Ray-Casting Algebraic Surfaces using Stream Computing

Johan Seland – `johan.seland@sintef.no`
Joint work with Martin Reimers

24. January
Geilo Winter School 2008

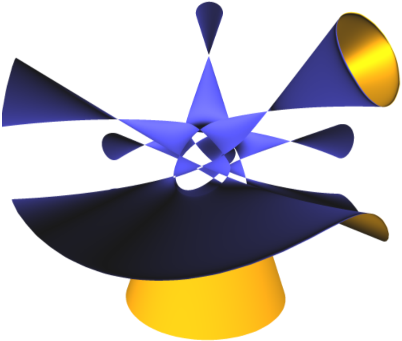
Algebraic surfaces

Definition

For a function $f : \mathbb{R}^3 \rightarrow \mathbb{R}$, an implicit surface can be defined by the level set of the equation $f(x, y, z) = c$, where $x, y, z \in \mathbb{R}$.

Definition

If the function f is a polynomial, it is called algebraic. The resulting surface is called an algebraic surface.



Goals for this talk

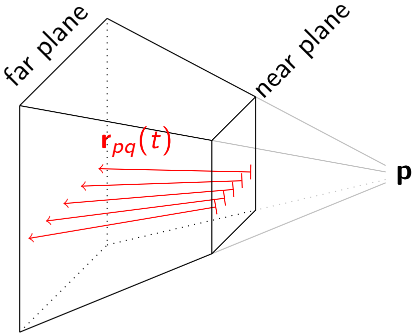
- Give a brief introduction to ray-casting
- Demonstrate hybrid CPU/GPU usage
- Demonstrate pre-evaluation

Raycasting

Raycasting amounts to "shooting" rays inside a view frustum (VF) and determine if they intersect the surface.

- Ray casting has traditionally been a very slow process

- Assume a screen resolution of $(m+1) \times (n+1)$ pixels.
- Pixel (p, q) corresponds to a ray through $\mathbf{p}$ and the pixel with coordinates $(p/m, q/n)$.



Raycasting algorithm

Along a ray $\mathbf{r}_{pq}(t)$, we seek $t \in [0,1]$ such that

$$f((1-t)\mathbf{n}_{pq} + t\mathbf{f}_{pq}) = f(\mathbf{r}_{pq}(t)) = 0$$

Naive approach:

- Work directly on f
- Solve using Newton like method

- Conceptually "easy"
- f is expensive to evaluate
- How to ensure we find the first solution?
- f could be numerically unstable
- Embarrassingly parallel
  - $\Rightarrow$ perfect for stream processing?

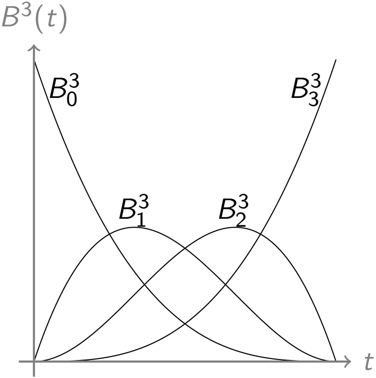
Bernstein Polynomials

We would like to work on a univariate Bernstein polynomials

$$f(\mathbf{r}_{pq}(t)) = \sum_{k=0}^{d} c_{pqk} B_k^d(t) = 0$$

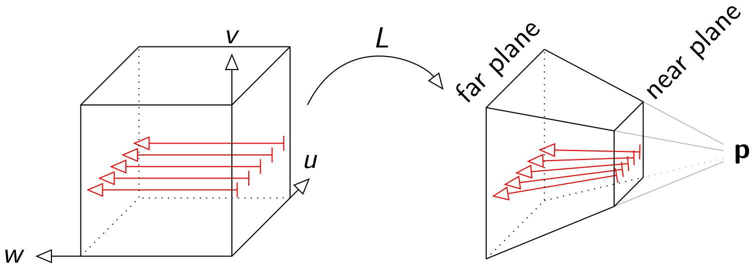
The Bernstein Basis:

- $B_k^d(t) = \binom{d}{k} t^d (1-t)^{d-k}$
- $\Sigma_{k=0}^d B_k^d(t) = 1$
- $b_k^d(t) \geq 0, \; t \in [0,1]$
- Not orthogonal basis
- Proved to be numerically optimal
- Nice algorithms for root finding



The View Frustum Form

Idea: *Parameterize the view frustum over the unit cube, s. t.* $(u, v, 0)$ *and* $(u, v, 1)$ *maps to points on the near and far plane.*



A ray in the view frustum is given by: $\mathbf{r}_{pq}(w) = L(p/m, q/n, w)$. We define the View Frustum Form to be:

$$g = f \circ L : [0, 1]^3 \to \mathbb{R}.$$

Using the composition $g = f \circ L$,

$$f(L(\frac{p}{m}, \frac{q}{n}, w)) = g(\frac{p}{m}, \frac{q}{n}, w) = \sum_{i,j,k=0}^{d,d,d} g_{ijk} B_i^d(\frac{p}{m}) B_j^d(\frac{q}{n}) B_k^d(w)$$

$$= \sum_{k=0}^{d} \underbrace{\left(\sum_{i,j=0}^{d,d} g_{ijk} B_i^d(\frac{p}{m}) B_j^d(\frac{q}{n}) \right) B_k^d(w).}_{c_{pqk}}$$

Yielding univariate ray equations of degree d,

$$f(\mathbf{r}_{pq}(t)) = \sum_{k=0}^{d} c_{pqk} B_k^d(t).$$

Computing VFF Coefficients

We choose to find the VFF coefficients $G = (g_{ijk})$ by solving an interpolation problem.

- Choose $(d+1)^3$ distinct interpolation points (u_p, v_q, w_r) on a grid.
- Solve

$$\sum_{i,j,k=0}^{d,d,d} g_{ijk} \underbrace{B_i^d(u_p)}_{\Omega_p} \overbrace{B_j^d(u_q)}^{\Omega_q} \underbrace{B_j^d(u_r)}_{\Omega_r} = f(L(u_p, v_q, w_r))$$

- Needs inverse of Bernstein collocation matrices $\Omega_p = (B_i^d(u_p))$.
  - Pre-evaluate: Only dependent on d
- Use Chebyshev interpolation points for stability.
- Not dependent on the representation of f.

Computing ray coefficients

Remember $g = f \circ L$:

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

- Basis functions evaluated at every pixel
    - Only dependent on screen resolution and degree
- Pre-evaluate Bernstein collocation matrices as well
    - $M = (m_{ij}) = B_i^d(\frac{p}{m})$, $N = (n_{ij}) = B_i^d(\frac{q}{n})$

Algorithmic flow

This suggest the following “passes”:
For a given d and screen resolution:

- Pre-process:
  - Evaluate the inverse Ω matrices
  - Evaluate the pre-evaluated ray-polynomials M, N
- For each frame
  - Evaluate $f \circ L$ at $(d+1)^3$ interpolation points
  - Apply Ω^{-1}
  - Calculate ray-coefficients MC_kN
  - Find first intersection of each ray
  - Shade all intersections based on normal-estimate

Algorithmic flow

This suggest the following "passes":
For a given d and screen resolution:

- Pre-process:
    - CPU: Evaluate the inverse Ω matrices
    - CPU: Evaluate the pre-evaluated ray-polynomials M, N
- For each frame
    - CPU: Evaluate $f \circ L$ at $(d+1)^3$ interpolation points
    - CPU: Apply Ω^{-1}
    - Transport ray coefficient to GPU
    - GPU: Calculate ray-coefficients $MC_k N$
    - GPU: Find first intersection of each ray
    - GPU: Shade all intersections based on normal-estimate

Matrix-Tensor product

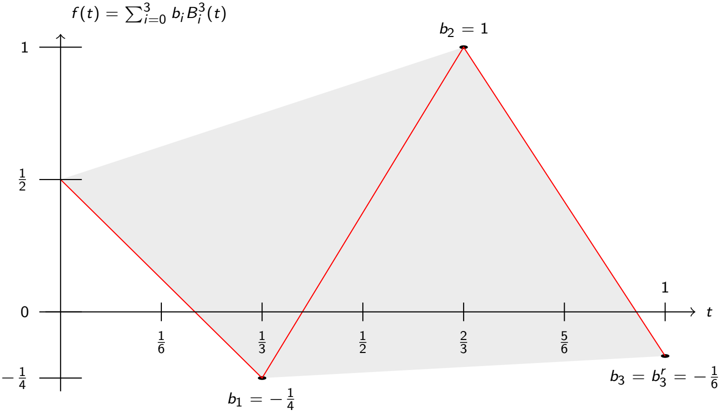
For each frame we calculate the ray coefficients

$$C_k = MG_kN$$

Matrix-Tensor product is perform in a dedicated CUDA kernel

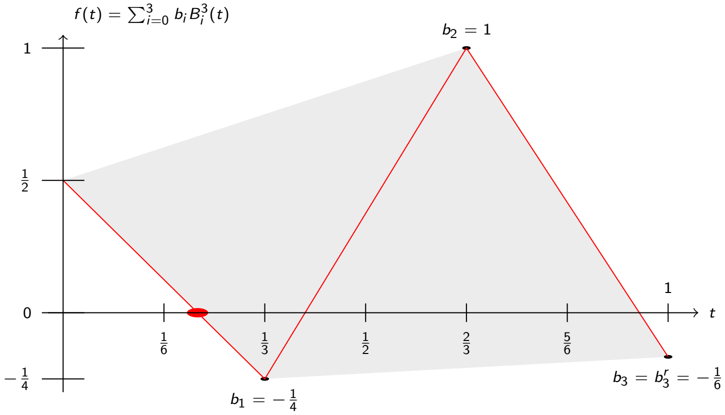
- M and N are stored in texture memory
- G are stored in constant memory
- Coalesced read of `float4` into shared memory
- Blockwise matrix-multiply
- Coalesced write of ray coefficients (4-components at a time)
- Repeated $(d/4+1)$ times

Root finding



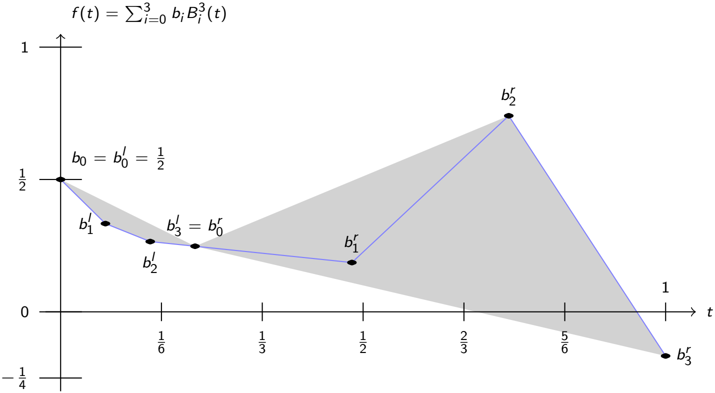
Init: We only know the control points

Root finding



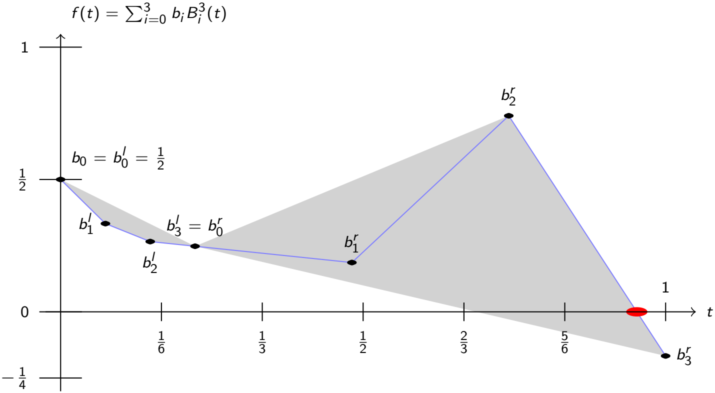
Find the first root of the control polygon, $t = 2/9$

Root finding



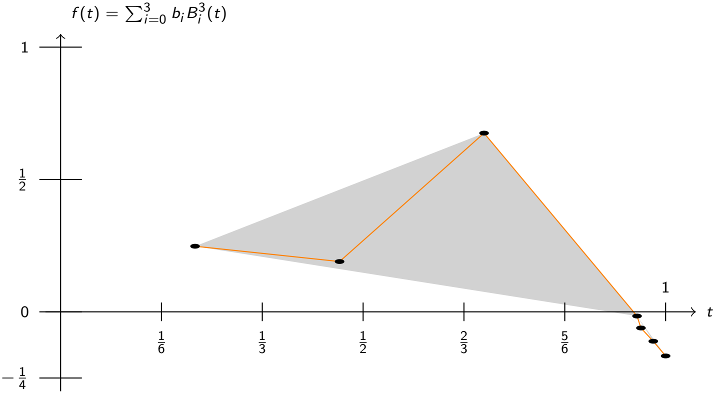
Subdivide at t → two new control polygons

Root finding



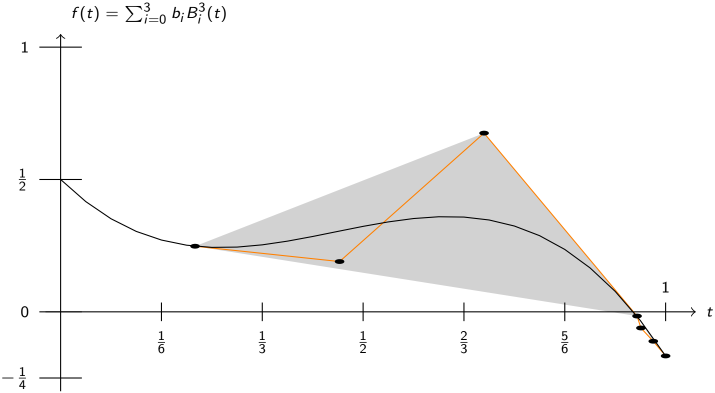
Again, find zero of control polygon – subdivide

Root finding



Yields two new control polygons – repeat

Root finding



Method converges quadratically

CUDA implementation of root finding

- Each ray is processed by a ray
- Coefficients are read (coalesced) 4-coeffients at a time
- The root finding kernel is specialized per degree
- Can lead to very divergent behavior within one warp
    - Future work: Predict behavior based on ray coefficients

Results

- Visualizes surfaces up to degree 18
- 24 FPS for $d = 8$, 12 FPS for $d = 10$, 3FPS for $d = 18$
- Accepted to Eurographics 2008
- Fierce competition
    - Several approaches to this problem fight for performance crown
- OpenMP performance much lower due to thread startup cost

That's all folks!

Thank you for listening

Shameless advertisement!

SINTEF has open positions if you are interested in (GP)GPU, Cell or CMP programming!