Real time rendering of skeletal implicit surfaces.

Olivier Rouiller
IMM-M.Sc.-2011-66

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

1 Preface

This report summarizes and presents my work during my master project at the Computer graphics and Image analysis at the Technical University of Denmark. This six-month work was focused on geometric modelling and rendering with implicit surfaces. My objective was to study recent methods for ray-tracing implicit surfaces on the GPU and to adapt them to render models modelled by their skeleton. The motivation was to use the simplicity that provide skeletal implicit surfaces to model surfaces without having to tessellate them as it is usually done. In order to be interesting, the method had to allow animation and texturing, thus being usable to render characters for example, and to be able to be used for real time applications.

The first part of this report presents the most relevant results of my work, it is presented on the form of a conference article. I propose here a practical framework to model and render skeletal implicit surfaces. I first review the bibliography that is relevant to this problem. I present the surface representation that I chose to create the models and the algorithm that I used to render the surfaces. I also present a method to animate the models using a hierarchical bone data structure and a method to allow texturing of the models with user control.

In Appendix A, I present some work that I did on using a bounding volume hierarchy to accelerate the ray-tracing algorithm. This work did not provide a significant advantage on the efficiency of the algorithm for the size of the models that I used. Hierarchical data structures are indeed not convenient to use on the GPU because traversing the BVH induces an overhead in branching and additional computation to test the ray against the bounding boxes of the node. However, as it is presented in the article, the algorithm suffers from a poor scalability and using a BVH could become an advantage for complex models or for a scene with several models. Therefore I provide in appendix a description of the algorithm and some implementation details.

2 Acknowledgements

I would like to thank my supervisor on this project Jakob Andreas Bærentzen for the original idea proposal and for the advises along the project. I would like to thank my family and friends for their support. I would like to thank the IMM staff and the staff from the Computer Graphics and Image Analysis department for making IMM an attractive environment to study and for the quality of the teaching proposed.

A practical framework for real time ray-tracing of animated skeletal implicit surfaces on the GPU.

Olivier Rouiller*
Department of Informatics and Mathematical Modelling, Danmarks Tekniske Universitet

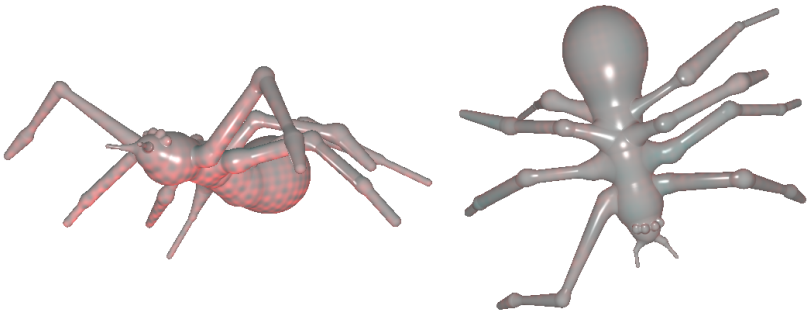


Figure 1: *A spider modelled and rendered with our framework.*

Abstract

We present a framework consisting of a surface representation based on skeletal implicit surfaces and a ray tracing algorithm that allows to model animated surfaces in real time. This framework allows common effects used in computer graphics such as texturing and displacement mapping and can be integrated in a real time renderer. To demonstrate the power of the framework, we build an interactive modelling tool.
The surface representation that we use is a subset of skeletal implicit surfaces and we limit ourselves to simple primitives such as points and line segments. We choose degree four polynomials for the convolution kernel. We combine this skeletal representation of the geometry with a hierarchical bone data structure to allow animation.
The rendering is done on the GPU in a single pass via ray-tracing. The ray-tracing algorithm uses bisection to find the ray-surface intersection and relies on finding points on the ray that are inside the surface. We propose a method to apply 3D and 2D textures to the model in the fragment shader that is compatible with animation. This method consists in transforming the position of the fragment to it's position in the model's rest pose to compute the texture coordinates with local projectors.

Keywords: Geometric Modelling, Implicit Surfaces, Real Time Ray-Tracing

*e-mail: s090842@student.dtu.dk

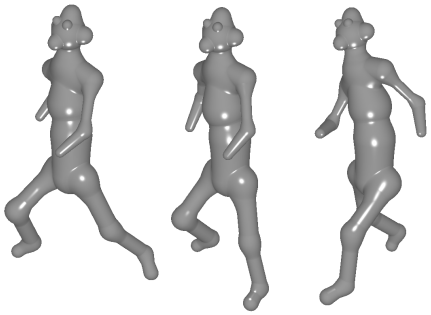


Figure 2: *A animated alien modelled with skeletal implicit surfaces and ray-traced on the GPU.*

1 Introduction

The usual real time rendering pipeline is based on polygon rasterization. Models are sent to the GPU as lists of indexed polygon as well as other geometric quantities such as normals, texture coordinates, tangent frames and bone transformations in the case of animated models. This way of representing and rendering the geometry is efficient and have proven its usability over years. However, modelling smooth surfaces requires a large number of polygons. Also, polygonal models use a significant memory storage space and sending the polygons to the GPU is a bottleneck on modern architectures.

Other surface representations such as surface patches and implicit surfaces can be used to represent surfaces in a more compact way. These surfaces also allow high level manipulations and can simplify the modelling process. Skeletal implicit surfaces are for instance only defined by their skeleton. Solutions to render these surfaces exist. The modern graphics pipeline allows to tessellate surface patches on the GPU, thus freeing the bandwidth between the CPU and the GPU. Implicit surfaces can be polygonised and rendered as triangle meshes.

Recently, the increasing power of graphics hardware allowed to ren-

der in real time implicit surfaces by ray-tracing on the GPU. This approach allows to benefit from the smoothness of these surfaces. The modelling power of skeletal implicit surfaces, their compactness and the possibility to ray-trace them in real time on the GPU make them good candidates for an alternative of polygon rendering.

We investigate the possibility to use skeletal implicit surfaces to design and implement a framework for modelling and real-time rendering of smooth surfaces. In order to be viable, the solution should allow animation and texturing and ideally provide performances comparable to polygonal rendering.

2 Related work

Implicit surfaces have been used in geometric modelling and procedural geometry modelling because they require few information to represent smooth surfaces of arbitrary topology.

An implicit surface S is defined as the zero level set of a scalar field F defined on the 3D space. $S = \{p \in R^3, F(p) = T\}$ where T has positive value.

We review some interesting works about modelling with implicit surfaces and about rendering implicit surfaces by ray-tracing. In the second section of this report we will present our method and how this previous work is interesting for our problem.

2.1 Geometric modelling with skeletal implicit surfaces

A skeletal implicit surface is an implicit surface whose potential field is defined by explicit geometric primitives such as points, line segments, polygons or other surfaces. Each of the geometric primitives are independent sources of potential field and the different fields are combined by mathematical operations. In the most simple case, the potential fields are summed. This produces smooth surfaces that blend together. In this case the potential field F is the sum of independent functions F_i, $F(x) = \sum_i F_i(x)$. Other operations such as difference, minimum or maximum produce more rich surfaces. These operators results in operation on the surface such as intersection, difference and union which are used in Constructive Solid Geometry (CSG).

2.1.1 Metaballs

The most simple kind of implicit surfaces used in computer graphics for modelling is metaballs. Metaballs were invented by Jim Blin [Blinn 1982], are now very popular in the computer graphics community and are often referred to as Blinn's blobs. Metaballs are implicit surfaces defined by point primitives. The surface of a single metaball is a sphere and when two metaballs are close enough, they blend into a single smooth surface.

The potential field F_i of a metaball i with center p_i and radius r_i is defined by a decreasing function f_i of the distance to the center of the ball, $F_i(x) = f_i(\|x - p_i\|)$. The original definition of Blinn's Blobs used an exponential function as the function f_i, $f_i(r) = e^{-ar_i}$ where a is a positive scalar.

In practise and for efficiency reasons, we use functions with a similar shape and with compact support. Usually polynomials of degree 4 or 6 are used. These functions are interesting because they are fast to compute. Also the function vanishes at a certain distance R of the centre of the ball.

Typically the function used are in the form $f(r) = \left(1 - \frac{r^2}{R^2}\right)^2$ if $r \leq R$, 0 otherwise. The radius R is referred to as effective radius of the metaball. The sphere centred at the center of the metaball and of radius R is called surface of influence of the metaball.

Metaballs have been mainly used in computer graphics to simulate and render fluids.

2.1.2 Skeletal implicit surfaces

Skeletal implicit surfaces are a more general concept than metaballs and are mathematically defined as convolution surfaces. Modelling with skeletal implicit surfaces have been introduced by Bloomenthal in his doctoral dissertation [Bloomenthal 1995].

A convolution surface is defined as an implicit surface with a potential field f defined with explicit primitives and convolution functions. The field F is of the form $f(x) = g(x) \times h(x) = \int_{R^3} g(r)h(x - r)\,\mathrm{d}r$.

g is defined by the explicit geometry. Typically, $g = 1$ on the geometry, 0 elsewhere. h is the convolution kernel, it is usually a distance function. With h and g defined in this way, and using point primitives, the integral boils down to the potential field of a metaball.

Convolution surfaces and their applications to computer graphics and geometric modelling have been thoroughly studied by Andrei Sherstyuk in his doctoral dissertation [Sherstyuk 1999].

2.1.3 The BlobTree

Although simple and powerful, skeletal implicit surfaces lack of local control and it is especially hard to create sharp edges using them.

Charles Wyvill addressed this issue by developing a data structure based on skeletal implicit surfaces [Wyvill et al. 1998] and expanding the operation done on the function field to allowing CSG operations. This data structure called the BlobTree as is constructed as a directed acyclic graph where the leaf nodes are geometric primitives and the non-leaf nodes are operation.

2.1.4 Applications to sketch based modelling

Finally, recent works on sketched based modelling and high level modelling showed the possibility to build powerful modelling tools based on the BlobTree and allow intuitive editing of the surface without concern about it's mathematical representation.

Sugihara et. all presented in [Schmidt et al. 2005] a system to edit BlobTree surfaces by sketches and more recently in [Sugihara et al. 2010] a system that allows to edit an implicit surface by drawing lines on it and by pushing and pulling them.

Although these works are beyond the scope of our project, they are worth mentioning to show the modelling power of implicit surfaces and to justify their study.

2.2 Ray tracing implicit surfaces on the GPU

Rendering implicit surfaces can be done by extracting a polygonal approximation of the surface and rendering the triangle mesh using the usual real time rendering pipeline. The most popular algorithm to polygonize an implicit surface is marching cubes [Lorensen and Cline 1987]. Marching cubes samples the implicit function on the grid, the cubes defined by the grid are then visited and polygons are created if there are sign alternations on the edges. This algorithm

can be optimised to run in real time thus allowing interactive editing and can also be implemented in the GPU [Geiss 2007] for real time rendering of procedural surfaces. However, the output surface requires a large number of polygons in order to produce smooth rendering and it is hard to produce a mesh with polygons that follows the features of the shape.

The other approach to render implicit surfaces is to ray-trace them. Implicit surfaces are suitable to be ray-traced because of their mathematical representation. Finding a intersection of a ray with an implicit surface boils down to finding the zeros of the field function along the ray.

With algebraic surfaces, it is possible to compute exactly the intersection by finding the root of an univariate polynomial.

Another approach is to use iterative methods to approximate the root of the function.

2.2.1 Root finding techniques

Fukuyama presented in [Fukuyama et al. 1994] a method to display metaballs by using Bézier tetrahedra. This method consists in computing the roots of the field function along the ray by solving an univariate polynomial equation.

In [Loop and Blinn 2006], Loop et al. presented a method to ray-trace arbitrary algebraic surfaces on the GPU. With this method it is possible to ray-trace accurately implicit surfaces on the GPU. The method is also based on finding an analytic solution of the polynomial.

More recently [Kanamori et al. 2008], this technique was adapted for metaballs to ray-cast a large number of metaballs at interactive frame rate. The algorithm also use Bézier clipping as well but evaluates only the metaballs contributing for a pixel by using depth peeling.

2.2.2 Ray marching and interval arithmetic techniques

Another way to ray-trace implicit surfaces is to use ray marching techniques. These techniques consist in finding the ray-surface intersection by evaluating the field function along the ray.

In [Mitchell 1990], Mitchell proposed algorithms of interval arithmetic such as bisection to compute efficiently the ray surface intersection with implicit surfaces.

2.2.3 Sphere tracing

Sphere tracing is an adaptive step length search and is known to be efficient to ray-trace surfaces to which we have an evaluation of the distance of a point in space to the surface. This technique was introduced by JC Harts in [Hart 1996].

This technique requires to have a signed distance field of the objects that are ray-traced. The value of the signed distance field at a point in space is the distance from this point to the closest point on the ray-traced surfaces.

It is possible to generate a distance field for many geometric object and to compose scenes with the SDFs. Recently in [Reiner et al. 2011], this method was for interactive modelling.

Sphere tracing usually yields better performances for ray marching implicit surfaces but with metaballs and convolution surfaces in general, it is hard to compute the distance field and it requires to evaluate the derivative of the field function which is an expensive computations when the number of primitives is high.

2.3 Spatial data structures for real time rendering on the GPU

For the last few years, ray tracing on the GPU have been a subject of interest in the computer graphics community. The parallel architecture of the GPU has been used to accelerate off-line renderers and real time ray tracing became possible with the increasing power of the hardware.

In the mean time, acceleration data structures such as bounding volume hierarchies and kd-trees have been adapted and improved to fit the need of a GPU ray-tracer.

In [Stich et al. 2009], an algorithm is presented to build bounding volume hierarchies for animated scenes ray-traced on the GPU. The BVH is built from top to bottom at every frame using spatial splits.

Spatial splits allow to use axis aligned bounding boxes that tightly fit the geometric primitives and the resulting boxes do not overlap.

This technique was adapted for metablalls in [Gourmel et al. 2010] to ray-trace thousands of metaballs at interactive frame rate.

3 Surface Representation

With our model, the field function F of the implicit surface is defined as a sum of polynomials f_i of the distance to point and segment primitives P_i.

$$F(x) = \sum_{P_i} f_i(dist(x, P_i)),$$

where $dist(x, P_i)$ is the distance of x to the primitive P_i.

For the functions f_i, we chose degree 4 polynomials limited to a radius R_i. These functions have the advantage to be fast to compute and to have a compact support, which allows us to discard the primitives that don't have an influence when ray tracing the surface.

$$f_i(r) = \left(1 - \frac{r^2}{R_i^2}\right)^2$$

Figure 3 illustrates our surface representation. The dashed lines represent the surfaces of influence of the primitives, the red solid curve represents the surface. For more variety of surfaces, we allow the segment to have different effective radius at the extremities, the effective radius is interpolated on the segment between the two values.

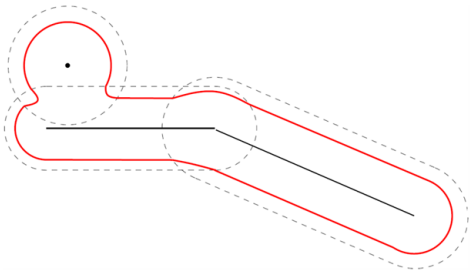
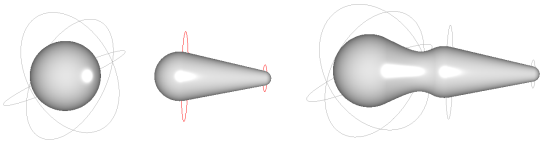


Figure 3: *Illustration of the skeletal surface representation.*

Figure 4a shows the basic primitives of our model, a metaball and a metatube with two different radius. The circles show the effective radius of the primitives. Figure 4b shows the two primitives blended together when their surfaces of influence intersect.



(a) (b)

Figure 4: *Basic shapes of our surface representation.*

3.1 Advantages of the representation

This surface representation allows to model easily complex models. The models shown on this paper were created in less than an hour with our modelling tool. It is particularly suitable to create smooth surfaces without the need to use complex frameworks and algorithms such as surface subdivisions or surface patches. Also this representation is very compact, the gecko presented on figure 5 is composed of only 66 metaballs whereas the polygonised model requires over 5000 vertices and more than 10000 triangles to present a comparable smoothness. This compactness is an advantage for both storage in memory and to save the bandwidth with the GPU for rendering.

Finally, the data structures used to build and maintain the models are very simple and map naturally to the GPU memory, a plain array of metaballs is sufficient to store the model.

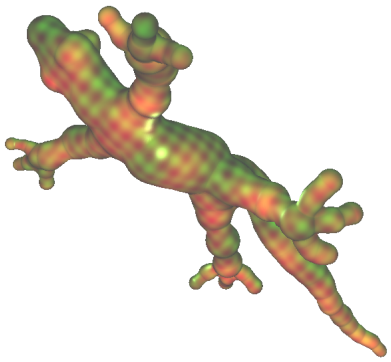


Figure 5: *A Gecko modelled of 66 metaballs.*

4 Ray-tracing Algorithm

With such a surface representation, it is easy to implement a ray tracer on the GPU. Positions of the point primitives and of the extremities of the line segment are sent to the fragment shader as well as effective radius. On the fragment shader, we compute a ray in world space.

When the ray is computed, several ray marching techniques and optimizations are possible to find the ray surface intersections.

When the intersection is found, the normal of the surface is computed by differentiating the field function, shading is applied and effects such as texturing, normal mapping or advanced lightings effects using other rays can be applied.

4.1 Ray marching

The easiest way to find the ray surface intersection is to perform ray marching. A naive approach is to use a constant step length and to stop the search as soon as we find a sign alternation of the function. The code for this method are presented in listing 1. This approach is not fast enough to achieve interactive frame rates with complex models since it requires an important number of function evaluations.

Algorithm 1 Ray-surface intersection with naive ray marching

```
// Construct ray from eye to fragment
vec3 rayDir = normalize(worldPosition-cameraPos);
vec3 ray = worldPosition;

int steps = 0;
float value = fieldFunction(ray);

while( value > 0 && steps < maxSteps ){
 ray += stepLength*rayDir;
 steps++;

 value = fieldFunction(ray);
}
```

To reduce the number of evaluations, we implemented an interval arithmetic search algorithm, we find a point outside the surface, one inside and we find the intersection by bisection. This method is similar to the method used in [Gourmel et al. 2010]. They use the BVH to reduce the number of metaballs evaluated. We instead discard primitives by testing their surface of influence for intersection against the ray.

4.2 Bisection

A bisection can be used to reduce the number of steps needed to find a zero of the field function along the ray. This technique relies on finding a point inside the surface after the first intersection of the ray with the surface and a point outside the surface.

These two points define an interval along the ray that can be iteratively subdivided until a certain precision is reached. The code for this algorithm is presented in listing 2.

Algorithm 2 Ray-surface intersection with bisection

```
vec3 hi; // A point inside the surface
vec3 low; // A point outside the surface

vec3 mid = 0.5(hi+low);
int steps = 0;
float vmid = fieldFunction(mid);

while(steps < maxSteps && abs(vmid) > eps)
{
 steps++;
 mid = 0.5*(low+hi);
 float vmid = fieldFunction(mid);

 if(vmid < 0)
     low = mid;
 else
     hi = mid;
}
vec3 intersection = mid;
```

To find the first interval for the bisection, one can do a constant step ray marching, stop it when a point with a positive value is found and use the two last iterates to initialize the bisection interval.

However we have seen that such a method is expensive in number of evaluations. Also this method does not give any guaranty that geometric details are not missed.

We use the information that we have on the geometry of the sources of the field to find points inside the surface efficiently.

We first discard the primitives whose effective surface does not intersect the ray. For all the primitives that pass this test, we compute a point that we know is likely to be the candidate for a point inside the surface. Finally, we choose among these points the one that is the closest to the eye position.

4.2.1 Finding points inside the surface

To find points that are likely to be inside the surface, we compute for each metaball whose effective surface is intersected along the ray the projection of the center on the ray.

For tube primitives, we have to compute more than one candidate. We compute the intersection of the ray with the plane containing the segment and aligned to the viewer as well as the projections of the segment's extremities with the ray. We keep from these three points the one that is the closest to the line segment.

Figure 6 shows an illustration of how the interval for bisection is computed with metaballs only. The method is the same with line segment primitives. On Figure 6a, we show in red the metaballs that are discarded because their sphere of influence do not intersect the ray. The blue metaball is discarded because the value of the field function at the projection of the center on the ray is negative. Only the remaining metaballs are used for future evaluations of the field function and the projections of the centers are stored. On Figure 6b, the closest of these points is selected as well as the first intersection of the ray with the bounding box of the surface to initialize the bisection.

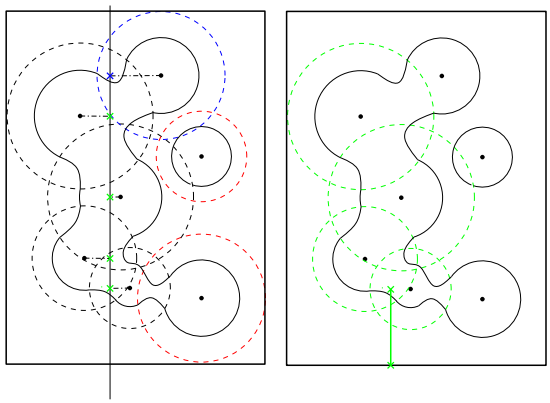


Figure 6: *Illustration of the ray-surface intersection algorithm.*

4.2.2 Limitations

With this technique we can track the surface efficiently but some ray-intersection are missed. This is the case when two primitives are just close enough to start blending. Figure 7 illustrates an example of a ray that miss the surface.

Our method to find a point inside the surface with a line segment primitive also causes artefacts when the direction oh the view is close to being parallel to the segment's axis.

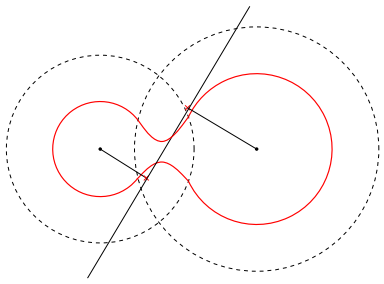


Figure 7: *Example of a ray that miss the surface with our intersection method.*

4.3 Evaluation of the method

Our algorithm allows to render our skeletal implicit surfaces at interactive frame rate. The root finding by bisection is fast and easy to implement. The surface is found in less than 10 bisection steps for a precision that doesn't present visible artefacts. However, selecting the primitives that have an influence on the ray requires to evaluate the entire function for each primitive which leads to a n^2 complexity. This does not scale well for large models. A solution would be to use a bounding volume hierarchy as presented in [Gourmel et al. 2010]. However, constructing such a BVH at every frame is expensive and traversing the hierarchy on the GPU has a non negligible cost.

Ray-tracing the surface remains very slow in comparison to rendering the tessellated mesh, our models are rendered at 20 to 100 fps depending on the complexity of the models against around 2000 fps for the tessellated meshes on an NVidia Quatro FX 5800.

It is also worth mentioning that the algorithm is very sensitive to the area on the screen covered by the surface since most of the rendering computations are done on the fragment shader. This is interesting because the cost of rendering the model is resolution dependant rather than geometry dependant. To provide a more scalable level of detail technique, it could be interesting to discard primitives that are too small to produce visible details.

5 Rigging and Animation

To animate the surface, we simply update the positions of the primitives in the CPU. This can be done easily by attaching them to a hierarchical skeleton data structure.

We investigated two different work-flows to produce rigged and animated surfaces. The first approach is very similar to the rigging process used with polygonal models. It consists in modelling the surface with metaballs and metatubes in a first time. Then the modeller creates a rig for the surface and finally the primitives are attached to the rig.

This work-flow with our modelling tool is illustrated in figure 8. From the left to the right : the model is sculpted with metaballs, a rig is positioned on the model, primitives are selected and attached to a specific bone, the surface is ready to be animated and we can create poses by orienting the bones.

Since we use line segment primitives in our surface representation, the segments are natural candidates to provide the bone structure.

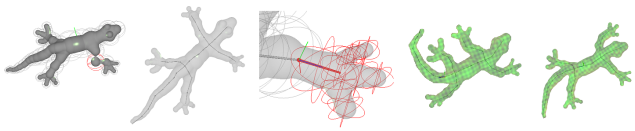


Figure 8: *Work flow with metaball sculpting.*

With this in mind, we developed another work-flow. We start by creating the armature of the model but the bones are used as primitives for the implicit surface. Then, the radius of these primitives are adjusted to give the right silhouette to the model. Finally we can add details with metaballs and attach them to the bones.

Figure 9 illustrates how we model a dinosaur with this work-flow. From the left to the right : the skeleton is created with tube primitives attached to the bones, the surface is edited by modifying the radius of the metatubes and by adding a few metaballs for the head and the feet and the surface is ready to be animated.

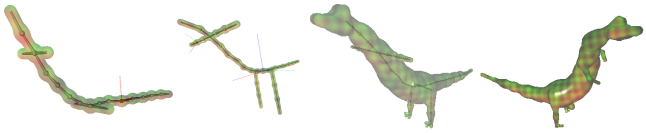


Figure 9: *Work flow with skeletal implicit surfaces.*

6 Texturing

Texturing implicit surfaces is not straightforward because we don't have access directly to points on the surface so we cannot attach texture coordinates as we would do with a triangle mesh.

A common approach to tackle this problem is to use hyper textures or 3D textures to cover the space in which the surface is embedded and simply looking up the texture with the fragment's position.

Another approach is to use texture projectors to generate the texture coordinated for a 2D texture mapping. For example, an implicit surface generated by a line segment can be intuitively textured using a cylindrical projection. When we draw a fragment on this surface, we compute its cylindrical coordinates in the referential of the line segment and use these as texture coordinates.

Figure 10 shows a simple tube surface textured with a cylindrical projection.

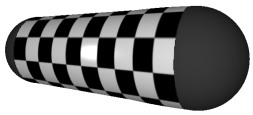


Figure 10: *A simple tube surface texture mapped with a cylindrical projection.*

It is possible to use other kind of projections such as spherical or planar.

Wyvill et al. proposed a more advanced scheme in [Tigges and Wyvill 1998]to texture skeletal implicit surfaces by shooting particles from the surface to a well parametrized support surface using the gradient of the field. This method allow less distortion in the mapping than simple projections but problems still occurs at the intersection of primitives and computing the trajectories of the particles for each fragment is prohibitive for a real time ray-tracer.

6.1 Surface skinning

Using local texture projections and computing the texture coordinates on the fragment shader allows to apply texture to our surfaces with user control.

However, with animation, having a coherent texture mapping requires to animate the projectors as well as the model. Also, having a projector for each primitive is not adapted for regions where several primitives have an influence. One could blend the different textures but this gives poor results.

To address this issue, we propose a method that is inspired by vertex skinning.

The method consists in texturing the model in a rest position using local texture projectors. The designer can use an arbitrary number of projectors with different projections.

Then, when rendering the animated model, we assign bone weights to the fragment depending on it's distance to the bones and we use them to transform the fragment's position to the rest pose space. Then we compute the texture coordinates of the fragment by using the local projectors and look up the texture accordingly.

Figure 11 shows an example of how the textures are applied. The model is composed of two segment primitives associated to bones and textured by a single cylindrical projector. On the left, the colours show the weights of the fragments associated with the two bones. On the right, we see that the texture mapping follows the deformation of the surface.

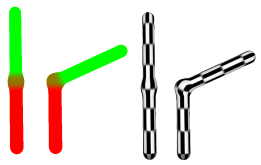


Figure 11: *Illustration of the skinning scheme used for texturing.*

Listing 3 shows the pseudo code for this technique.

This method showed good results and is adaptable for 3D textures as shown in figure 12. To implement this technique, we send to the GPU the transformation matrices of the bone in rest position as well as the inverse of the bone's current transformation matrices. The computation to find the position of the fragment in rest position are negligible in comparison to the ray tracing of the surface. We implemented this method as a proof of concept but giving the control on the radius of influence of the bones can lead to more robust implementations. Also, using this technique as well as using local texture projectors implies that we have to search for the bones and projectors that have an influence on a point of the surface.

6.2 Displacement mapping

To enhance details on the surface, techniques such as normal mapping and displacement mapping are commonly used in computer graphics. To apply these effects with a polygonal representation, we need to provide a texture mapping and the tangent vector to the surface.

Algorithm 3 Transformation of the vertex positions for texturing

```
vec3 pointOnSurface;
vec3 restPosition = vec3(0,0,0);

for(int bone = 0; bone <  nbBones; bone++)
{
 // compute fragment weight wtrt. the bone
 float w = weight(bone, pointOnSurface);
 // transform
 vec3 transformed = pointOnSurface;
 // Back to object space
 transformed += boneInvTrans[bone][3].xyz;
 transformed = boneInvTrans[bone]*transformed;
 // to rest world space
 transformed = boneRestTrans[bone]*transformed;
 transformed.xyz += boneRestTrans[bone][3].xyz;
 // sum
 restPosition += w*transformed.xyz;;
}

// Compute the texture coordinates of the fragment in rest position
vec2 uv = texCoord(restPosition);
```

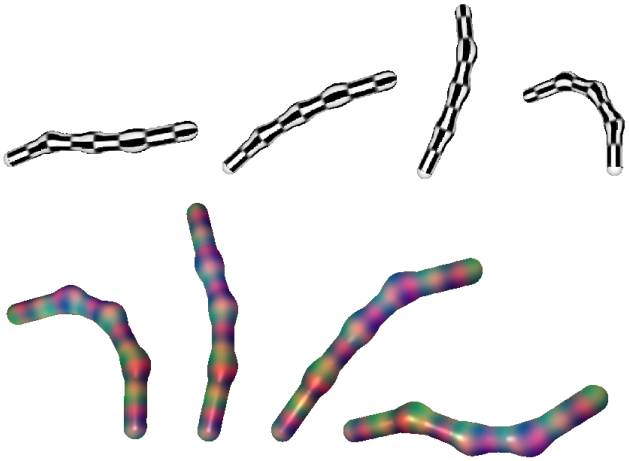


Figure 12: *Illustration of the skinning scheme used for texturing.*

With implicit surfaces, we can add a value to the field function to add relief details. Also, because the normal is computed from the gradient of the field function, the perturbation is applied automatically to the surface normal.

In this way, if the perturbation is small enough to keep the function null outside the support of the original function, we can apply the displacement while the ray-surface intersection is searched.

This method applies very easily with hyper-textures as shown on figure 13. It can also be applied with projected 2D as shown on figure 14 but the uv coordinates have to be computed at each step of the intersection search.

This method is not robust enough when using our optimization with bisection because the point inside the surface computed as described in section 4.2 is no longer guaranteed to be inside the surface. Also with sharp derails there might be more than one intersection on the ray between our initial point and the outside.

For animated surfaces the method cannot be applied either.

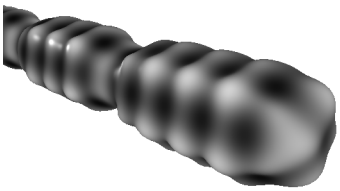


Figure 13: *Displacement mapping with a 3D noise function.*

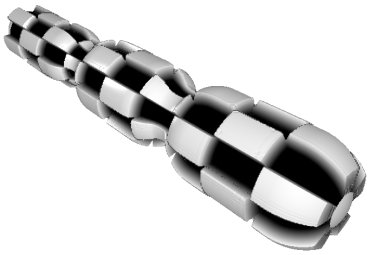


Figure 14: *Displacement mapping with a projected 2D texture.*

7 Results and discussions

We revisited geometric modelling with implicit surfaces. We used a simple surface representation with skeletal implicit surfaces that allowed to model quickly interesting models. We implemented a real time renderer based on ray-tracing adapted for our surface representation. This method allows to render at interactive frame rate animated models. Although this method is far from being as efficient as polygon rendering, it has some advantages such as it's simplicity, the compactness of the models representation and a very small amount of memory to send to the GPU to render smooth surfaces.

We investigated the possibilities to allow texturing with user control and developed a method to keep a coherent texture mapping during animation.

We showed the possibilities to integrate displacement mapping in the rendering but this method is limited to static models and can't benefits of the speed up provided by the bisection algorithm.

In conclusion, using implicit surfaces for the surface representation and ray-tracing for the rendering seems to be an alternative to polygon rendering that provides simplicity as well as flexibility. The main issue that remains is efficiency. Using the current graphics architecture and our method, using implicit surfaces to render models is prohibited in commercial applications where the scene includes a great number of models.

However, it is likely that better ray-tracing algorithm and future advances of the graphics hardware can provide better performances with the same advantages. We have mentioned for example sphere tracing, Bézier clipping and the use of bounding volume hierarchies.

In future works, it would be interesting to investigate the possibility to implement a real time ray-tracer for a wider range of skeletal surfaces. For example, it would be interesting to use convolution surfaces with primitives such as parametric curves. It would also be interesting to implement a real time ray-tracer for the Blob-Tree,

thus providing a renderer for sketch-based modelling tools.

References

BLINN, J. F. 1982. A Generalization of Algebraic Surface Drawing. *ACM Trans. Graph. 1*, 3 (July), 235–256.

BLOOMENTHAL, J. 1995. *Skeletal design of natural forms*. PhD thesis, Calgary, Alta., Canada, Canada. UMI Order No. GAXNN-03088.

FUKUYAMA, T. N., NISHITA, T., AND NAKAMAE, E. 1994. A method for displaying metaballs by using bezier clipping. *Computer Graphics Forum 13*, 271–280.

GEISS, R. 2007. Generating complex procedural terrains using the gpu. *GPU Gems 3 3*.

GOURMEL, O., PAJOT, A., PAULIN, M., BARTHE, L., AND POULIN, P. 2010. Fitted bvh for fast raytracing of metaballs. *Computer Graphics Forum 29*, 2 (may), 281–288.

HART, J. C. 1996. Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer 12*, 527–545. 10.1007/s003710050084.

KANAMORI, Y., SZEGO, Z., AND NISHITA, T. 2008. GPU-based fast ray casting for a large number of metaballs. *Computer Graphics Forum (Proc. of Eurographics 2008) 27*, 3, 351–360.

LOOP, C., AND BLINN, J. 2006. Real-time gpu rendering of piecewise algebraic surfaces. In *ACM SIGGRAPH 2006 Papers*, ACM, New York, NY, USA, SIGGRAPH '06, 664–670.

LORENSEN, W. E., AND CLINE, H. E. 1987. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph. 21* (August), 163–169.

MITCHELL, D. P. 1990. Robust ray intersection with interval arithmetic. In *Proceedings on Graphics interface '90*, Canadian Information Processing Society, Toronto, Ont., Canada, Canada, 68–74.

REINER, T., MCKL, G., AND DACHSBACHER, C. 2011. Interactive modeling of implicit surfaces using a direct visualization approach with signed distance functions. *Computers Graphics 35*, 3, 596 – 603. Shape Modeling International (SMI) Conference 2011.

SCHMIDT, R., WYVILL, B., SOUSA, M. C., AND JORGE, J. A., 2005. Shapeshop: Sketch-based solid modeling with blobtrees.

SHERSTYUK, A. 1999. *Convolution Surfaces in Computer Graphics*. PhD thesis, Monash, Monash, Australia, Australia.

STICH, M., FRIEDRICH, H., AND DIETRICH, A. 2009. Spatial splits in bounding volume hierarchies. In *Proc. High-Performance Graphics 2009*.

SUGIHARA, M., WYVILL, B., AND SCHMIDT, R. 2010. WarpCurves: A tool for explicit manipulation of implicit surfaces. *Computers & Graphics 34*, 3, 282–291. Shape Modeling International (SMI) 2010.

TIGGES, M., AND WYVILL, B. 1998. Texture mapping the blobtree. In *In Proceedings of the Third International Workshop on Implicit Surfaces*, 123–130.

WYVILL, B., GALIN, E., AND GUY, A. 1998. The Blob Tree, Warping, Blending and Boolean Operations in an Implicit Surface Modeling System. *Implicit Surfaces 3* (June). Chosen for inclusion in a special issue of Computer Graphics Forum.

A Towards a better scalability of the ray-tracing algorithm.

The algorithm to find the ray surface intersections that we presented is relatively fast but suffers from a poor scalability. The first reason is that when discarding the non-contributing primitives, we evaluate the entire function for each test. Also, when we search for intersection by bisection, we evaluate the potential of all the primitives whose surface of influence intersect the ray, some of which might not have an influence at the ray-surface intersection. To address this issue, Gourmel et al. use a fitted BVH [1]. Using a bounding volume hierarchy allows to reduce the number of primitives tested when we compute the first interval for the bisection.

I tried to adapt this algorithm for my surface representation but I was not able to achieve better performances with it. However, this technique, carefully implemented and optimised could allow to increase the scalability of the algorithm.

A.1 Building the BVH

As presented in [1], [2] and [3], we build the BVH at every frame to allow animation. The BVH is built like a kd-tree so that the resulting nodes do not overlap. The leaf nodes contain references to all the primitives whose surface of influence intersect the AABB of the node. Primitives may be referenced by several nodes.

A.1.1 Overview of the algorithm.

The BVH that we build is a BVH where the leaf nodes contain references to the primitives that have an influence inside their axis aligned bounding box. The BVH is built from top to bottom and at each stage, we split the AABB of the node and distribute to the two children its primitives. Finding the splitting plane should be done using a Surface Area Heuristic (SAH). The algorithm evaluates the cost of traversing the BVH during ray-tracing of several possible splits and choose the one that has the smallest heuristic cost.

When splitting the node, primitives are distributed to the left child or to the right child according to their position with respect to the splitting plane. Then, the primitives of the right child are tested for intersection with the primitives of the left child. The primitives that do intersect are added to the left child as split primitives. The same operation is done for the primitives of the left child.

Finally, the AABBs of the children are computed as the AABBs of all the primitives that they contain intersected with the half AABB of the parent node. This way, the nodes do not overlap, are tightly enclosing the primitives that have an influence inside their AABB and contain references to these primitives.

A.1.2 Memory layout

The BVH has to be transferred to the GPU for ray traversal. As suggested in [**?**], we transfer it as texture memory. To fit the memory layout of the texture memory,

the BVH is encoded in a plain array where the children of a node i are located at indices $2i + 1$ and $2i + 2$.

We choose to encode the BVH in a 4 component floating point texture. We store a node in two pixels, one for each corner of the AABB. The w component of the pixels are used to store a flag, it's value is -1 if the node is not a leaf, otherwise, it is an index to the list of the primitives contained in the node.

We also write after the tree in the texture the lists of primitives referenced by the nodes. We allocated two pixels to store the indices to the metaballs and two pixels to store the indices to the metatubes. This way a node can contain at most 8 metaballs and 8 metatubes. It would be preferable to allow nodes to have an arbitrary number of references to primitives and to store them in a separate texture of one component texels so that the list of primitive can be read as a single array and does need to load separate 4 component vectors.

We stored the primitives as uniform variables. This is convenient since uniform memory is faster to access than texture memory. However, the uniform memory is bounded and in our implementation it was a limitation. It would be preferable to store them in texture memory instead.

A.2 Traversing the BVH

The BVH is traversed from top to bottom and from front to back, when we reach a leaf node, we try to find a point on the ray and inside the surface as described in the article. If no such point is found, we backtrack and visit other node farther on the ray. This algorithm requires to maintain a stack where we push the indices of the nodes that should be visited. Listing 1 shows the code used to traverse the BVH on the GPU. It returns the node where the ray intersects the surface if any. It also loads in global variables the primitives that have an influence inside the node and caches the point inside the surface used to initialize the bisection.

References

[1] Olivier Gourmel, Anthony Pajot, Mathias Paulin, Loic Barthe, and Pierre Poulin. Fitted bvh for fast raytracing of metaballs. *Computer Graphics Forum*, 29(2):281–288, may 2010.

[2] Martin Stich, Heiko Friedrich, and Andreas Dietrich. Spatial splits in bounding volume hierarchies. In *Proc. High-Performance Graphics 2009*, 2009.

[3] Ingo Wald. On fast construction of sah-based bounding volume hierarchies. In *In Proceedings of the 2007 IEEE/EG Symposium on Interactive Ray Tracing. IEEE*, pages 33–40, 2007.

Algorithm 1 Traversing the BVH

```
int RayIntestectsBVH(vec3 origine, vec3 dir){
 vec4 bottomC, topC;
 top = 0;
 int node = 0, c1 =0, c2 = 0;

 // push the BVH top node on the stack
 push(node);
 while(top > 0){
     // First non expanded node
     node = pop();
     // Load the AABB of the node
     bottomC = nodeBottom(node);
     topC = nodeTop(node);

     // If this is a leaf check for intersection with the surface
     if(bottomC.w != -1.0 || topC.w != -1.0){
         // This test loads the primitives if the node and tries to find a point inside the surface.

         // The test is positive if such a point is found.
         if(rayIntersectsSurfaceInNode(origine, dir, node)) return node;

     }else{
         // Otherwise expand
         c1 = child1(node);
         c2 = child2(node);
         // Distances from the eye to the AABBS intersection
         float d1,d2;

         // Test c1 for intersection
         bottomC = nodeBottom(c1); topC = nodeTop(c1);
         bool c1Intersects = RayIntestectsAABB(origine, dir, bottomC.xyz, topC.xyz, d1);

         // Test c2
         bottomC = nodeBottom(c2); topC = nodeTop(c2);
         bool c2Intersects = RayIntestectsAABB(origine, dir, bottomC.xyz, topC.xyz, d2);


         // Push the children if they intersect by order of distance to the eye
         if( c1Intersects ){
             if( c2Intersects ){
                 if( d1 <= d2 ){
                     push(c2); push(c1);
                 }else{
                     push(c1); push(c2);
                 }
             }else
                 push(c1);
         }else if( c2Intersects )
             push(c2);
     }
 }
 return -1;
}
```