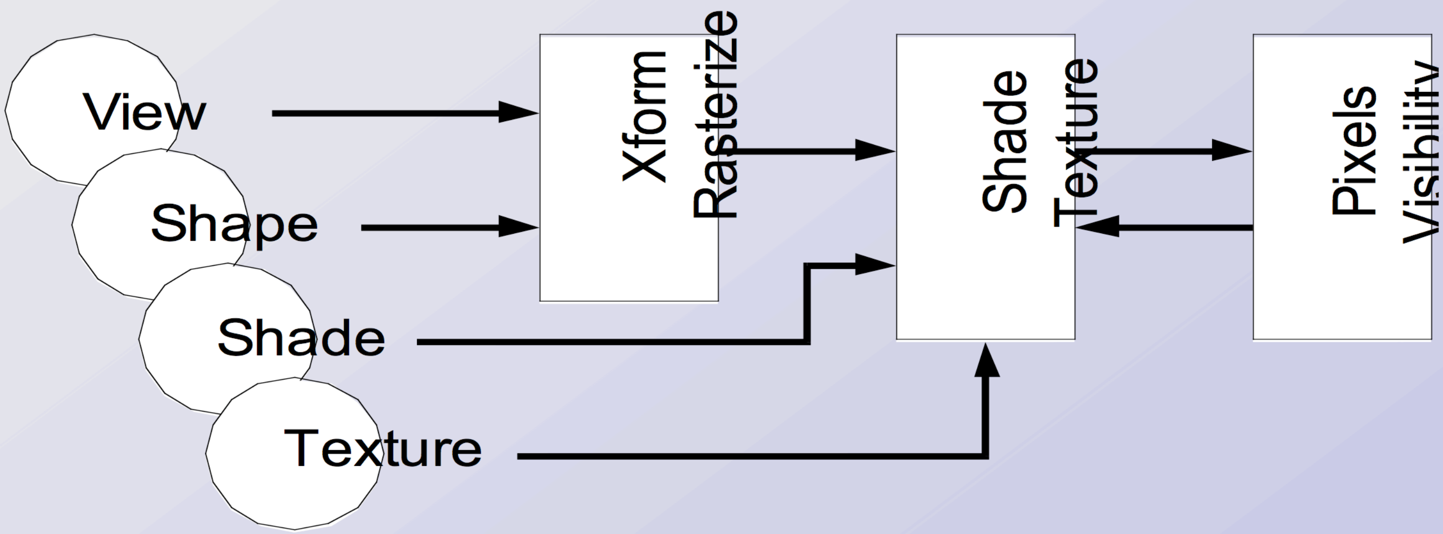
# Fully Procedural Graphics

Turner Whitted, Jim Kajiya
Microsoft Research

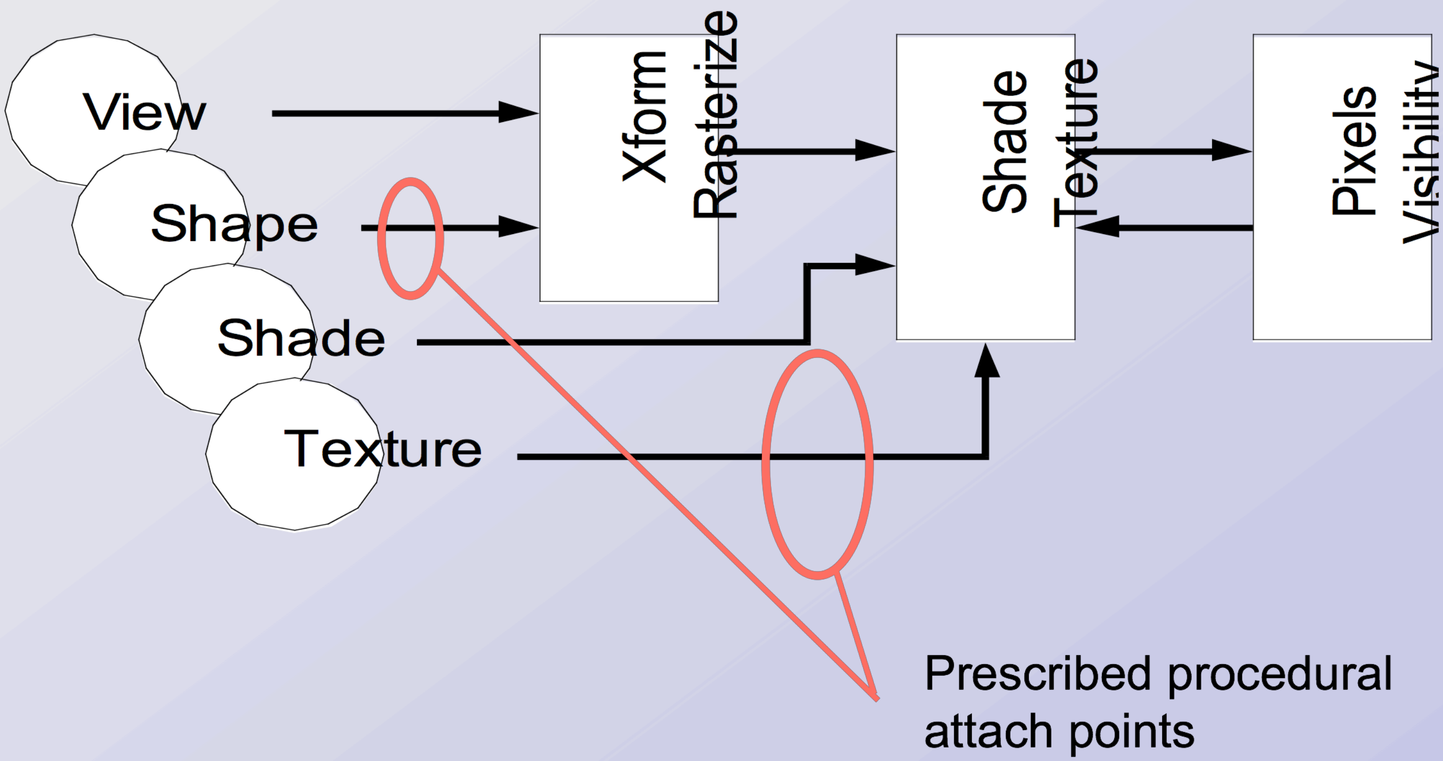
# Motivation: Commentary on *The Pipeline*

- We've done graphics the same way for 30 years
  - Display lists, polygons, pixels, …
    - Nice, clean plumbing
  - Tack on procedures at pre-defined attach points
    - Structure is not so clear
- Hypothesis: procedures *are* graphics
  - And the old conceptual framework just gets in the way
  - Let's think outside the pipe.

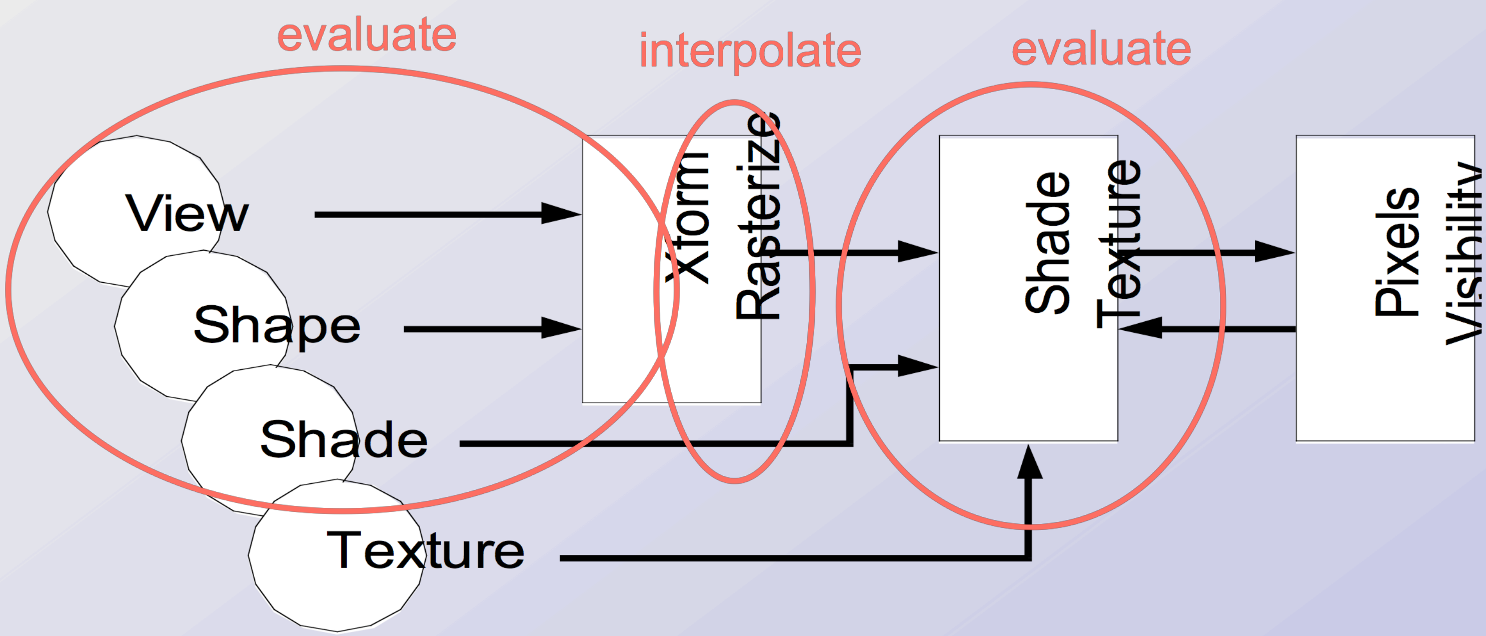
# Conventional graphics



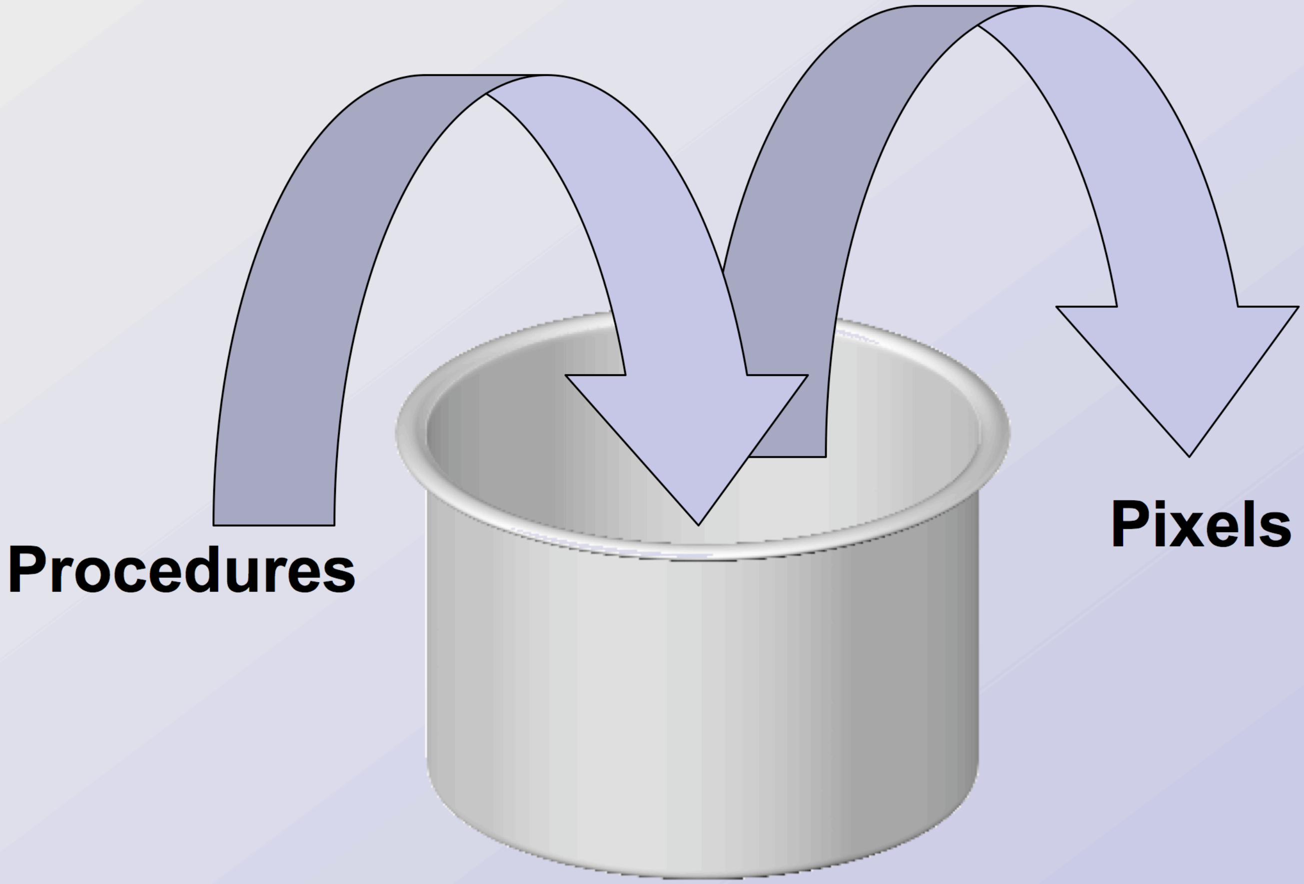
# Conventional graphics



# Conventional graphics



# Procedural soup

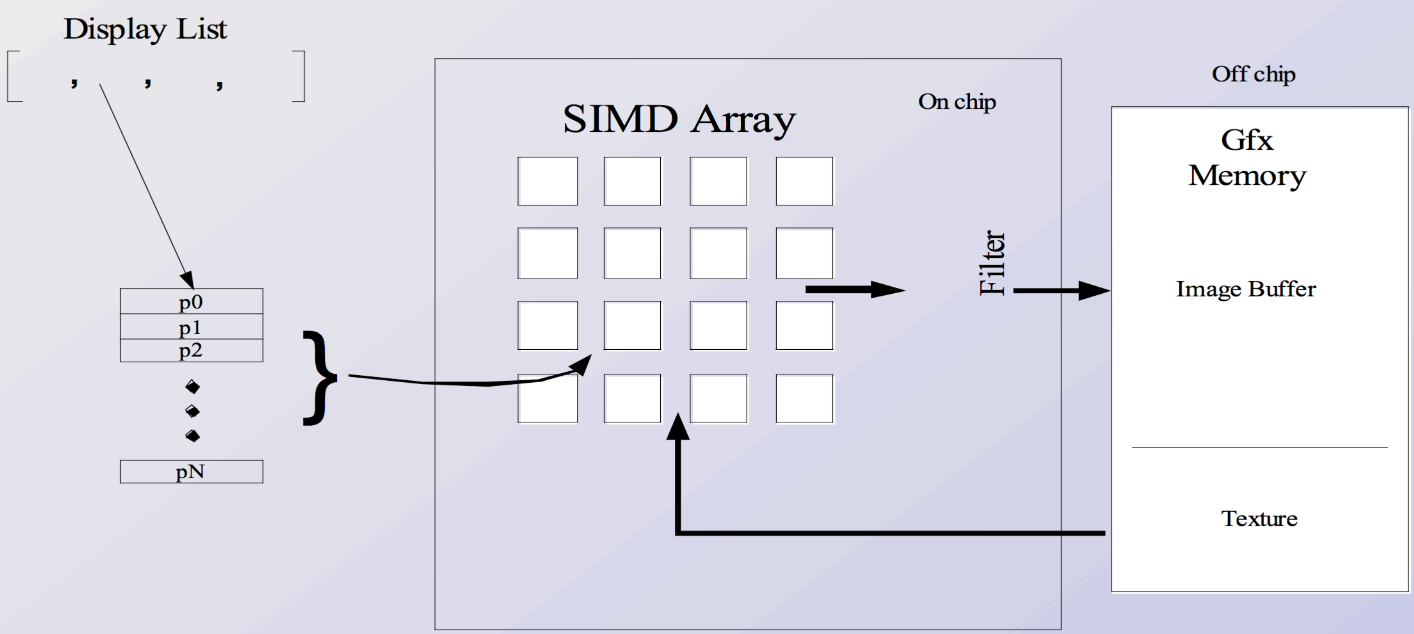



Basic rendering process: toss into the pot and stir.

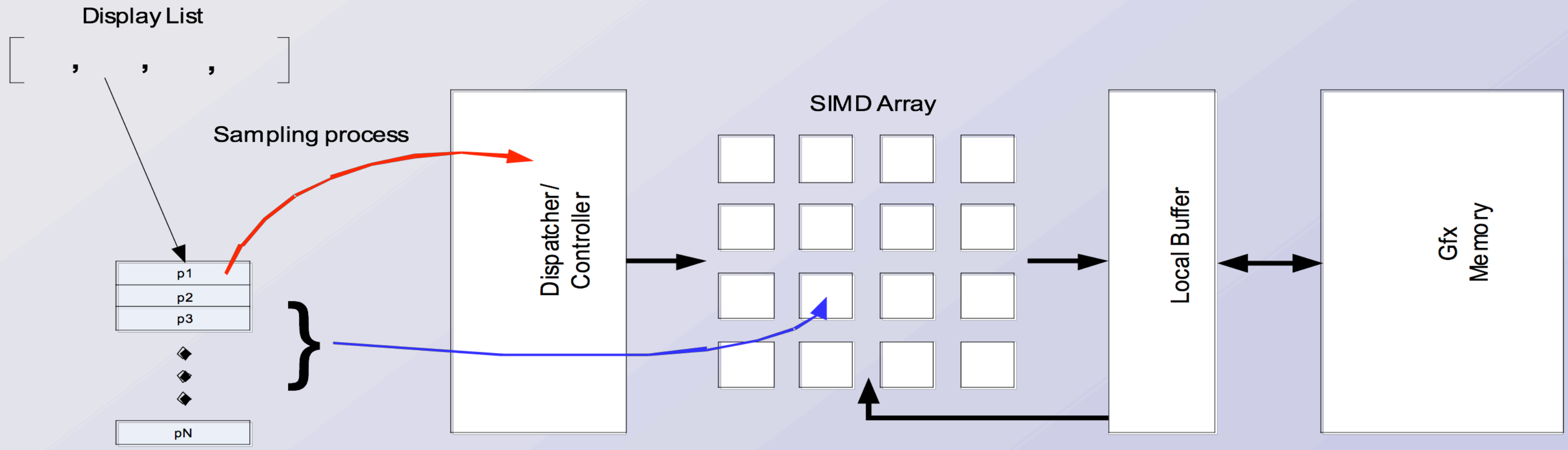
# Procedural testbed: overview

- Testbed structure
- Procedural representations
- Sampling details
- Examples/Features/Performance
- What this paper should have really been about …

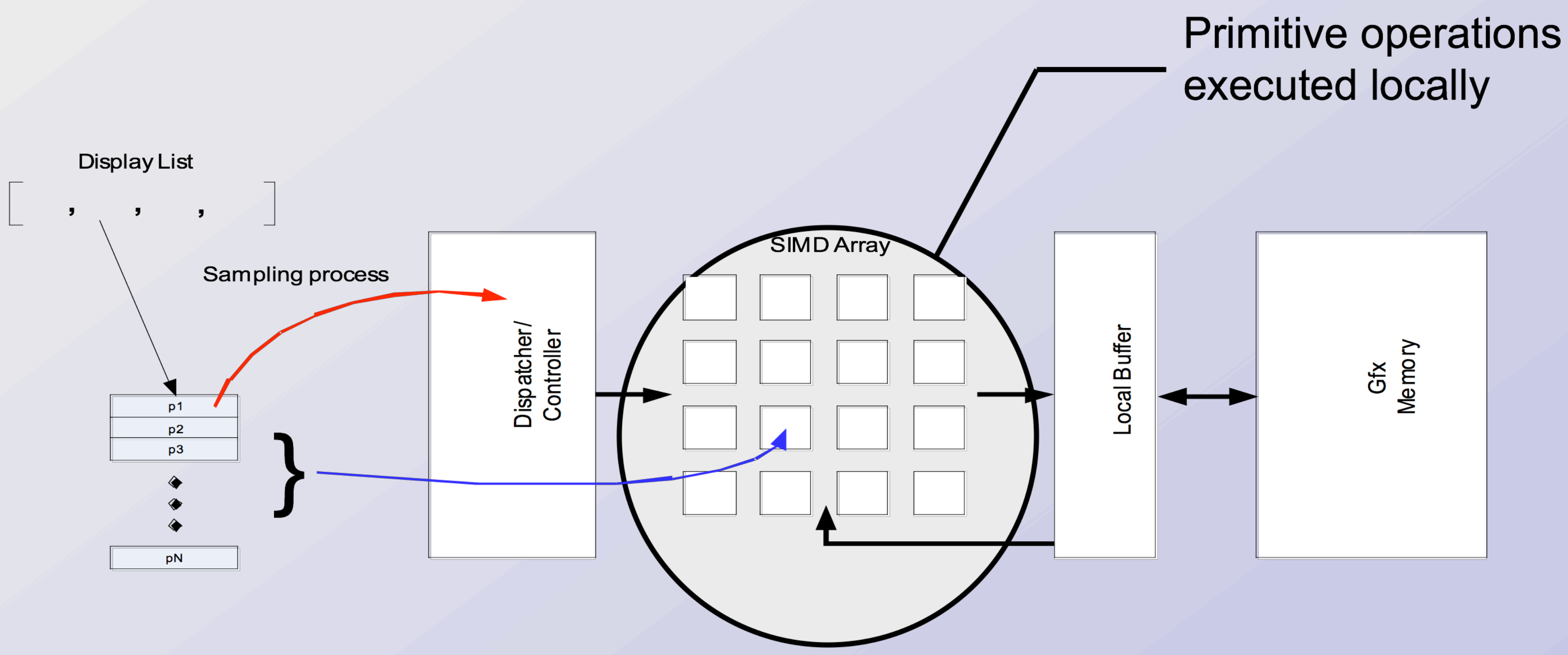
# Brute force testbed



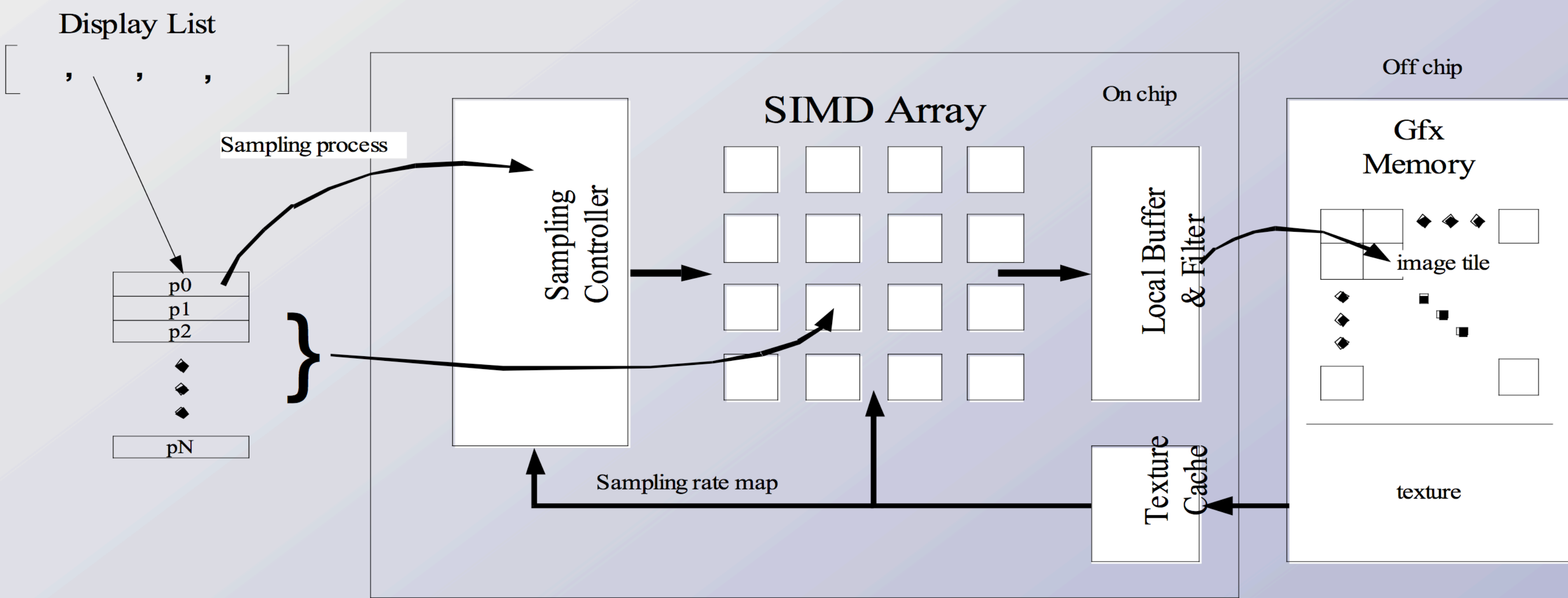
# Practical testbed



# Practical testbed



# Practical testbed - details



# Brute force procedural representation

- Procedural  modules with fixed interprocess interface
- Use points as intermediate results
- Points stored locally
- No compiler
- … there are consequences of this choice

```
struct point
{
        float u, v;
        float x, y, z, w;
        float nx, ny, nz;
        float r, g, b, a;
}
```

# Details of the representations

Partial pseudocode for swept surface

```
// initial section: bi-cubic surface
// compute u curve and normal
// 80 multiplies, 44 adds
ucmp = 1.0-u;
var0 = u*u*u;
var1 = 3.0*u*u*ucmp;
var2 = 3.0*u*ucmp*ucmp;
var3 = ucmp*ucmp*ucmp;
xu = var0*x0+var1*x1+var2*x2+var3*x3;
yu = var0*y0+var1*y1+var2*y2+var3*y3;
nxu = 3*(y0-3*y1+3*y2-y3)*u*u
        +6*(y0-2*y1+x2)*u+3*(y1-y0)
nyu = 3*(-x0+3*x1-3*x2+x3)*u*u+6*(x0-2*x1+x2)*u
        +3*(x1-x0);
scln = recipsqrt(nxu*nxu+nyu*nyu);
nxu = nxu*scln;
nyu = nyu*scln;
// compute v curve and normal
…
// compute point on swept surface
p.x = xu+nxu*xv;
p.y = yv;
p.z = yu+nyu*xv;
```

# Details of the representations

Pseudocode for [large] polygonal surface

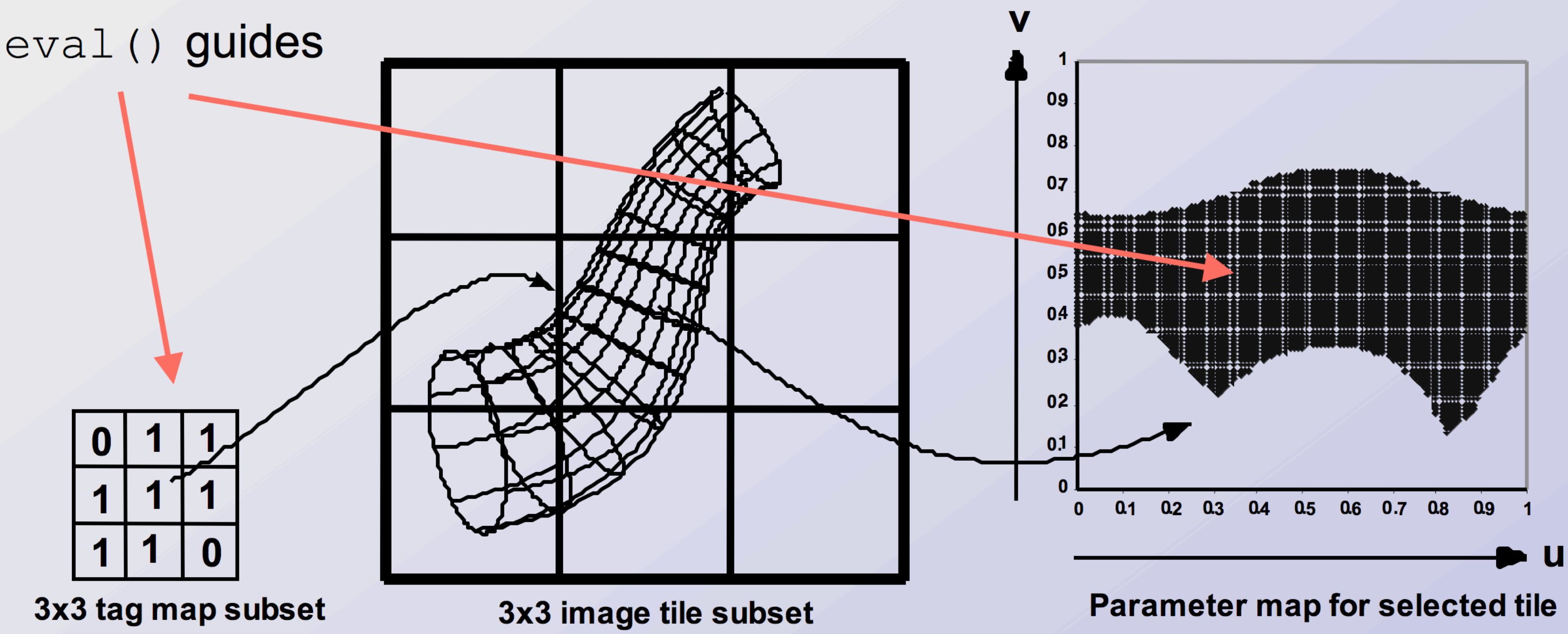
```
// normal is constant
pt.nx = -1.0;
pt.ny = 0.0;
pt.nz = 0.0;

// x is constant
// linearly interpolate y and z
pt.x = xC;
pt.y = yL + t*(yU-yL);
pt.z = zL + s*(zR-zL);
```

Cost of evaluating a surface varies greatly and depends on the type of surface.

For polygons, **`eval()`** costs no more than rasterizer interpolation.

# Details of sampling controller

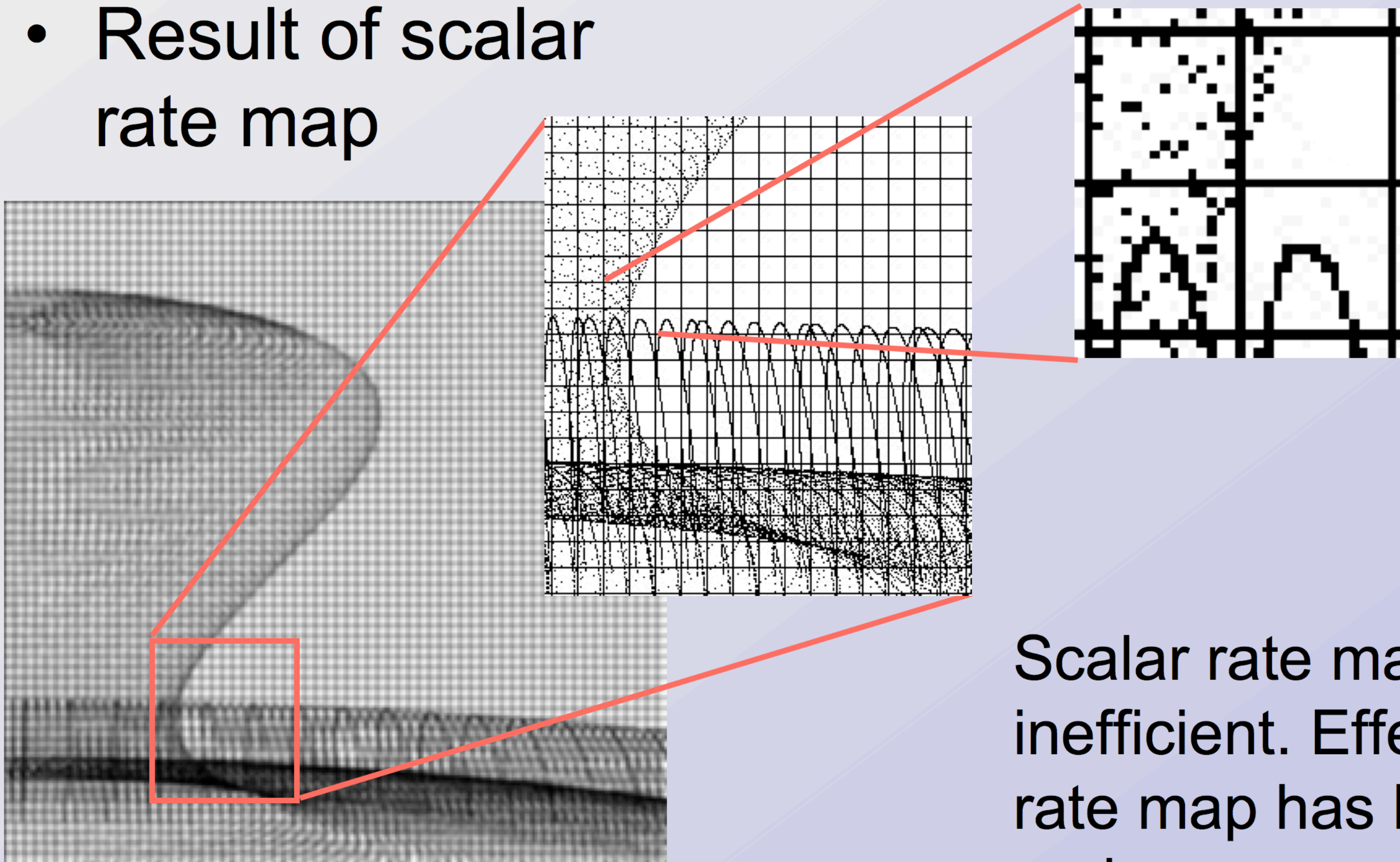



Rendering pass 1: create tag map, uv map, rate map

Rendering pass 2: use the maps to control final sampling

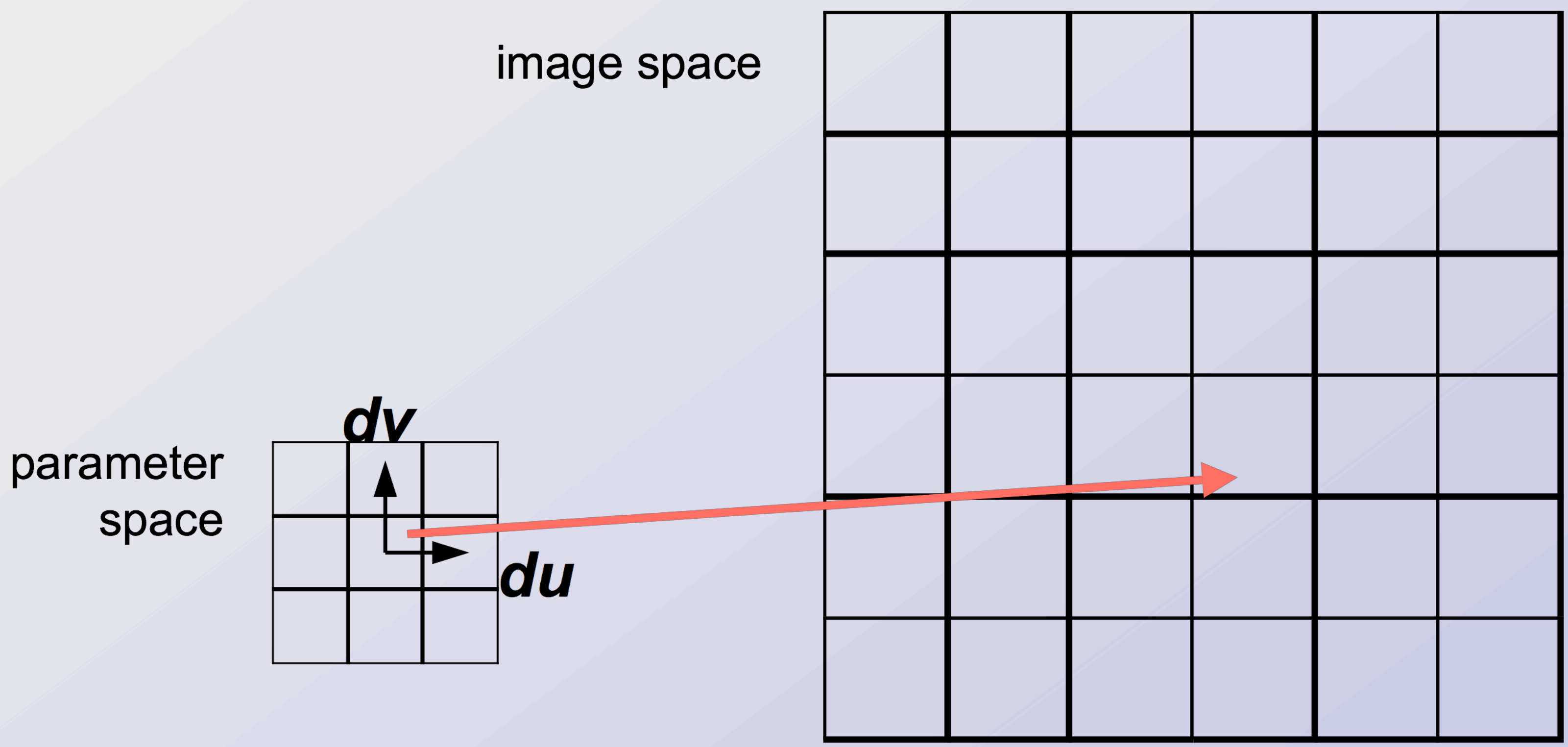
# Details of rate map

- Result of scalar rate map

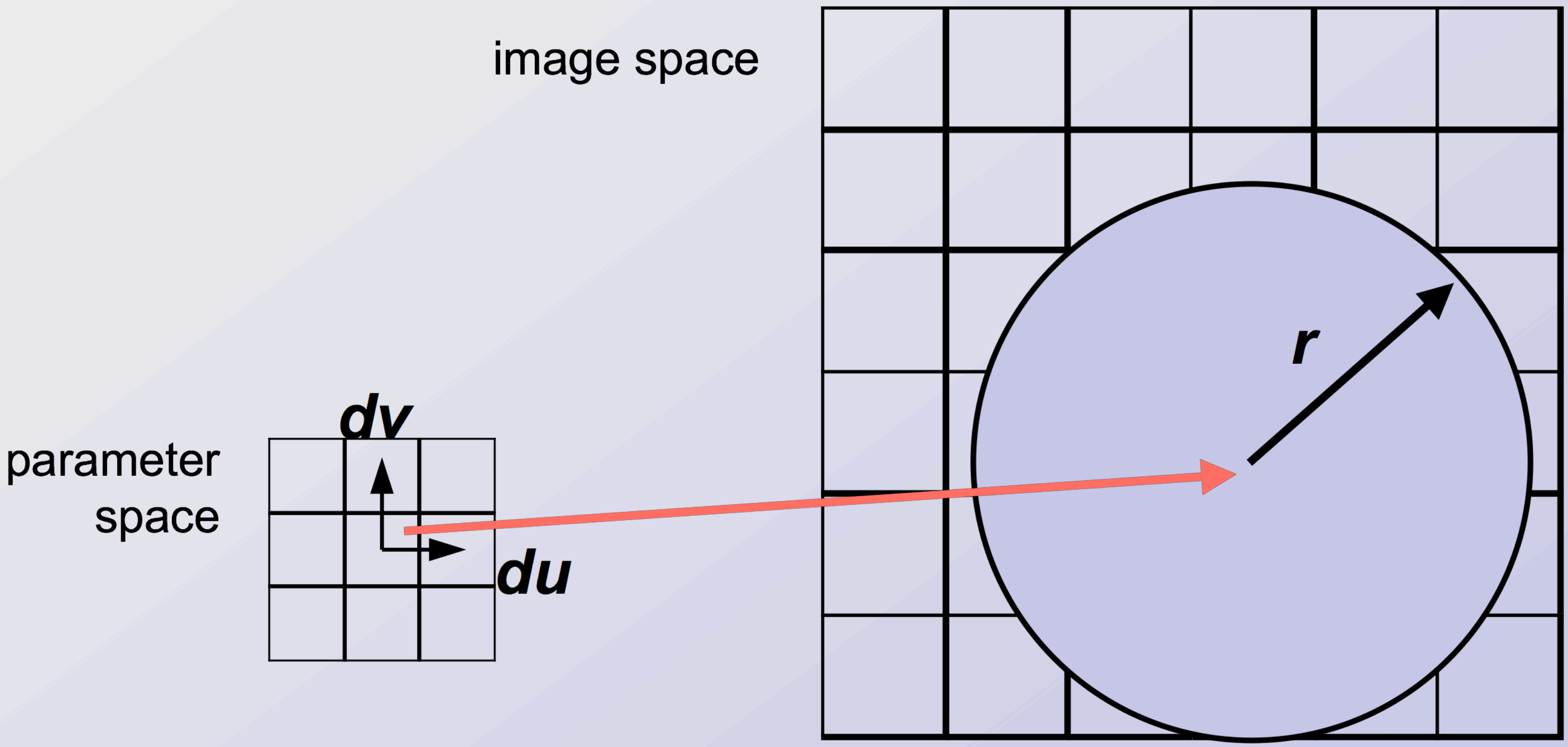


Scalar rate map is inefficient. Effective rate map has both *u* and *v* components.

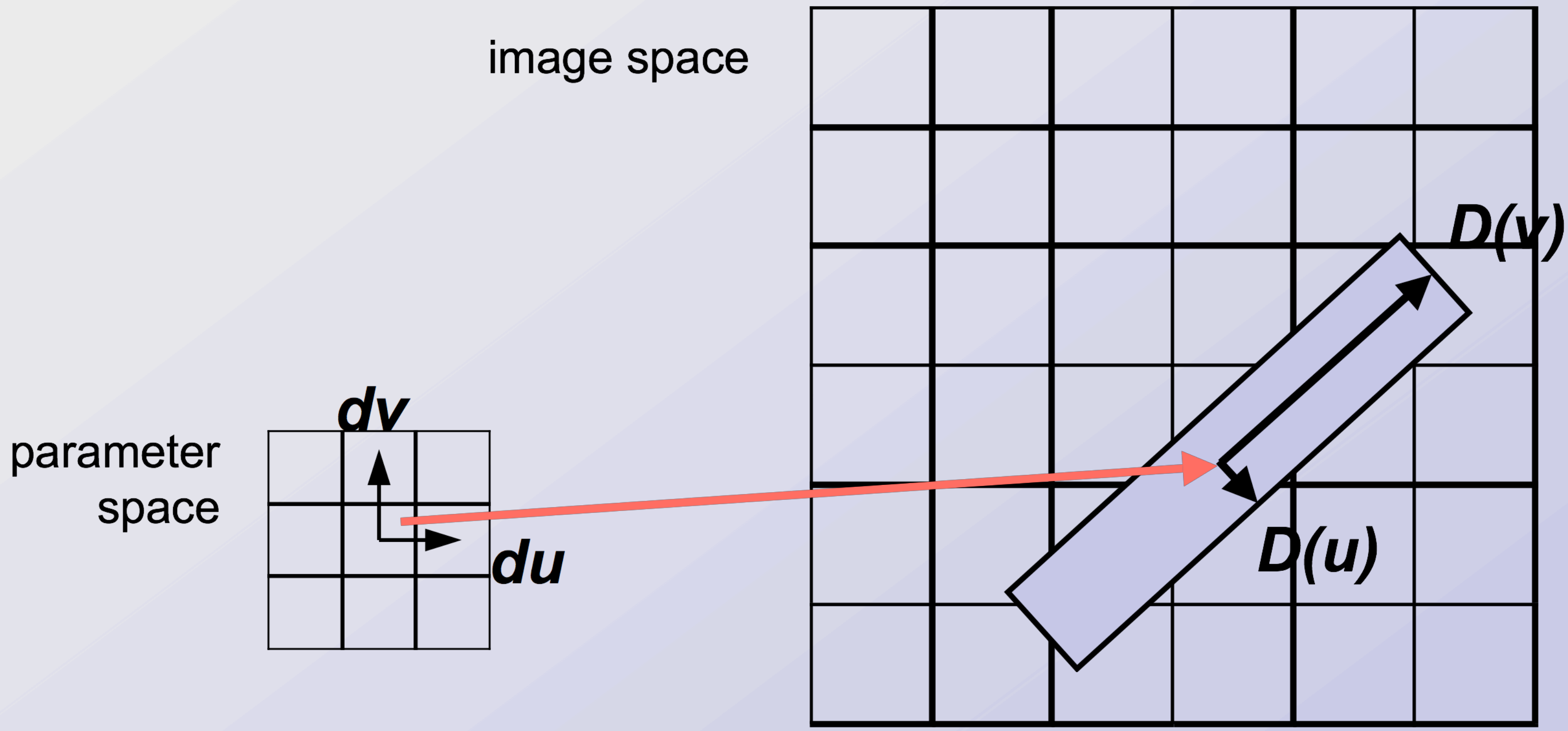
# 2-component rate map



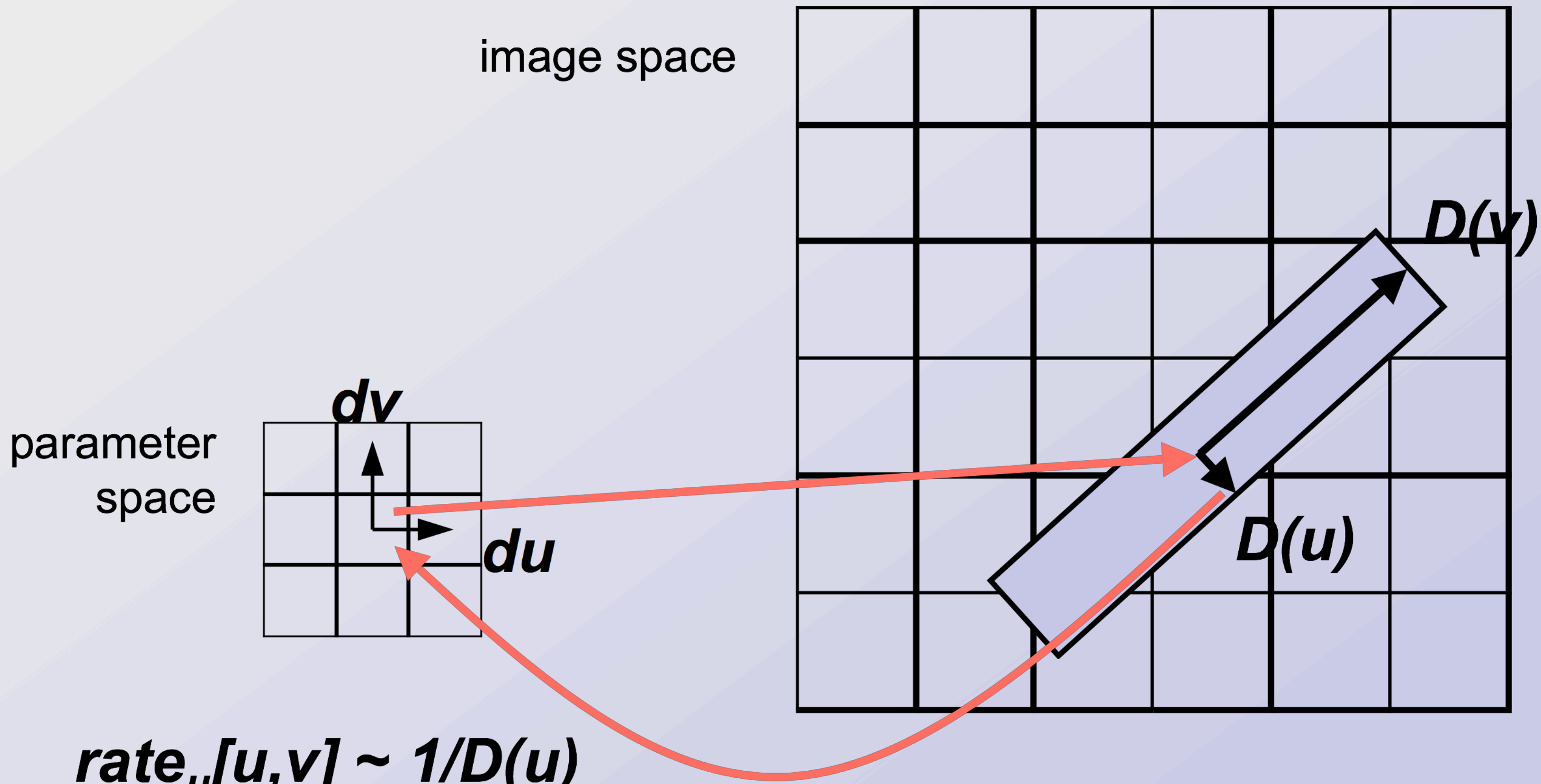
# 2-component rate map



# 2-component rate map



# 2-component rate map

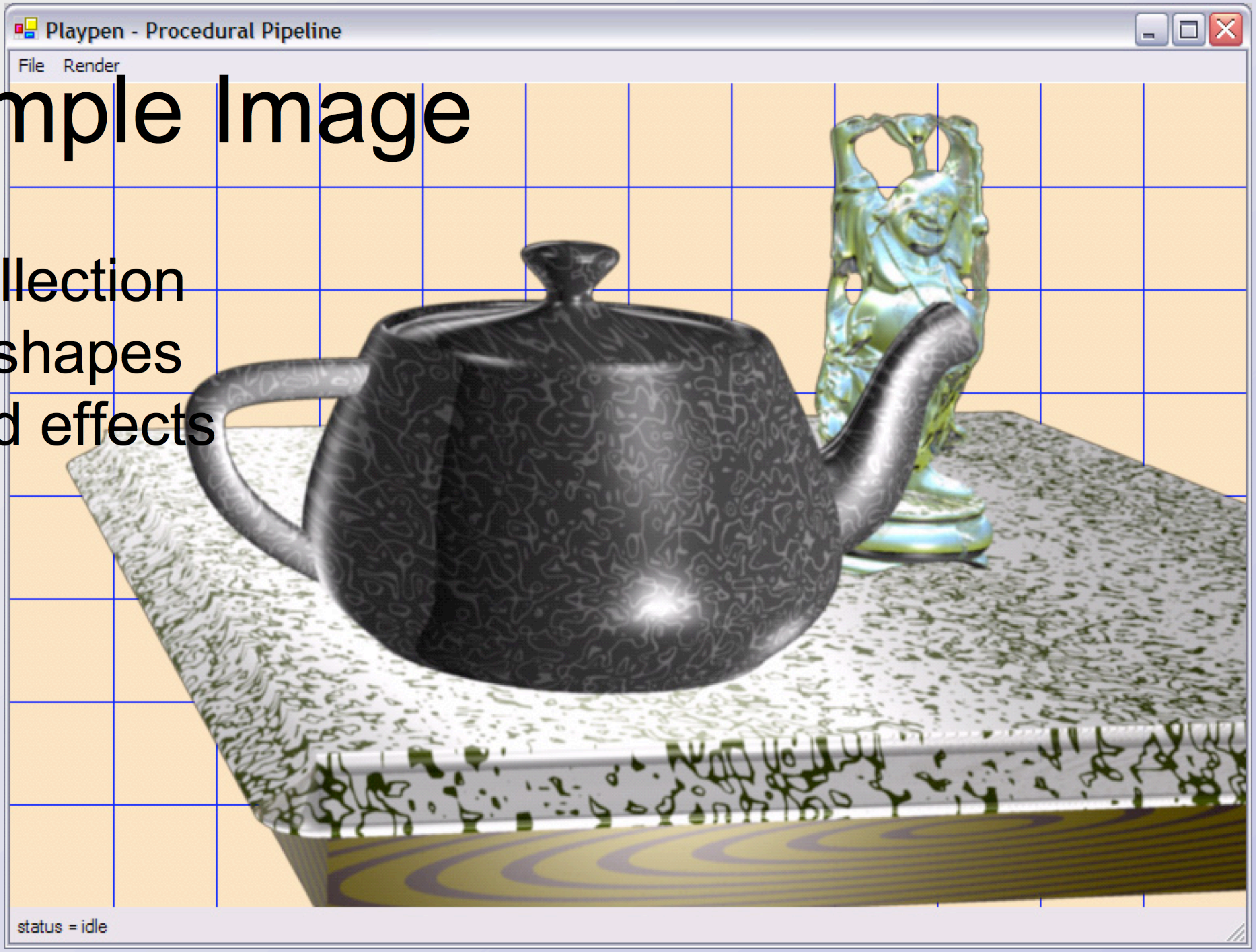



$rate_u[u,v] \sim 1/D(u)$

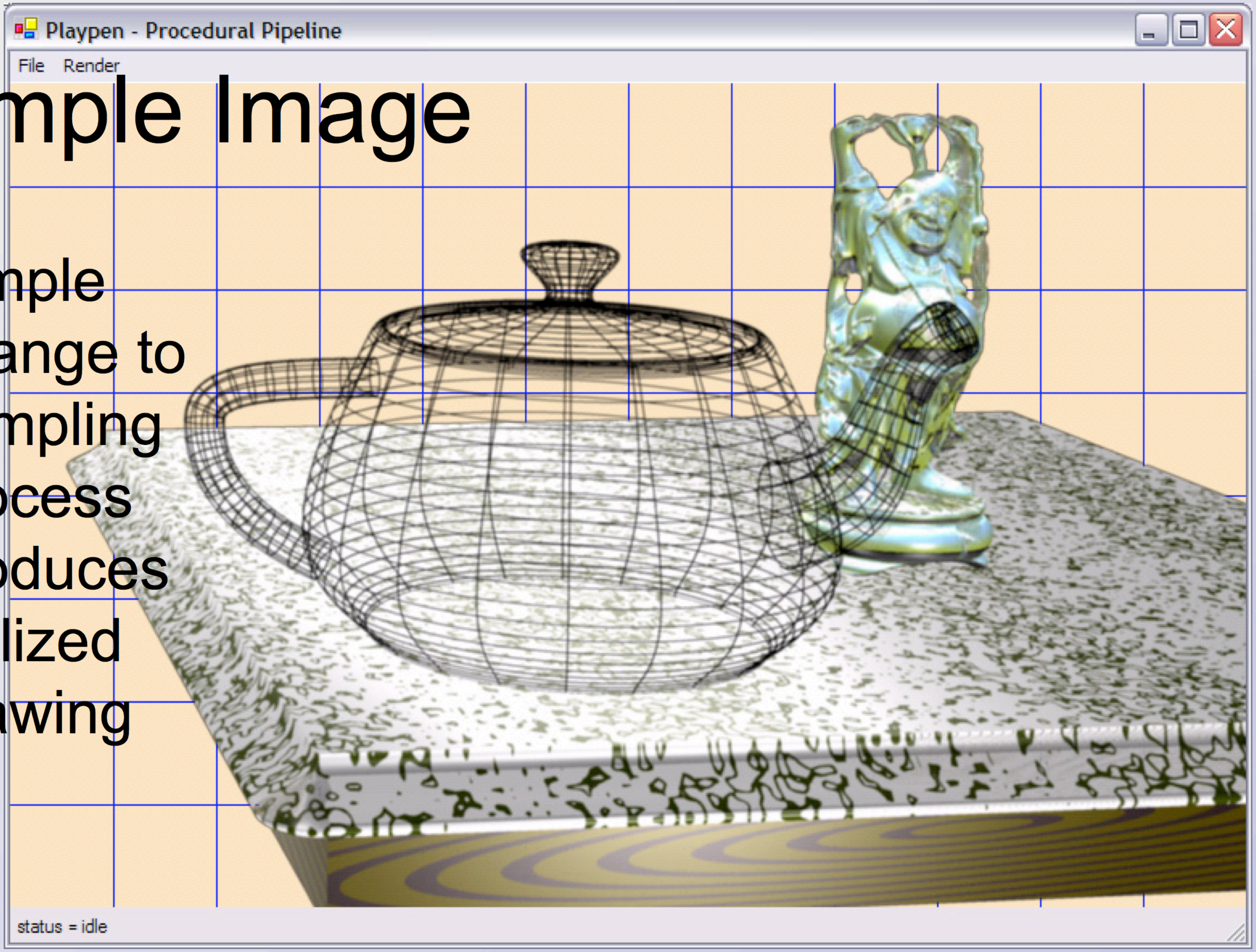
$rate_v[u,v] \sim 1/D(v)$

# Sample Image

- Collection of shapes and effects



# Sample Image



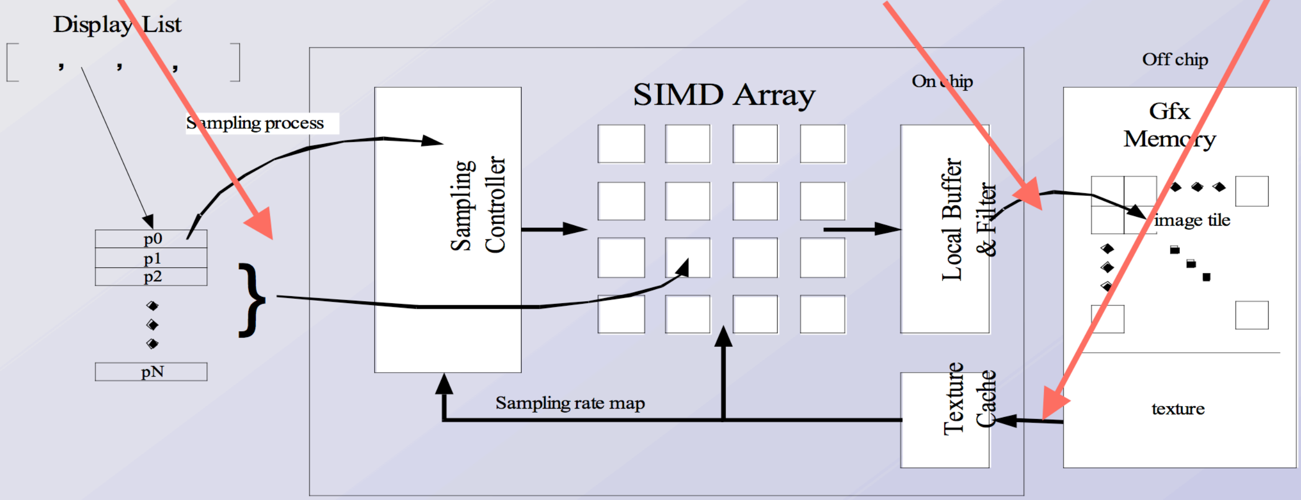

- Simple change to sampling process produces stylized drawing

# Performance – external BW

Procedure transfer (uncached): 6 MB/s
Procedure transfer (cached): 197 KB/s

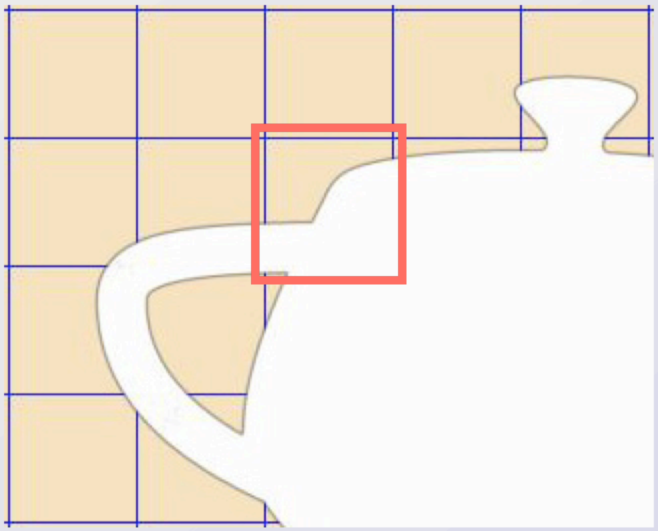
Geometry image read: 90.1 MB/s

Frame buffer R/W: 377 MB/s




… versus 212 MB/s for equivalent small polygon model

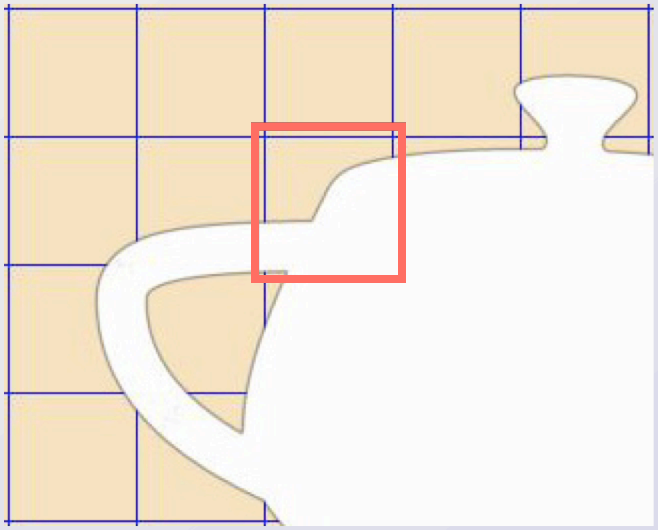
# Performance

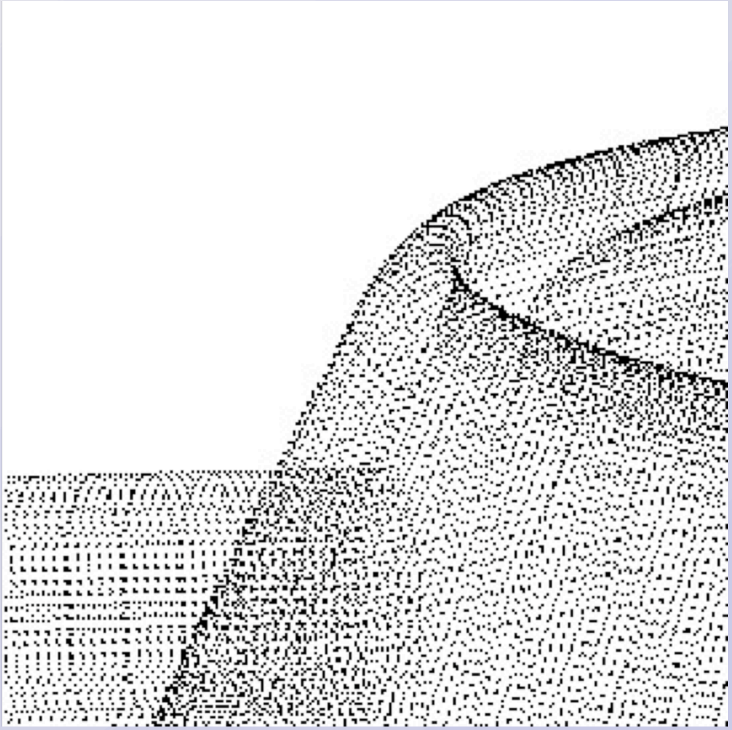


How much sampling density is enough?

# Performance

How much sampling density is enough?

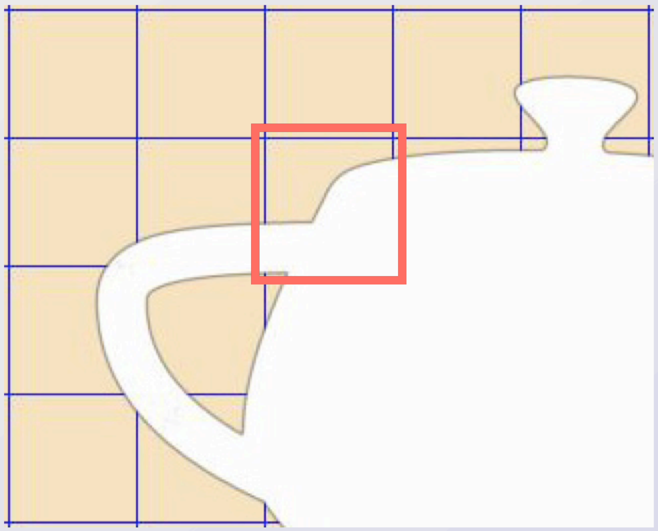


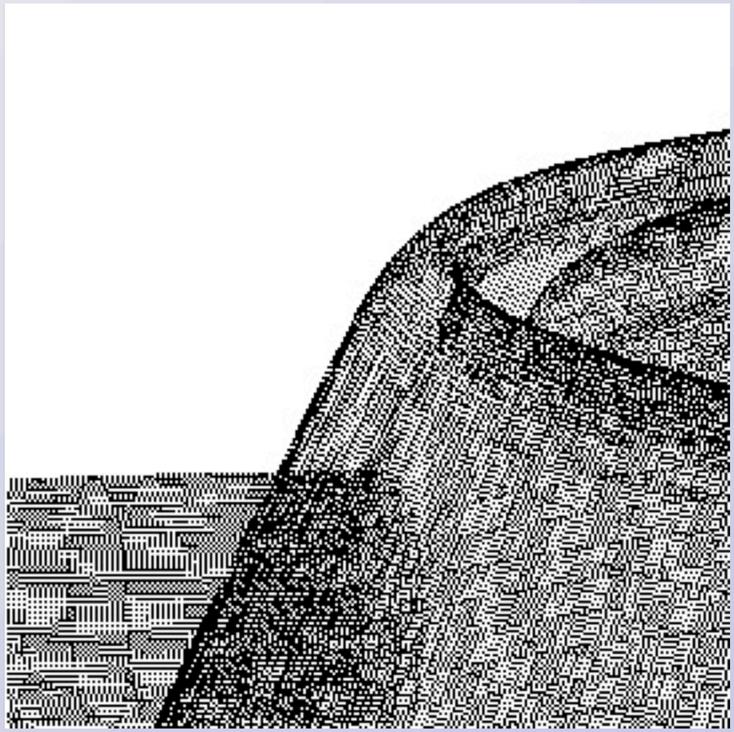


Grossly undersampled

# Performance

How much sampling density is enough?

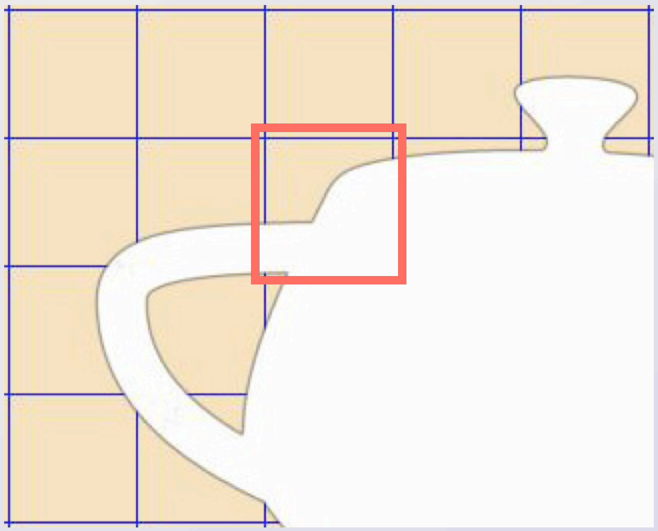


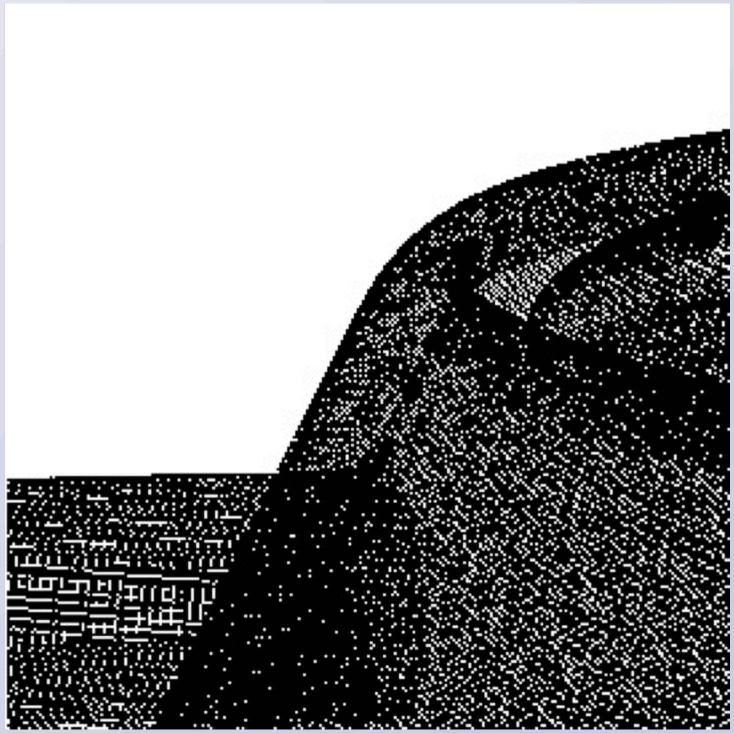


Undersampled

# Performance

How much sampling density is enough?

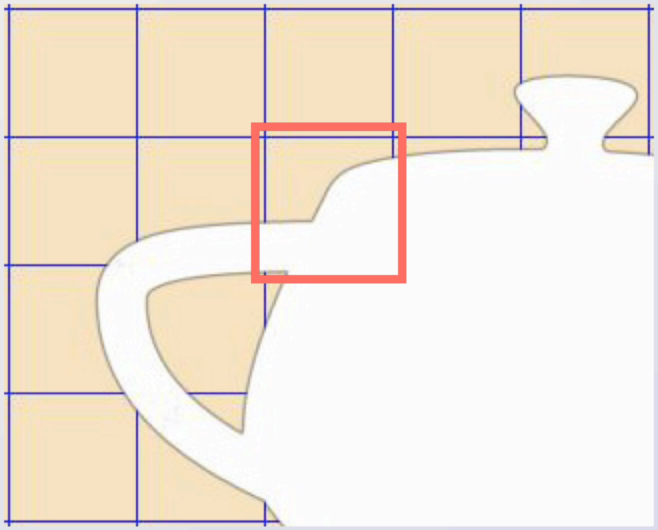


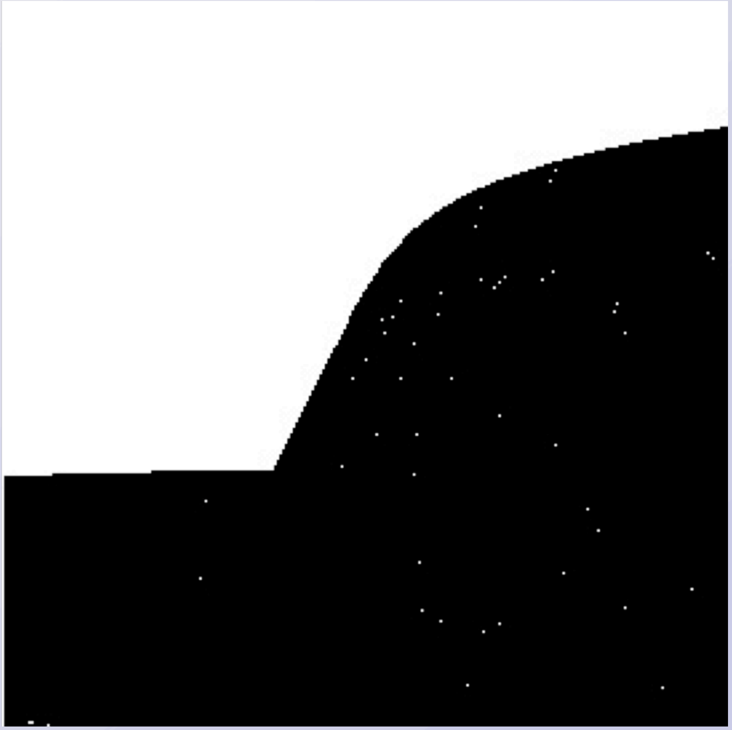


Almost adequately sampled

# Performance

How much sampling density is enough?

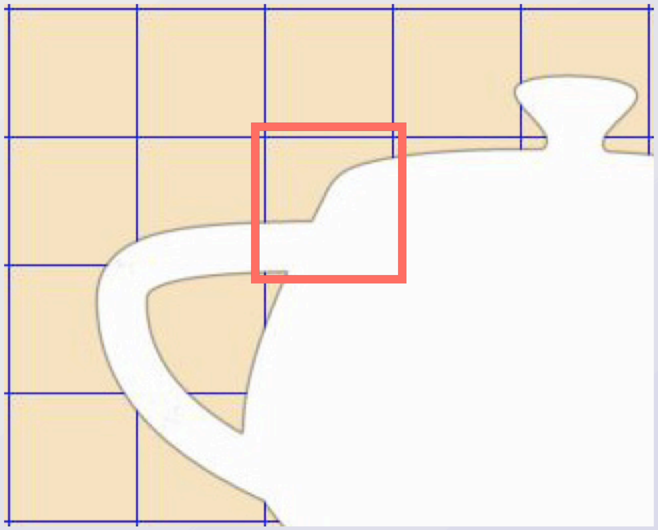


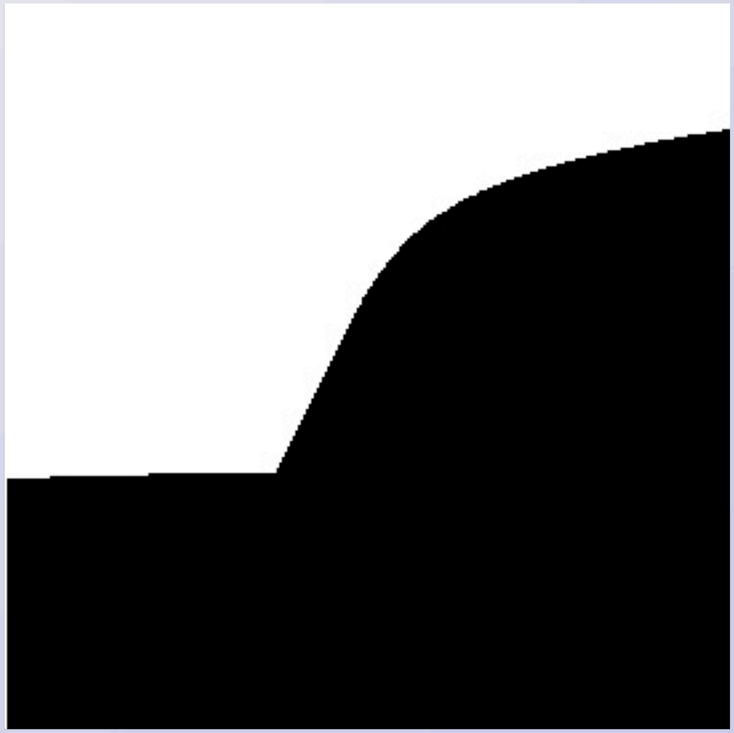


Adequately sampled

# Performance

How much sampling density is enough?





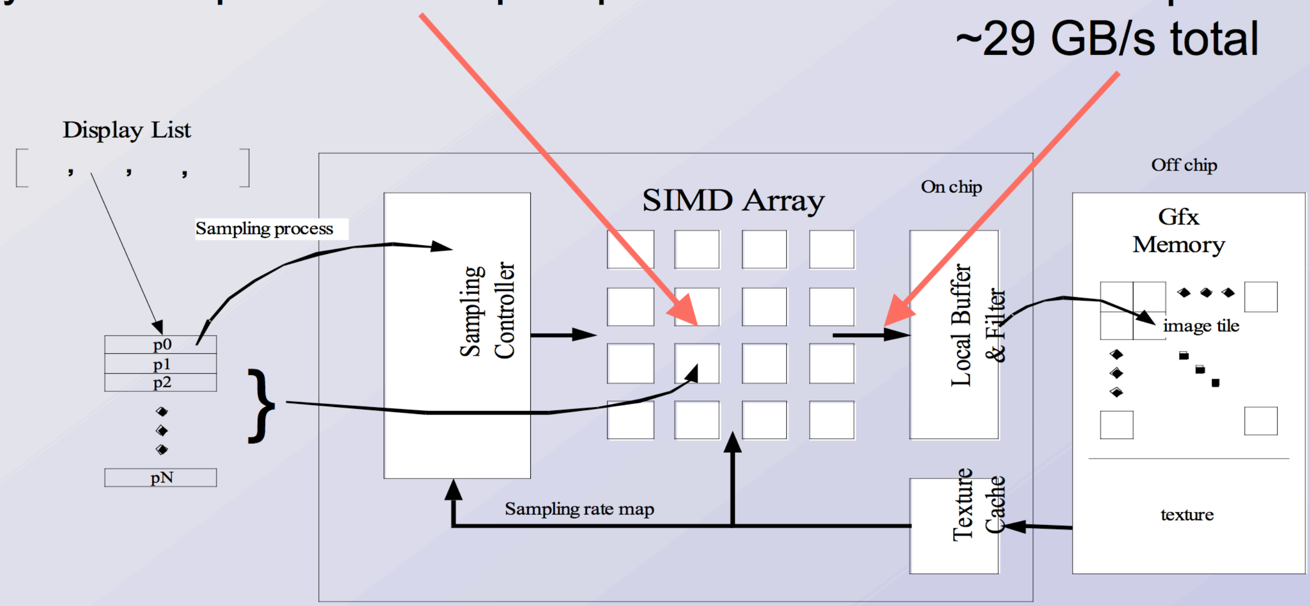
Oversampled

… all performance numbers are based on oversampled case

# Performance - cycles

~140 cycles/sample X 43 samples/pixel

18 M samples/frame
~29 GB/s total




… versus 2 vertices/pixel for equivalent small polygon model

# Rates

- Rasterizer: dual rate
  - Adaptive vertex rate
  - Fixed pixel rate, fixed routing
- SIMD procedural testbed: single rate
  - Adaptive `eval()`, pixel routing
- Future: multi-rate
  - Fully adaptive, flexible routing

# References

- Essential background
  - John Snyder, *Generative modeling for computer graphics and CAD*, Academic Press (ISBN 0-12-654040-3), 1992.
  - Marc Olano, “A programmable pipeline for graphics hardware,” PhD dissertation, UNC-CH, 1998.
- Related work
  - Charles Loop and Jim Blinn, “Resolution independent curve rendering using programmable graphics hardware,” Proceedings of SIGGRAPH 2005.

# Why procedural?

- Flexibility
  - Not demonstrated in these examples, but essentially unlimited
- Compact size
  - Already widely used for shape, shading, and texture
- Efficiency
  - Needs work

# Summary

- Single rate fully procedural display dramatically reduces off-chip bandwidth
  - at the price of dramatically increased processing
- Fully procedural display delivers *all* of the flexibility of procedural representations
- There is a need for more agile representations.

# Acknowledgement


- Hugues Hoppe, Marcel Gavriliu
- GH 2005 reviewers
- *Especially* GH 2005 papers chairs