Ray-casting Algebraic Surfaces using the Frustum Form

Martin Reimers    Johan Seland*

Eurographics 2008
Crete, Thursday April 17.
Algebraic Surfaces

- Zero set of polynomial $f : \mathbb{R}^3 \rightarrow \mathbb{R}$

\[ f(x, y, z) = \sum_{0 \leq i+j+k \leq d} f_{ijk}x^i y^j z^k = 0 \]

- Sphere: $x^2 + y^2 + z^2 - 1 = 0$
- Classic field of mathematics
- Very compact representation
- Renewed interest as geom. rep.
Ray-casting Overview

- Given a view frustum
- and screen resolution 
  \((m+1) \times (n+1)\)
- Let each pixel correspond to a ray \(r_{pq} : [0, 1] \rightarrow \mathbb{R}^3\)
- Find first intersection i.e.
  \[ f(r_{pq}(t)) = 0 \]
- Find final pixel color
- Little brother of ray-tracing
Ray-casting Overview

- Given a view frustum
- and screen resolution \((m + 1) \times (n + 1)\)
- Let each pixel correspond to a ray \(r_{pq} : [0, 1] \rightarrow \mathbb{R}^3\)
- Find first intersection i.e. 
\[
f(r_{pq}(t)) = 0
\]
- Find final pixel color
- Little brother of ray-tracing
Ray-casting Overview

- Given a view frustum
- and screen resolution \((m + 1) \times (n + 1)\)
- Let each pixel correspond to a ray \(r_{pq} : [0, 1] \rightarrow \mathbb{R}^3\)
- Find first intersection i.e. \(f(r_{pq}(t)) = 0\)
- Find final pixel color
- Little brother of ray-tracing
Ray-casting: Challenges

Conceptually simple - computationally challenging

- Working directly on $f$ is expensive
- Better approach: use univariate representation for each ray
- How to find the ray equations (in Bernstein form)

$$ f(r_{pq}(t)) = \sum c_{pqr} B^d_r(t) = 0 $$

- How to find the roots?
- Embarrassingly parallel - use GPU?
Outline

Ray coefficient computation

Root finding

Implementation details

Results and Conclusion
Parameterizing the View Frustum

Idea: Parameterize the view frustum over the unit cube by

\[ L(u, v, w) = \sum_{i,j,k=0}^{1,1,1} v_{ijk} B_i^1(u) B_j^1(v) B_k^1(w) \]
Parameterizing the View Frustum

Idea: *Parameterize the view frustum over the unit cube by*

\[
L(u, v, w) = \sum_{i,j,k=0}^{1,1,1} v_{ijk} B_i^1(u) B_j^1(v) B_k^1(w)
\]

\[\blacktriangleright\] \((u, v, 0)\) is mapped to the front plane
Parameterizing the View Frustum

Idea: *Parameterize the view frustum over the unit cube by*

\[ L(u, v, w) = \sum_{i,j,k=0}^{1,1,1} v_{ijk} B_i^1(u) B_j^1(v) B_k^1(w) \]

- \((u, v, 0)\) is mapped to the front plane
- \((u, v, 1)\) is mapped to the back plane
Parameterizing the View Frustum

Idea: *Parameterize the view frustum over the unit cube by*

\[ L(u, v, w) = \sum_{i,j,k=0}^{1,1,1} v_{ijk} B_i^1(u) B_j^1(v) B_k^1(w) \]

- \((u, v, 0)\) is mapped to the front plane
- \((u, v, 1)\) is mapped to the back plane
- Rays are mapped to rays (but no longer parallel)
The Frustum Form (FF)

We call $g = f \circ L : [0, 1]^3 \rightarrow \mathbb{R}$ the Frustum Form.
The Frustum Form (FF)

We call $g = f \circ L : [0, 1]^3 \rightarrow \mathbb{R}$ the Frustum Form

$g = 0$ corresponds to $f = 0$
The Frustum Form (FF)

We call $g = f \circ L : [0, 1]^3 \to \mathbb{R}$ the Frustum Form

- $g = 0$ corresponds to $f = 0$
- $g$ is a tri-degree $d$ polynomial
The Frustum Form (FF)

We call $g = f \circ L : [0, 1]^3 \rightarrow \mathbb{R}$ the Frustum Form

$g = 0$ corresponds to $f = 0$

$g$ is a tri-degree $d$ polynomial

Bernstein form $g(u, v, w) = \sum_{i,j,k=0}^{d,d,d} g_{ijk} B_i^d(u) B_j^d(v) B_k^d(w)$
The Frustum Form (FF)

We call \( g = f \circ L : [0, 1]^3 \rightarrow \mathbb{R} \) the Frustum Form

- \( g = 0 \) corresponds to \( f = 0 \)
- \( g \) is a tri-degree \( d \) polynomial
- Bernstein form \( g(u, v, w) = \sum_{i,j,k=0}^{d,d,d} g_{ijk} B_i^d(u) B_j^d(v) B_k^d(w) \)
- Rays are iso-parametric (parallel), ray polynomials of degree \( d \)
Ray coefficients

For pixel \((p, q)\) this yields

\[
g\left(\frac{p}{m}, \frac{q}{n}, w\right) = \sum_{i,j,k=0}^{d,d,d} g_{ijk} B_i^d\left(\frac{p}{m}\right) B_j^d\left(\frac{q}{n}\right) B_k^d(w)
\]
Ray coefficients

For pixel \((p, q)\) this yields

\[
g(p/m, q/n, w) = \sum_{i,j,k=0}^{d,d,d} g_{ijk} B_i^d(p/m) B_j^d(q/n) B_k^d(w)
\]

where \(c_{pqk}\) are the ray coefficients.
Ray coefficients

For pixel \((p, q)\) this yields

\[
g(p/m, q/n, w) = \sum_{i,j,k=0}^{d,d,d} g_{ijk} B_i^d(p/m) B_j^d(q/n) B_k^d(w)
\]

\[
f(r_{pq}(t)) = \sum_{k=0}^{d} c_{pqk} B_k^d(t)
\]

\(c_{pqk}\) are the ray coefficients
Outline

Ray coefficient computation

Root finding

Implementation details

Results and Conclusion
Bernstein Polynomials

Ray equations in Bernstein form

- \( f(\mathbf{r}_{pq}(t)) = \sum_{k=0}^{d} c_{pqk} B_{k}^{d}(t) \)
- \( B_{k}^{d}(t) = \binom{d}{k} t^{k} (1 - t)^{d-k} \)
- Numerically optimal [Farouki]
- Coefficients form control polygon

Properties of Control Polygon

- The function is approximated by the control polygon
- Variation Diminishing Property
  - \# of zeros in polynomial \( \leq \# \) of zeros in C. P.
- Control Polygon can be subdivided/refined
**Bernstein Polynomials**

Ray equations in Bernstein form

- $f(r_{pq}(t)) = \sum_{k=0}^{d} c_{pqk} B_{k}^{d}(t)$
- $B_{k}^{d}(t) = \binom{d}{k} t^k (1 - t)^{d-k}$
- Numerically optimal [Farouki]
- Coefficients form control polygon

**Properties of Control Polygon**

- The function is approximated by the C. P.
- Variation Diminishing Property
  - # of zeros in polynomial $\leq$ # of zeros in C. P.
- Control Polygon can be subdivided/refined
Numerical Root Finding

Algorithms: Repeated refinement of control polygon, either
- Bézier subdivision [Lane & Riesenfeld’81, Rockwood et al’89]
- B-Spline knot insertion [Mørken & Reimers’07]
Numerical Root Finding

Algorithms: Repeated refinement of control polygon, either
- Bézier subdivision [Lane & Riesenfeld’81, Rockwood et al’89]
- B-Spline knot insertion [Mørken & Reimers’07]
Numerical Root Finding

Algorithms: Repeated refinement of control polygon, either

- Bézier subdivision [Lane&Riesenfeld’81, Rockwood et Al’89]
- B-Spline knot insertion [Mørken&Reimers’07]
Numerical Root Finding

Algorithms: Repeated refinement of control polygon, either

- Bézier subdivision [Lane&Riesenfeld’81, Rockwood et Al’89]
- B-Spline knot insertion [Mørken&Reimers’07]
Numerical Root Finding

Algorithms: Repeated refinement of control polygon, either
- Bézier subdivision [Lane & Riesenfeld'81, Rockwood et al'89]
- B-Spline knot insertion [Mørken & Reimers'07]
Numerical Root Finding

Algorithms: Repeated refinement of control polygon, either
- Bézier subdivision [Lane&Riesenfeld’81, Rockwood et Al’89]
- B-Spline knot insertion [Mørken&Reimers’07]
Numerical Root Finding

Algorithms: Repeated refinement of control polygon, either

- Bézier subdivision [Lane & Riesenfeld’81, Rockwood et al’89]
- B-Spline knot insertion [Mørken & Reimers’07]
Numerical Root Finding

Algorithms: Repeated refinement of control polygon, either

- Bézier subdivision [Lane & Riesenfeld’81, Rockwood et al’89]
- B-Spline knot insertion [Mørken & Reimers’07]
Properties of rootfinders

- Easy to implement
- Robust and automatic (don’t miss any zeros)
- Quadratic convergence to simple zeros (like Newton’s method)
- Recent extension [Mørken & Reimers] converge quadratically to multiple zeros (e.g. near silhouettes)

Knot insertion framework is very flexible, and allow for variations:
- “Prime” using neighbor rays/previous frames (“Coherency”)
- Detect critical points, search for singularities.
Anti-aliasing

Problem: Silhouette edges fall between rays

- At silhouettes $g(z) = 0$, $g_w(z) = 0$
Anti-aliasing

Problem: Silhouette edges fall between rays

- At silhouettes \( g(z) = 0, g_w(z) = 0 \)
- When \( p_1 \) miss and \( p_2 \) hits
  - Start from minima of \( g(p_1) \)
  - Use Newton on \( (g(s), g_w(s)) \)

\[ p_1 \rightarrow p_2 \]

Internal silhouettes

25% - 50% performance loss
Anti-aliasing

Problem: Silhouette edges fall between rays

- At silhouettes $g(z) = 0, g_w(z) = 0$
- When $p_1$ miss and $p_2$ hits
  - Start from minima of $g(p_1)$
  - Use Newton on $(g(s), g_w(s))$
- Find pixel color by Wus method
  - $\text{color}(p_1) = (1 - \alpha) \text{color}(p_2)$
Anti-aliasing

Problem: Silhouette edges fall between rays

- At silhouettes $g(z) = 0, g_w(z) = 0$
- When $p_1$ miss and $p_2$ hits
  - Start from minima of $g(p_1)$
  - Use Newton on $(g(s), g_w(s))$
- Find pixel color by Wu's method
  - $\text{color}(p_1) = (1 - \alpha) \text{color}(p_2)$
- Internal silhouettes
- 25% - 50% performance loss
Outline

Ray coefficient computation

Root finding

Implementation details

Results and Conclusion
Implementation

Use GPUs for parallelism

Our algorithm is designed for efficiency on modern hardware

- Precompute as much as possible
- Split computations between CPU and GPU for efficiency
  - Also allows controlling the precision
- We use NVidia’s CUDA API
  - Same source code for GPU and CPU versions
- No OpenGL/DirectX involved
Calculating the Frustum Form Coefficients

We find the coefficients $g_{ijk}$ by interpolation

1. Choose $(d + 1)^3$ interpolation points
Calculating the Frustum Form Coefficients

We find the coefficients $g_{ijk}$ by interpolation

- Choose $(d + 1)^3$ interpolation points
- Form tri-tensor $F$ by evaluating $f \circ L$

$$
\sum_{i,j,k=0}^{d} g_{ijk} B_i^d(u_p) B_j^d(v_q) B_k^d(w_r) = f(L(u_p, v_q, w_r))
$$
Calculating the Frustum Form Coefficients

We find the coefficients $g_{ijk}$ by interpolation

- Choose $(d + 1)^3$ interpolation points
- Form tri-tensor $F$ by evaluating $f \circ L$

$$\sum_{i,j,k=0}^{d} g_{ijk} B^d_i(u_p) B^d_j(v_q) B^d_k(w_r) = f(L(u_p, v_q, w_r))$$

- Calculate matrix $\Omega^u = (B^d_i(u_j))$
Calculating the Frustum Form Coefficients

We find the coefficients $g_{ijk}$ by interpolation

- Choose $(d + 1)^3$ interpolation points
- Form tri-tensor $F$ by evaluating $f \circ L$

$$
\sum_{i,j,k=0}^d g_{ijk} B_i^d(u_p) B_j^d(v_q) B_k^d(w_r) = f(L(u_p, v_q, w_r))
$$

- Calculate matrix $\Omega^u = (B_i^d(u_j))$
- In Einstein notation

$$
\Omega^u_{ip} \Omega^v_{jq} \Omega^w_{kr} G_{ijk} = F_{pqr}
$$

\[
\begin{pmatrix}
B_0^d(u_0) & \cdots & B_d^d(u_0) \\
\vdots & \ddots & \vdots \\
B_0^d(u_d) & \cdots & B_d^d(u_d)
\end{pmatrix}
\]
Frustum Form Coefficients cont.

\[ \Omega_{ip}^u \Omega_{jq}^v \Omega_{kr}^w G_{ijk} = F_{pq\ell} \]

- Solve as a nested sequence of matrix products

\[ G_{ijk} = (\Omega^w)_{kr}^{-1} (\Omega^v)_{jq}^{-1} (\Omega^u)_{ip}^{-1} F_{pr\ell} \]

- \( \Omega^{-1} \)-matrices are only dependent on \( d \) \( \Rightarrow \) precompute
- Size of system is small \((d + 1)^3 \Rightarrow \) perform in double on CPU
- Use Chebyshev points for numerical stability
- Not dependent on \( f \) to be a polynomial
Ray coefficient computation

\[
f(r_{pq}) = \sum_{k=0}^{d} \left( \sum_{i,j=0}^{d,d} B^d_i \left( \frac{p}{m} \right) g_{ijk} B^d_j \left( \frac{q}{n} \right) \right) B^d_k(w)
\]

\[c_{pqk}\]

Implemented as matrix products on the GPU
Ray coefficient computation

\[
f(r_{pq}) = \sum_{k=0}^{d} \left( \sum_{i,j=0}^{d,d} B_i^d \left( \frac{p}{m} \right) g_{ijk} B_j^d \left( \frac{q}{n} \right) \right) B_k^d(w)
\]

Implemented as matrix products on the GPU
- Dedicated CUDA kernel computes 4 “slices” in one pass
Root finding computations

Each ray is processed by a CUDA thread

- Memory layout crucial to maximize performance
  - Ensure that memory reads/writes are coalesced
- Degree $d$ is template parameter to specialize algorithms
  
```c
template<int d>
float deCasteljau( float t, float* CP ) {
  float *R[d+1];
  float *L[d+1];

  //
}
```

- Avoid idle threads by “spatial regularity”
  - Rays $8 \times 8$ threadblock converge in $\approx$ same iteration
Outline

Ray coefficient computation

Root finding

Implementation details

Results and Conclusion
$d = 5, 74$ FPS

$\begin{array}{c}
\text{d = 5, 74 FPS} \\
\text{d = 6, 60 FPS} \\
\text{d = 6, 45 FPS} \\
\text{d = 8, 33 FPS} \\
\text{d = 10, 12 FPS} \\
\text{d = 16, 3.1 FPS}
\end{array}$
Conclusion: Benefits of the Frustum Form

- For fixed $m, n, d$ – precompute basis functions
- Reduce algorithmic complexity
  - Evaluation of $c_{pqk}$ requires $(d + 1)^2$ muls/adds.
  - Evaluation of $f$ requires $(d + 1)(d + 2)(d + 3)/6$ muls/adds.
- Univariate ray equations for root finding

\[ f(r_{pq}(t)) = \sum_{k=0}^{d} c_{pqk} B^d_k(t) = 0 \]

- Bernstein form for robust root finding
- Sample based methods might miss thin features
  - Interval Methods[Knoll et. al.] are more robust
  - Fast for sparse polynomials
Future Work

Topologically correct ray-casting

- Beam tracing

Mathematical exploration

- Surface intersections
- Singularity detection

Performance and stability

- High degrees ⇒ use *blossoming* to find Frustum Form coeffs.
- Sort rays based on estimated complexity
Thank you for listening!

Questions?