marching Cubes running on the GPU. We used this code as a starting point for the implementation of our approach running on the GPU.

In the following sections we will describe each of the aforementioned steps in detail.

1.3.1 Function evaluation

First of all, model parameters need to be updated and uploaded to the GPU. These parameters are stored in the constant memory and need to be modified before model evaluation. `cudaMemcpyToSymbol` function can be used for this purpose:

```
// parameters of the segments defining convolution surface
__constant__
CONVOLUTION_SEGMENT segmentsOnDevice[segmentsNumMax];
// other parameters of the model
...
// copy segments from CPU to constant GPU memory
cudaMemcpyToSymbol( segmentsOnDevice, segmentsOnHost,
                    sizeof(CONVOLUTION_SEGMENT));
// copy other parameters:
...
```

The volume where the defining function will be evaluated needs to be defined. This volume is uniformly divided in a number of cells according to required resolution. Values of the function at the corners of each cell are evaluated in parallel threads, one thread per value.

We need to save the sampled function values to a temporary buffer in order to avoid function re-evaluations in the future. Writing to global memory is a relatively slow operation and should only be performed if computationally expensive functions are evaluated. Otherwise, the time required to save and load the results may appear to be significantly higher than the time needed for function evaluation (see section 1.3.2). Depending on the computational complexity of the function being evaluated it may be beneficial to perform more than one evaluation in a kernel and store temporary results in a shared memory. After that temporary results need to be copied from shared memory to global memory in one instruction, thus achieving coalescing.

In many circumstances high-precision function evaluation is not required and faster math intrinsics available in CUDA can be used. They can either be called directly or automatically enabled via `use_fast_math` CUDA compiler option (refer to CUDA documentation for more details on the topic).

1.3.2 Isosurface extraction

We have already mentioned that Marching Cubes (MC) algorithm is commonly used to extract an isosurface from the scalar field. The MC algorithm works with individual cells uniformly distributed in the volume. The size and the number of the cells is determined by the required quality of the resulting isosurface. The algorithm allows us to find a set of polygons representing surface patch of the functional object enclosed in each individual cell. Each cell is handled by an independent thread. Here are a number of steps required to efficiently extract an isosurface on the GPU:

1. For each cell
   (a) write out the number of vertices it contains
   (b) write out the flag indicating whether it contains any geometry
2. Find the number of non-empty cells
3. Create a group of all non-empty cells using the flags information from step 1b
4. Generate the table of vertex buffer offsets for non-empty cells
5. For each non-empty cell
   (a) Find the number of vertices it outputs
   (b) Generate vertices of the triangles being output from the cell
   (c) Generate normal for each vertex being output
   (d) Save vertices and normals using offset generated at step 4

This may look complicated at first, because a number of additional issues arise when performing polygonization on parallel computing device. First of all, we want to find a set of cells that actually contain geometry in them. Usually the majority of the cells do not contain any geometry, as they are situated completely inside or outside the object, thus having no intersections with the surface of the object. It is important to discard such cells early in order to avoid redundant computations (step 3). Secondly, each non-empty cell outputs from 1 to 5 triangles. For each cell we need to know the offset in the vertex buffer where the vertices will be output. But this offset depends on the number of vertices output from preceding cells. In case of sequential MC this offset can be iteratively increased, while processing each cell one after another. But it gets more complicated when the cells are processed in parallel. This problem is solved at step 4 using CUDA Data Parallel Primitives Library [Sengupta et al. 08].

Step 1 MC case index needs to be determined, in order to find out whether a cell is empty or not. To do so, we need to retrieve function values at eight corners of each cell and determine its MC case index. At this point we use the data sampled before (see section 1.3.1):

```
// Based on original source code provided by NVIDIA Corporation

// get MC case index depending on function values
__device__ uint getMCIndex(const float* field, float threshold)
{
   uint indexMC;
   indexMC =  uint(field[0] < threshold);
   indexMC |= uint(field[1] < threshold) << 1;
   indexMC |= uint(field[2] < threshold) << 2;
   indexMC |= uint(field[3] < threshold) << 3;
   indexMC |= uint(field[4] < threshold) << 4;
   indexMC |= uint(field[5] < threshold) << 5;
   indexMC |= uint(field[6] < threshold) << 6;
   indexMC |= uint(field[7] < threshold) << 7;

  return cubeindex;
}

// sample volume data set at the specified point
__device__ float sampleVolume(uint3 point, uint3 gridSize)
{
   p.x = min(point.x, gridSize.x - 1);
   p.y = min(point.y, gridSize.y - 1);
   p.z = min(point.z, gridSize.z - 1);
   uint i = (point.z * gridSize.x * gridSize.y) +
            (point.y * gridSize.x) + point.x;
   return tex1Dfetch(volumeTex, i);
}

// output number of vertices that need to be generated for
// current cell and output flag indicating whether current
// cell contains any triangles at all
__global__ void
preprocessCell(...)
{
    ...
    float field[8];

    // retrieve function values at 8 corners of a cube
    field[0]= sampleVolume(gridPos, gridSize);
    field[1]= sampleVolume(gridPos + make_uint3(1,0,0),gridSize);
    field[2]= sampleVolume(gridPos + make_uint3(1,1,0),gridSize);
    field[3]= sampleVolume(gridPos + make_uint3(0,1,0),gridSize);
    field[4]= sampleVolume(gridPos + make_uint3(0,0,1),gridSize);
    field[5]= sampleVolume(gridPos + make_uint3(1,0,1),gridSize);
    field[6]= sampleVolume(gridPos + make_uint3(1,1,1),gridSize);
    field[7]= sampleVolume(gridPos + make_uint3(0,1,1),gridSize);

    // find out case index in the MC table
    uint indexMC =  getMCIndex(field, threshold);

    // read number of vertices produced by this case
    uint numVerts = tex1Dfetch(numVertsTex, indexMC);

    if (cellIndex < numCells) {
        // save the number of vertices for later usage
        cellVerts[cellIndex] = numVerts;
        // flag indicating whether this cell outputs any triangles
        cellsNonEmpty[cellIndex] = (numVerts > 0);
    }
}
```

where `numVertsTex` is a table containing the number of triangle vertices contained in a cell corresponding to certian MC case and `volumeTex` is the sampled volume data that was earlier bound to a 1D texture:

```
struct cudaChannelFormatDesc channelDesc =
    cudaCreateChannelDesc( 32,0,0,0,cudaChannelFormatKindFloat );

    cudaBindTexture(0, volumeTex, d_volume, channelDesc);
```

It is preferable to fetch data from textures rather than from global device memory as in this case texture cache can be utilized to reduce memory access times.

Step 2 Once output vertex information for each cell has been retrieved, scan algorithm (also known as “Parallel Prefix Sum”) can be used. This operation allows to generate an array, in which each element contains the sum of all preceding values of the input array (see fig. 1.8) [Sengupta et al. 07].

In our case input array `d_cellsNonEmpty` contains either “0” (empty cell) or “1” (non-empty cell). Hence each element of the array generated by an exclusive scan operation (`d_cellsNonEmptyScan`) applied to the input array contains the number of non-empty cells preceding it. The values of such elements can be interpreted as sequential indices of non-empty cell.

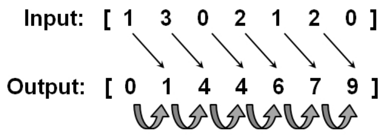


Figure 1.8. Exclusive scan

The last element of the generated array equals to the total number of all non-empty cells except the last one:

```
// scan array of non-empty cells
cudppScan( scanPlanExclusive, d_cellsNonEmptyScan,
           d_cellsNonEmpty, numCells);
// copy the value of the last element from the GPU
uint nonEmptyCellNumber, lastCellIsEmpty;
cudaMemcpy((void *) &nonEmptyCellNumber,
           (void *) (d_cellsNonEmptyScan + numCells - 1),
           sizeof(uint), cudaMemcpyDeviceToHost);
// add the value from the last cell as it may be non-empty too
cudaMemcpy((void *) &lastCellIsEmpty,
           (void *) (d_cellsNonEmpty + numCells - 1),
           sizeof(uint), cudaMemcpyDeviceToHost);

// final number of non-empty cells
nonEmptyCellNumber += lastCellIsEmpty;
```

Step 3 Stream compaction (aka enumerate operation) is used to generate an array containing indicies of only non-empty cells. “Stream compaction” operation requires two input arrays. First array contains boolean values indicating whether respective elements from the second array need to be copied to the output array. Example input datasets and output generated by this operation are shown in fig. 1.9. In our case we provide an array of flags `d_cellsNonEmpty` and a “scanned” array of non-empty cell indices `d_cellsNonEmptyScan`:

```
cudppCompact(compactPlan, d_compactedCells,d_nonEmptyCellNumber,
               d_cellsNonEmptyScan, d_cellsNonEmpty, numCells);
```

After this step `d_compactedCells` contains the set of indices of all non-empty cells.

Step 4 As was mentioned earlier, generation of vertex buffer offsets for each cell is performed using scan operation (similar to step 2). We apply an exclusive scan again, as we need each element in the array to contain the sum of previous elements excluding current element. First element in the array of offsets should be equal to “0”. After the application of scan

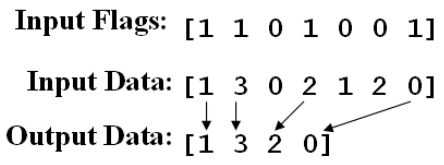


Figure 1.9. Compact algorithm

operation each element of the `d_cellVBOffsets` contains the total number of vertices contained in preceding cells (i.e. offset in the vertex buffer that can be used to output the vertices from the cell):

```
cudppScan( scanPlanExclusive, d_cellVBOffsets,
           d_cellVerts, numCells);
```

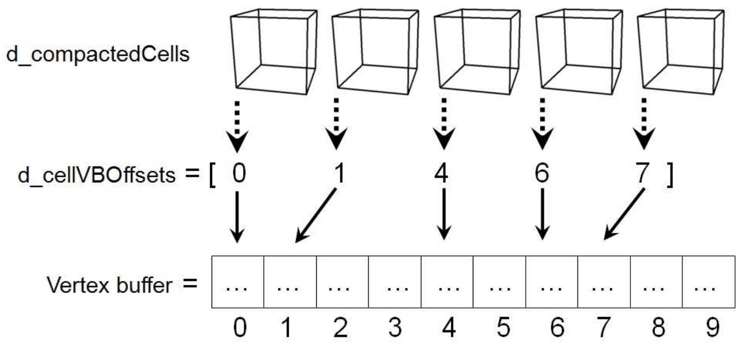


Figure 1.10. Offsets in vertex buffer for each cell

Step 5 Finally, we can start generating triangles and writing them to the specified vertex buffer. Each vertex being generated is placed along one of the 12 edges of the cell. Function value at each generated vertex is expected to be equal to the `threshold` value (in this case vertex is placed on the extracted isosurface). Thus we can linearly[1] interpolate function values along each edge in order to find locations where vertices have to be placed:

```
__device__
float3 interpolatePosition(float threshold,
                           float3 cellVertex1, float3 cellVertex2,
```

[1]Non-linear interpolation schemes can be used to improve the quality of the resulting mesh [Tatarchuk et al. 07].

```
                              float funcValue1, float funcValue2)
{
   float t = (threshold - funcValue1) /
             (funcValue2 - funcValue1);
   return lerp(cellVertex1, cellVertex2, t);
}

// write out vertices and normals
__global__ void generateTriangles(...)
{
   ...
   // vertices placed at the corners of current cell
   float3 cellVertices[8];
   ...
   // field values at the 8 corners of current cell
   float field[8];
   field[0] = sampleVolume(gridPos, gridSize);
   ...
   // array of vertices placed along 12 edges of the cell
   // shared between different threads
   __shared__ float3 vertices[12 * NTHREADS];

   // find positions of 12 vertices along all edges:
   vertices[threadIdx.x] = interpolatePosition(threshold,
                              cellVertices[0], cellVertices[1],
                              field[0], field[1]);

   vertices[NTHREADS+threadIdx.x] =
           interpolatePosition(threshold,
                               cellVertices[1], cellVertices[2],
                               field[1], field[2]);
   ...
   // last vertex
   vertlist[(NTHREADS*11)+threadIdx.x] =
           interpolatePosition(threshold,
                               cellVertices[3], cellVertices[7],
                               field[3], field[7]);
   // wait while threads are filling "vertices" buffer
   __syncthreads();
   ...
```

You can see that `vertices` array is placed in shared memory. This is done to decrease the amount of local storage required to run the kernel. Additionally, this memory is accessed in a special way. Each vertex of the cell is placed with a stride of `NTHREADS` elements. Thus helping to avoid bank conflicts between the threads, i.e. consecutive threads access consecutive memory addresses and such memory requests can be serialized[2]. These are well-known optimization techniques often used in CUDA applications.

Once all output vertex positions have been retrieved they need to be connected to form a set of triangles. We need to find MC case index again.

[2]This need not be done for devices with CUDA Compute Capability 1.2 and above

This index is used to read the set of vertex indices from the MC triangles table. After this step a set of vertices and normals can be output to a vertex buffer:

```
...
// get the number of triangles that need to be output
// in this MC case
uint indexMC = getMCIndex(field, threshold);
uint numVerts = tex1Dfetch(numVertsTex, indexMC);

for(int i=0; i < numVerts; i++) {

   // find the offset of this vertex in the vertex buffer
   uint vertexOffset = cellVBOffsets[cellIndex] + i;

   if (vertexOffset >= maxVerts) {
      continue;
   }

   // will get the vertex from the appropriate
   // edge of the cell
   uint edge = tex1Dfetch(triTex, (indexMC << 4) + i);

   // write out vertex position to VB
   float3 p = vertlist[(edge*NTHREADS)+threadIdx.x];
   positions[vertexOffset] = make_float4(p, 1.0f);
   // evaluate normal at this point
   normals[vertexOffset] = calcNormal(p);
}
```

In this case MC triangles table was earlier mapped to a 1D texture `triTex` (in a fashion similar to the mapping of volume texture performed at step 1).

Per-vertex analytic normals (see equation 1.3) are retrieved using forward differences approximation:

```
__device__
float4 calcNormal(float3 p)
{
   float f = fieldFunc(p.x, p.y, p.z);
   const float delta = 0.01f;
   // approximate derivative:
   float dx = fieldFunc(p.x + delta, p.y, p.z) - f;
   float dy = fieldFunc(p.x, p.y + delta, p.z) - f;
   float dz = fieldFunc(p.x, p.y, p.z + delta) - f;
   return make_float4(dx, dy, dz, 0.f);
}
```

It is important to note that polygonization does not have to be performed for each frame. Depending on the available processing power mesh extraction can be performed only once for a number of frames. Alpha blending between the extracted meshes can be applied to perform smoother transition between them.

Note: An issuse of loading from and saving to global memory was mentioned in section 1.3.1. From the code provided in this section it can be seen, that in case function values are not stored in the memory, eight function evaluations need to be performed for each cell (see `sampleVolume()` function). For instance, on a $64 \times 64 \times 64$ grid one would need to perform at least 2 million function evaluations (that is 8 evaluations for each of 262144 cells) only to find out MC case index for each cell. Add about 10-20% of this number to get the total number of all required function evaluations. This includes actual vertex positions and normals calculation for non-empty cells. For a number of example applications that will follow saving to global memory results in better performance as memory latency is hidden by the expensive function evaluations.

1.3.3 Rendering

Once vertex buffer has been filled with the geometry information it can be rendered as any conventional polygonal model. We only need to enable a shader making the extracted isosurface look more visually interesting (see section 1.2.3).

It is worth noting that FRep models can be rendered using ray-casting [Fryazinov et al. 08], thus avoiding the necessity to perform complex isosurface extraction procedure. Though at the moment only relatively simple models can be rendered at high resolution in real-time [Kravtsov et al. 08].

1.4 Applications

In section 1.2 we have briefly described a small subset of FReps. In this section we will demonstrate a number of applications (see fig. 1.1) that can be implemented using the combination of FRep objects and poygonal meshes.

1.4.1 Approach outline

Rigging skeleton is commonly used to animate polygonal meshes. Convolution surfaces described in section 1.2.2 as well as a number of other implicit surfaces can also be animated using similar skeleton[3]. We will use a skele-

[3]We have chosen convolution surfaces mainly because of their relatively simple defining function and an absence of bulges and other unwanted artefacts

ton as a base for the integration of FReps and polygonal models together. The use cases we will refer to can be generally classified as follows :

1. Embedding FRep object inside mesh object or coating mesh objects with FRep objects (fig. 1.11)

2. Attaching FRep object to the mesh (fig. 1.11)

3. Attaching polygonal object to the FRep object (fig. 1.19)

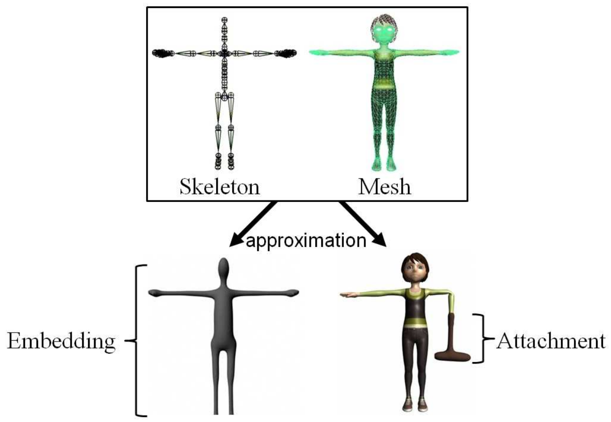


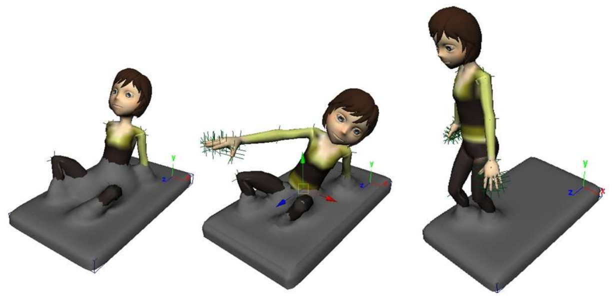
Figure 1.11. Possible approximations

We will provide detailed description of each case.

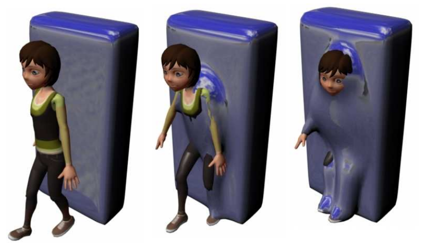
1.4.2 Embedding FRep object inside mesh object

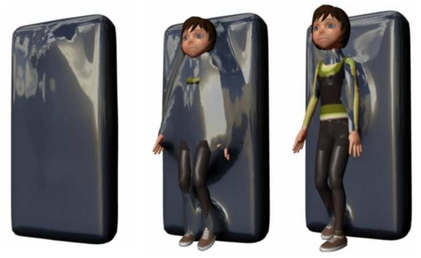
The embedding of FRep object allows us to mimic the interaction of a viscous object with an animated character (fig. 1.12) as well as "supra-natural" behaviour of the liquid material (fig. 1.16).

In this case we approximate an animated mesh with a convolution surface using the rigging skeleton. Resulting convolution surface is expected to be completely hidden inside the mesh. In most cases the approximation needs to be performed only once for the characters bind pose. We can estimate the parameters of the embedded convolution surface using the available information. For the initial approximation we use the rigging skeleton. Given the set of bones of the rigging skeleton, where each bone is a line segment in 3D space, we use the set of these segments as the basis for



(a) Preview of intermediate results as seen in the window of the tool





(b) "Sinking" (c) "Walking out"

Figure 1.12. The interaction of an animated object with viscous liquid

an initial convolution skeleton[4]. To calculate the radius of the convolution surface for each segment, we calculate the minimal distance between each line segment. For the set of rigging skeletal bones $s_i \in \mathbf{S}$ (where $\mathbf{S}$ is a set of skeletal bones) the radius of the i-th convolution surface associated with the i-th bone is:

$$r_i = \min_{p_j \in \mathbf{P}} (dist(s_i, p_j)) \ ,$$

where p_j is the j-th face of the polygonal mesh, $\mathbf{P}$ is a connected set of faces and $dist(s_i, p_j)$ is the distance between the bone s_i and the face p_j. Thus, each individual convolution surface is fitted inside the mesh in its initial position.

After the initial approximation a global optimization is usually required to achieve a better approximation of the given polygonal mesh using the embedded convolution surface (more details are provided in [Kravtsov et al. 08]). Additional embedding optimization step is usually necessary

[4]Convolution skeleton does not need to have the same configuration as the original rigging skeleton. For instance, it can have different number of bones or positions of the bones can differ. The only requirement is that convolution skeleton should be defined relative to the rigging skeleton

because only individual convolution surfaces are considered at the initial approximation step. In fact the fields produced by all convolution surfaces sum up, which is equivalent to the increase of the radius of the individual convolution surfaces. This is especially noticebale in the locations near the skeleton branches. Alternatively, instead of global non-linear optimization an artist can manipulate radius values of individual convolutions to achieve better embedding. Note that, if we wish to apply the blending union operation described in section 1.2.1, the quality of the initial approximation does not play a significant part in this process.

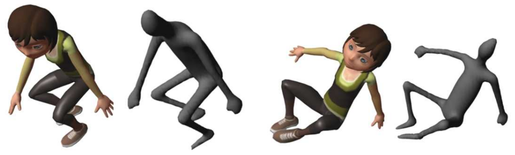


Figure 1.13. Synchonization of polygonal and functional objects

After the approximation step the segments of the convolution skeleton are transformed relative to the transformation of the rigging skeleton, hence the motion of the convolution surface is synchronized with the motion of the animated mesh (see fig. 1.13). Once the approximation of the animated mesh has been retrieved, we can apply FRep operations to achieve a number of effects (see fig. 1.14).

As the first application of our technique, we mimic the interaction of a viscous object with an animated object using the blending union of two implicit surfaces. As we mentioned above, the FRep object corresponding to the initial mesh is an embedded convolution surface. The second FRep object representing the viscous object can be modelled using a set of implicit primitives. If the defining functions of both objects have distance properties, the shape of the surface resulting from the blending operation depends on the distance between the original objects. The further the objects are from each other the less they are deformed (see fig. 1.15).

The behaviour of the blended shape visually resembles adhesion, stretching and breach of the viscous material. If the blended shape is rendered together with the polygonal mesh, a part of the convolution surface embedded within the mesh becomes visible due to the deformation, thus contributing to the material interacting with the mesh (see fig. 1.12). Thus, the quality of the initial approximation of the mesh by the convolution surface does not play a significant part in this application. It is more important just to fully embed the convolution surface into the mesh when no deformation is applied. Surely, such approach is aimed at achieving verisimilitude rather than physically correct results.

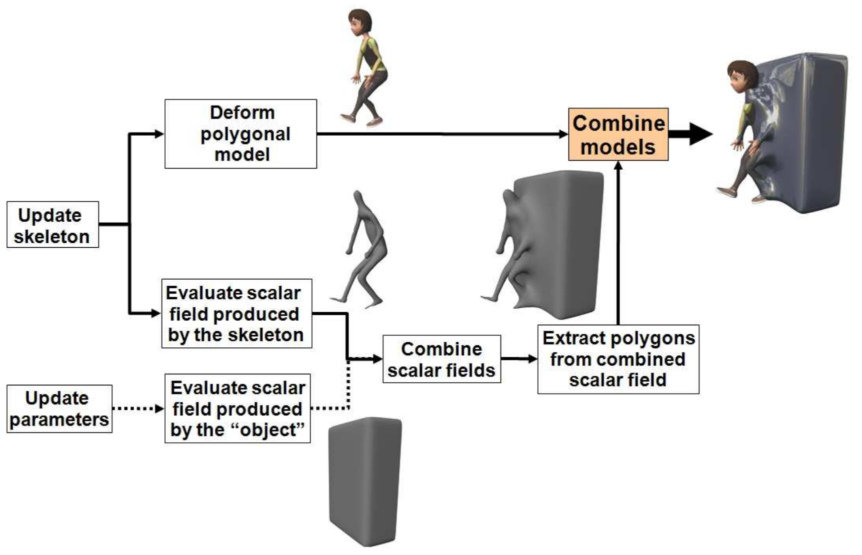

Deform polygonal model
Combine models
Update skeleton
Evaluate scalar field produced by the skeleton
Combine scalar fields
Extract polygons from combined scalar field
Update parameters
Evaluate scalar field produced by the "object"

Figure 1.14. Approach outline

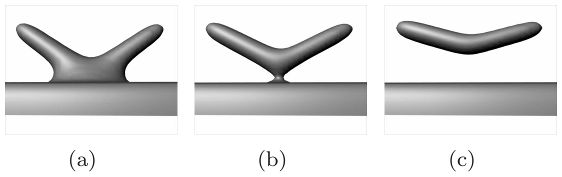
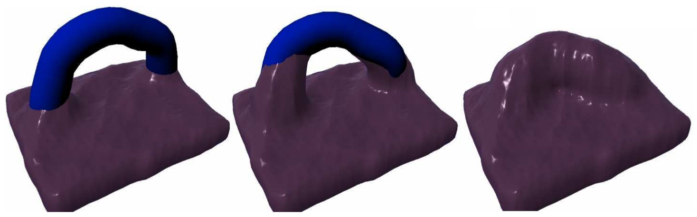
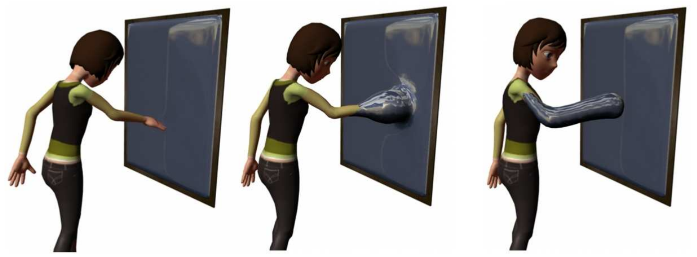


Figure 1.15. Phases of interaction between animated blended objects: (a) Two objects and a single blend shape during blending, (b) The boundary case before two shapes separate, (c) Two separate shapes with some deformation showing the objects' reciprocal attraction

The aforementioned approach can be used to model "supra-natural" behaviour of the liquid material. In such an animation effect the convolution surface radii are increased over time, which creates the effect of the liquid flowing up the mesh and gradually engulfing it. It is possible to automatically generate this sort of animation. The artist only needs to specify the first and last joint of the skeletal chain as well as the final "thickness" of the liquid flowing over the mesh (see fig. 1.16).


(a) Liquid covering animated mesh


(b) “Mirror”

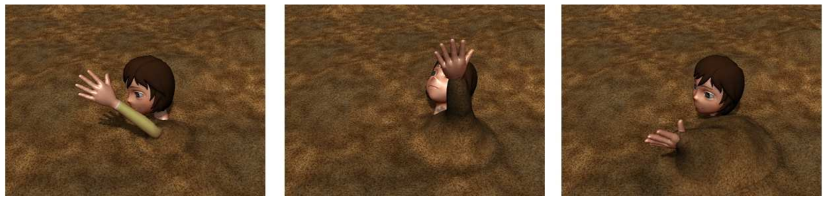

(c) “Desert”

Figure 1.16. The interaction of an animated object with viscous liquid

1.4.3 Attaching FRep object to the mesh

In this case we attach an implicit surface to the mesh. To do so, we attach a skeleton defining convolution surface to the rigging skeleton that is animated in a usual way (fig. 1.17). Optionally the implicit surface can be fitted at its boundary attachment to the polygonal mesh. Animation of the skeleton defining convolution surface leads to the automatic changes of the attached functional object, which means the resulting shape can be dramatically changed. No additional blending is required as convolution surfaces are automatically blended with each other. This approach can be used for the creation of easily metamorphosing parts of animated characters (1.18a). It is also possible to perform metamorphosis between implicit limbs with quite different geometry and topology. The user just needs to specify two skeletons (1.18b) and the time needed to morph from one to another. The intermediate meshes are generated automatically (1.18c).

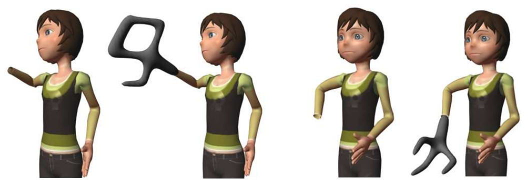
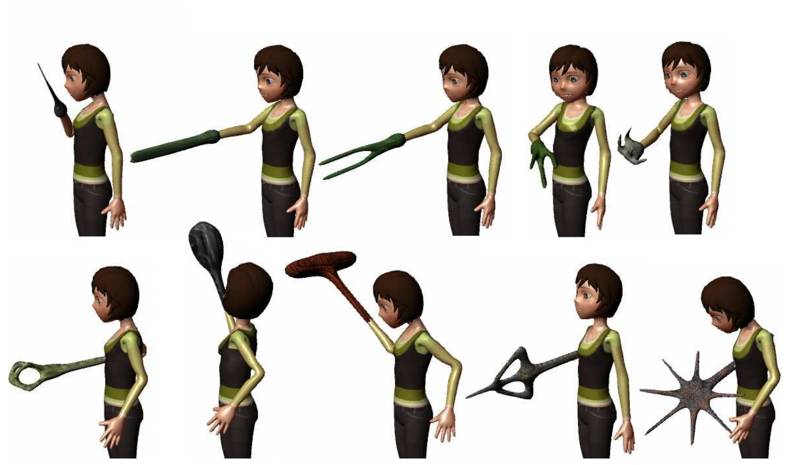
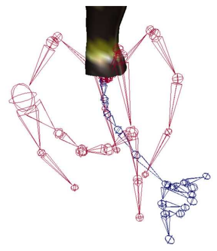


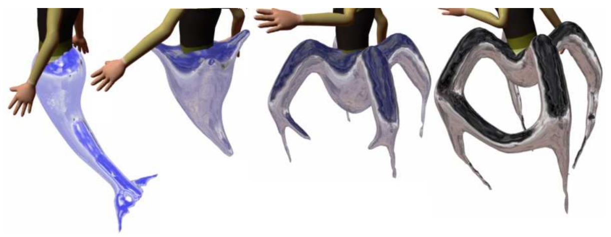
Figure 1.17. Attachment of functional object



(a) "Organic hand"



(b) "Mermaid" (blue) and "spider" (red) skeletons



(c) "Mermaid to spider"

Figure 1.18. Controlled metamorphosis sequences

1.4.4 Attaching polygonal object to the FRep object

In this case skeleton controlling functional object is defined independently. FRep object can be placed in the virtual environment as a self-contained entity. Various special effects can be implemented for this entity. The

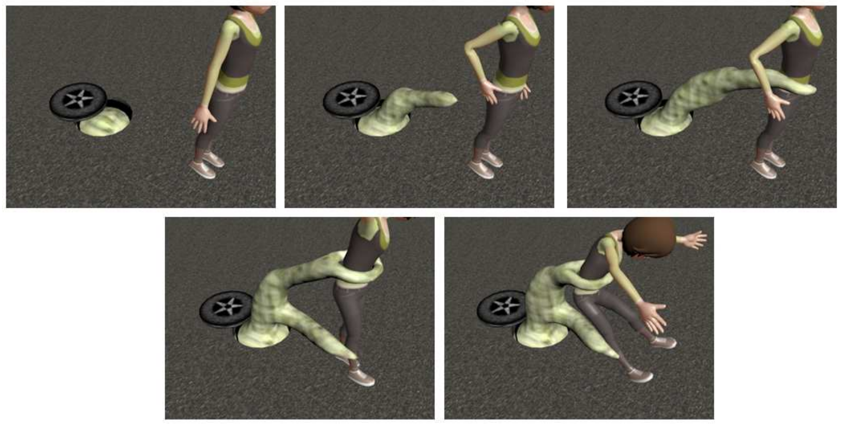


Figure 1.19. Attachment of polygonal object to the functional object

interaction with polygonal objects can be performed using the “implicit skeleton”. The polygonal objects can be attached to this skeleton and follow its motion (see fig. 1.19). Collision detection with FRep object can also be implemented in a relatively simple way, as the scalar field produced by such object has distance properties.

One can notice that implicit surfaces are a great tool in defining complex dynamic shapes with arbitrary topology. They can also be used for the creation and modification of the user generated content (similar to EA’s “Spore^{TM}”). The user can define the skeleton and tweak its parameters seeing the resulting shape in real-time (see fig. 1.1b). After the extraction of the convolution surface it can be assigned skinning weights and later used in the virtual environment. LODs for such a mesh can be generated automatically via variation of the polygonization grid resolution (fig. 1.2).

We show the times required to evaluate the field and extract the mesh in table 1.1.

1.5 Tools

Any technique loses its value if no appropriate tools are available for people who are actually producing the content. We wanted to demonstrate that the proposed approach can be employed in a conventional animation pipeline with near real-time preview without a significant effort. Thus we have implemented our approach as a plug-in for Maya^{TM}. We have chosen Maya as it is a popular tool for modelling and animation used by a lot of professional artists. Our plug-in performs polygonization on the CPU and

Grid resolution for polygonization	**“Supra-natural” (11 segments)**	**Andy (45 segments)**	**Hybrid Andy (10 segments)**
32x32x32	3 ms	9 ms	2 ms
64x64x64	7 ms	22 ms	3 ms
128x128x128	30 ms	95 ms	14 ms

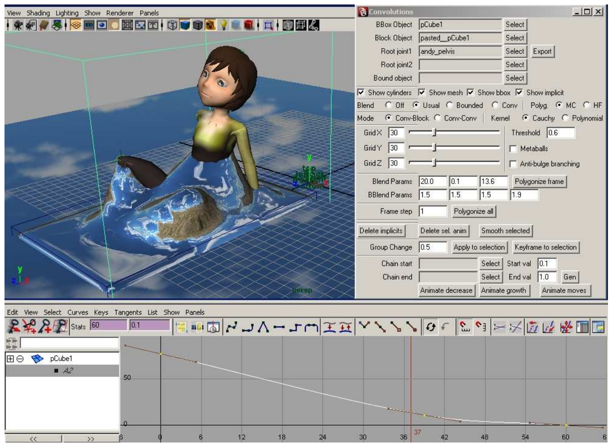
Table 1.1. Average times for mesh generation (milliseconds/frame) on an NVIDIA GeForce 8800 Ultra, 768 MB of RAM; Andy is a mesh model with an embedded convolution surface (Fig. 1.12), Hybrid Andy is shown in figure 1.18(a))

Grid resolution for polygonization	**“Supra-natural” (11 segments)**	**Andy (45 segments)**	**Hybrid Andy (10 segments)**
20x20x20	25 ms	80 ms	30 ms
30x30x30	80 ms	220 ms	60 ms
50x50x50	310 ms	930 ms	260 ms
70x70x70	810 ms	2580 ms	670 ms

Table 1.2. Average times for mesh generation (milliseconds/frame) on a PC with a Dual Core Intel Xeon (2.66 GHz), 2 GB of RAM

feeds the extracted mesh back to the modelling package. All the scenes illustrating the aforementioned use cases were defined using a set of developed plug-ins. Even though all calculations are performed on the CPU, intermediate results can be seen in the editor in near real-time (fig. 1.20). The actual times for a number of models are shown in table 1.2. Alternatively, the mesh extracted on the GPU could be copied to RAM and provided to Maya^{TM}for further manipulation and rendering.

Integration of our technique into an existing animation package can decrease the learning curve for the user. The user is free to produce an animation sequence in a way that he is accustomed to within the familiar software environment, while having an opportunity to see the results of his actions in near real-time. Thus, the incorporation of the plug-in in a general-purpose animation software package allows the user to easily integrate the produced animation into complex scenes developed using this package.


View Shading Lighting Show Renderer Panels
Edit View Select Curves Keys Tangents List Show Panels
pCube1
50
0
0 6 12 18 24 30 36 42 48 54 60
37

Convolutions

BBox Object: pCube1
Block Object: pasted__pCube1
Root joint1: andy_pelvis
Root joint2:
Bound object:
[✓] Show cylinders [✓] Show mesh [✓] Show bbox [✓] Show implicit
Blend: Off [✓ Usual] Bounded Conv — Polyg. [✓ MC] HF
Mode: [✓ Conv-Block] Conv-Conv — Kernel: [✓ Cauchy] Polynomial
Grid X: 30
Grid Y: 30
Grid Z: 30
Threshold: 0.6
[ ] Metaballs
[ ] Anti-bulge branching
Blend Params: 20.0 0.1 13.6
BBlend Params: 1.5 1.5 1.5 1.9
Frame step: 1
Group Change: 0.5
Chain start: — Start val: 0.1
Chain end: — End val: 1.0

Figure 1.20. A screenshot of the working environment

1.6 Limitations

The proposed approach has some limitations that we will discuss in this section.

The applied blending operation is based on the distance properties of the functions defining the initial geometric objects being blended. The scalar fields produced by known convolution surface kernels significantly decrease as the distance from the line segment increases. At a particular distance from the line segment the values of such a field are almost equal to zero and no blend shape is generated at these distances by the blending operation. Thus, it is hard to model the interaction between the mesh and the viscous object at large distances. In such cases, an approximation of the mesh with a set of blended quadric surfaces could provide better results. Additionally as the distance between the two blended objects increases, the deformation of the convolution surfaces decreases until these surfaces are again embedded into the polygonal mesh and are no longer visible. The proposed method does not allow us to easily model the separation of droplets of the viscous liquid from the mesh. If this effect is desired, some additional particles modelling this effect could be attached to the mesh. It is also possible to introduce particles to the viscous object. These can improve the visual quality and dynamism of the resulting animation sequence. Simplified particle based physical models can be applied to the implicit model to improve the default behaviour of the viscous object. A metaball representation of the particles is frequently used to integrate these particles into the implicit model. Particle positions retrieved after the physical simulation could be used to add metaballs to the final model. This would allow for partial modelling of physically correct behaviour within the existing geometric model.

Also proposed approximation for polygonal meshes can only provide good results for typical skeletal characters with axial symmetry. Other types of meshes may require additional efforts in order to achieve better approximation.

1.7 Conclusions

We have outlined a number of advantages of Function Representation (FRep) and demonstrated a number of applications suitable for computer animation and games. This representation has low memory requirements. Natural resolution independence of the original model allows us to adjust rendering quality according to available hardware specs. The discretization of the model can be performed in parallel, so that it is well suited for modern GPUs and CPUs with an ever increasing number of internal cores.

We believe that FReps have many more useful applications in the fields of computer animation and games.

1.8 CD contents

CD contains source code of the implementation of the proposed approach based on NVIDIA CUDA SDK release 2.1.

Bibliography

[Blinn 82] James F. Blinn. “A Generalization of Algebraic Surface Drawing.” *ACM Trans. Graph.* 1:3 (1982), 235–256.

[Bloomenthal and Shoemake 91] Jules Bloomenthal and Ken Shoemake. “Convolution surfaces.” *SIGGRAPH Comput. Graph.* 25:4 (1991), 251–256.

[Ebert et al. 02] David S. Ebert, Kenton F. Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing & Modeling: A Procedural Approach, Third Edition (The Morgan Kaufmann Series in Computer Graphics).* Morgan Kaufmann, 2002.

[Fryazinov et al. 08] O. Fryazinov, A. Pasko, and Adzhiev V. “An Exact Representation of Polygonal Objects by C^1-continuous Scalar

Fields Based on Binary Space Partitioning." Technical Report "TR-NCCA-2008-03", The National Centre for Computer Animation, Bournemouth University, UK, 2008.

[Geiss 07] Ryan Geiss. *GPU GEMS 3*, Chapter Generating Complex Procedural Terrains Using the GPU, pp. 7–38. Addison-Wesley Professional, 2007.

[Jin et al. 01] Xiaogang Jin, Chiew-Lan Tai, Jieging Feng, and Qunsheng Peng. "Convolution surfaces for line skeletons with polynomial weight distributions." *J. Graph. Tools* 6:3 (2001), 17–28.

[Kravtsov et al. 08] D. Kravtsov, O. Fryazinov, V. Adzhiev, A. Pasko, and P. Comninos. "Embedded Implicit Stand-ins for Animated Meshes: a Case of Hybrid Modelling." Technical Report "TR-NCCA-2008-01", The National Centre for Computer Animation, Bournemouth University, UK, 2008.

[Lorensen and Cline 87] William E. Lorensen and Harvey E. Cline. "Marching cubes: A high resolution 3D surface construction algorithm." In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, 21, 21, pp. 163–169. ACM Press, 1987.

[McCormack and Sherstyuk 98] Jon McCormack and Andrei Sherstyuk. "Creating and Rendering Convolution Surfaces." *Comput. Graph. Forum* 17:2 (1998), 113–120.

[NVIDIA 09] NVIDIA. "NVIDIA® Compute Unified Device Architecture (CUDA™). Introduction & Overview.", 2009.

[Pasko et al. 95] A. Pasko, V. Adzhiev, A. Sourin, and V. Savchenko. "Function Representation in Geometric Modeling: Concepts, Implementation and Applications." *The Visual Computer* :11 (1995), 429–446.

[Ricci 73] A. Ricci. "A Constructive Geometry for Computer Graphics." *The Computer Journal* 16 (1973), 157–160.

[Sengupta et al. 07] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. "Scan Primitives for GPU Computing." In *Graphics Hardware 2007*, pp. 97–106, 2007.

[Sengupta et al. 08] Shubhabrata Sengupta, Mark Harris, and Michael Garland. "Efficient Parallel Scan Algorithms for GPUs." Technical Report NVR-2008-003, NVIDIA Corporation, 2008.

[Tatarchuk et al. 07] Natalya Tatarchuk, Jeremy Shopf, and Christopher DeCoro. "Real-Time Isosurface Extraction Using the GPU Programmable Geometry Pipeline." In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, pp. 122–137, 2007.

[Uralsky 06] Y. Uralsky. "DX10: Practical Metaballs and Implicit Surfaces." Technical report, NVIDIA Corporation, 2006.