Programmability Features of Graphics Hardware

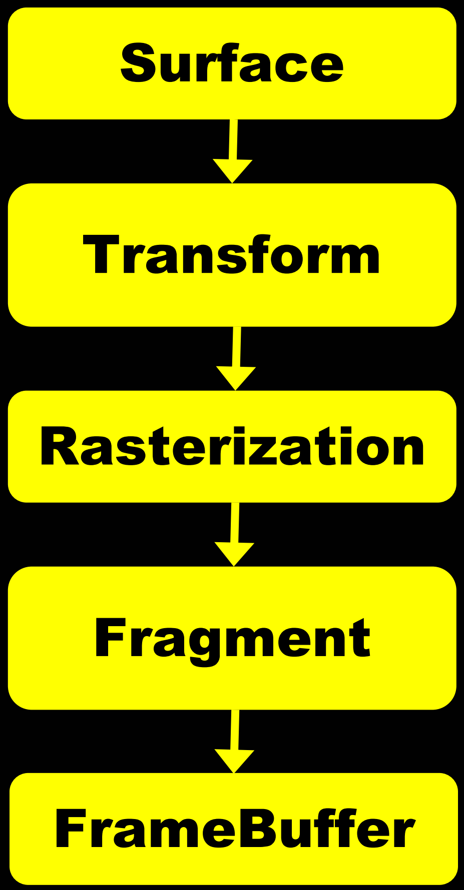
Michael Doggett
ATI Research
MDoggett@ati.com

Outline

- Graphics Hardware
- Transform Stage
  - Vertex Engine
    - OpenGL ARB vertex program
- Fragment Stage
  - Pixel Pipeline
    - OpenGL ARB fragment program
- Examples
  - Mandelbrot
  - FFT
  - Displacement Mapping
- Programming Options
  - OpenGL Shading Language overview
- Computation on GPUs

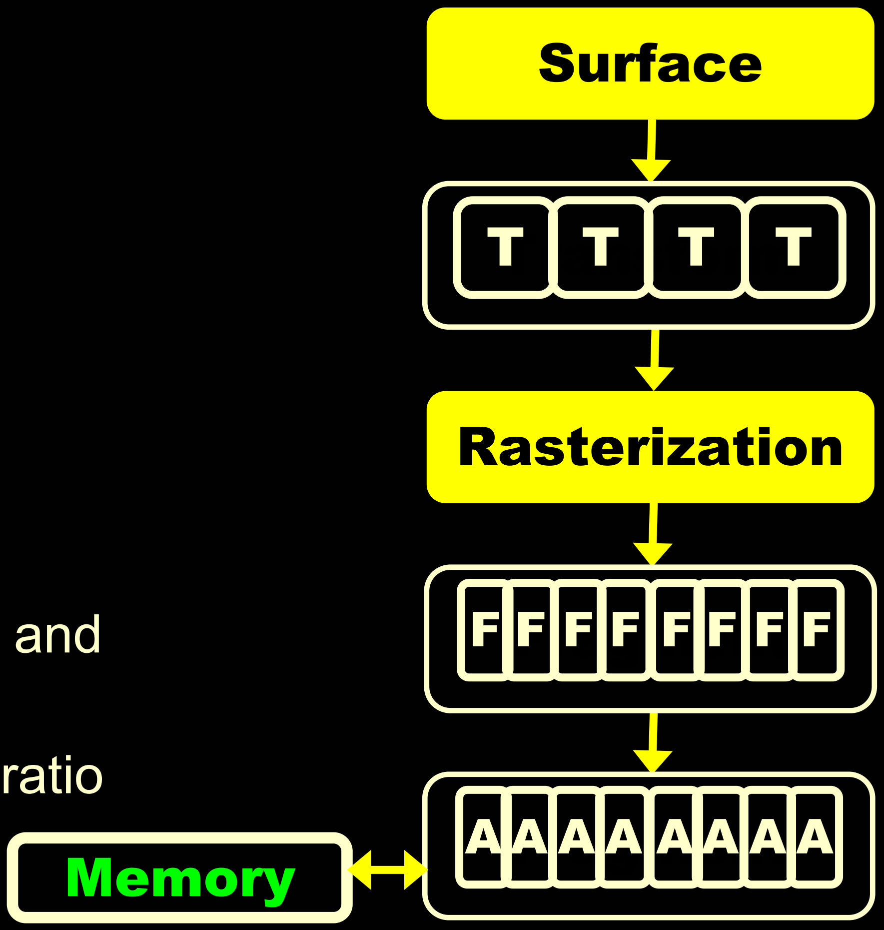
Graphics Hardware

- Hardware pipeline



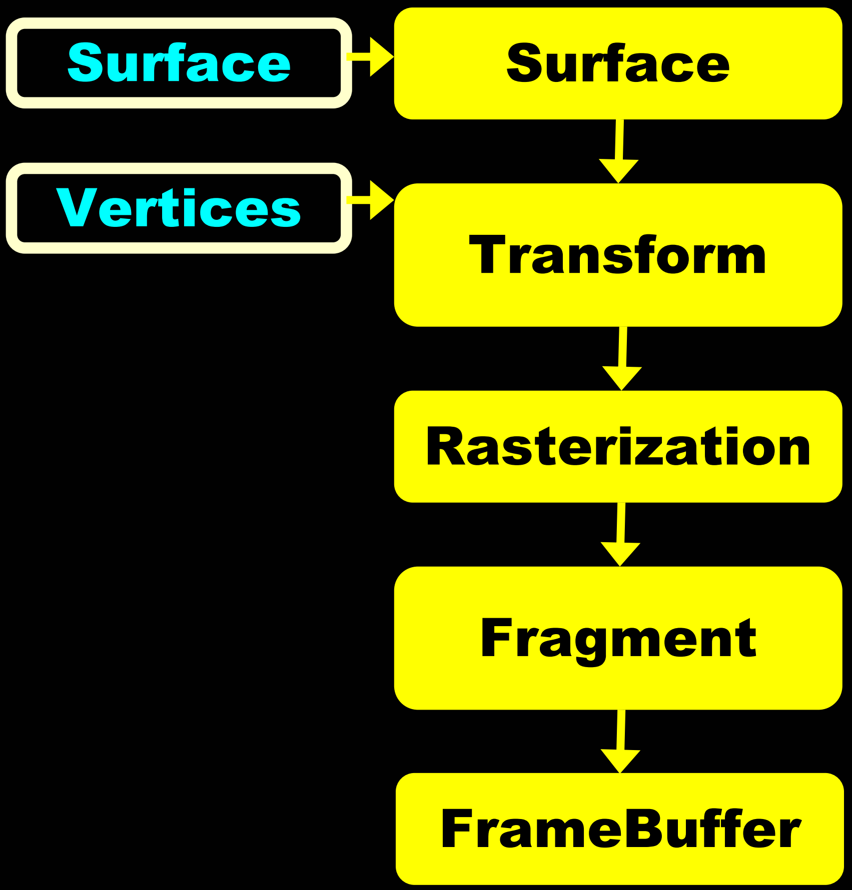
Graphics Hardware

- Hardware pipeline
  - Based on RADEON 9800
    - 380MHz
  - SIMD stages
    - Transform
    - Fragment
  - Highly parallel
  - 4 component vector registers and operations
  - High computation/bandwidth ratio
    - Arithmetic intensity



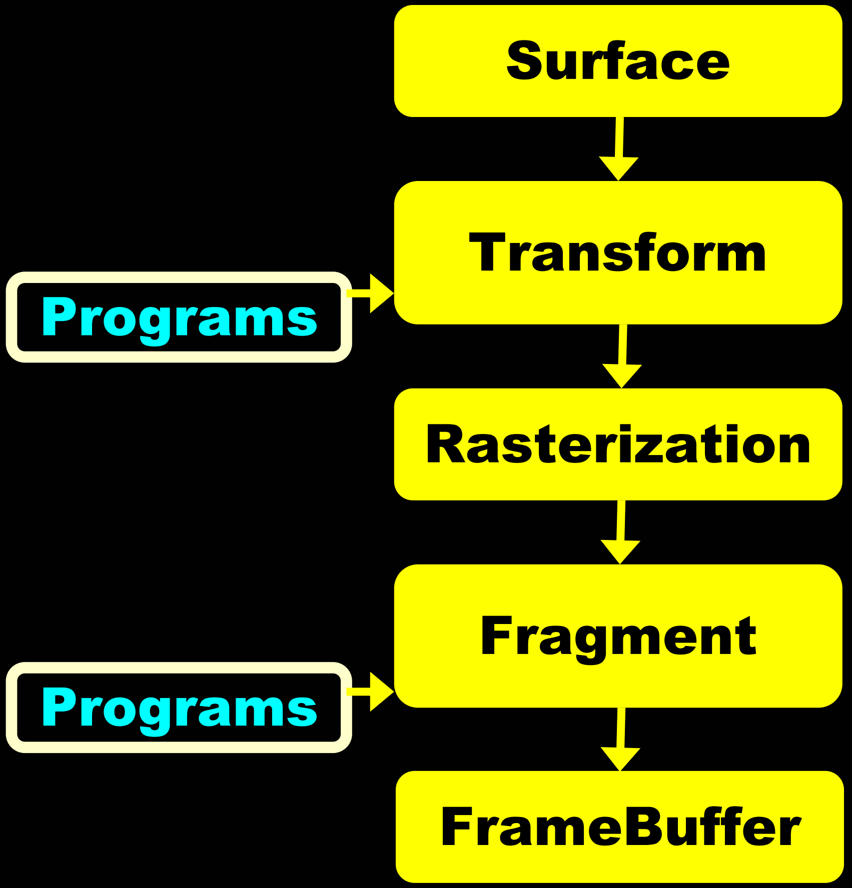
Graphics Hardware

- Input – 3D Scene
  - Typically triangles made up of vertices
  - 1D buffers of vertex data



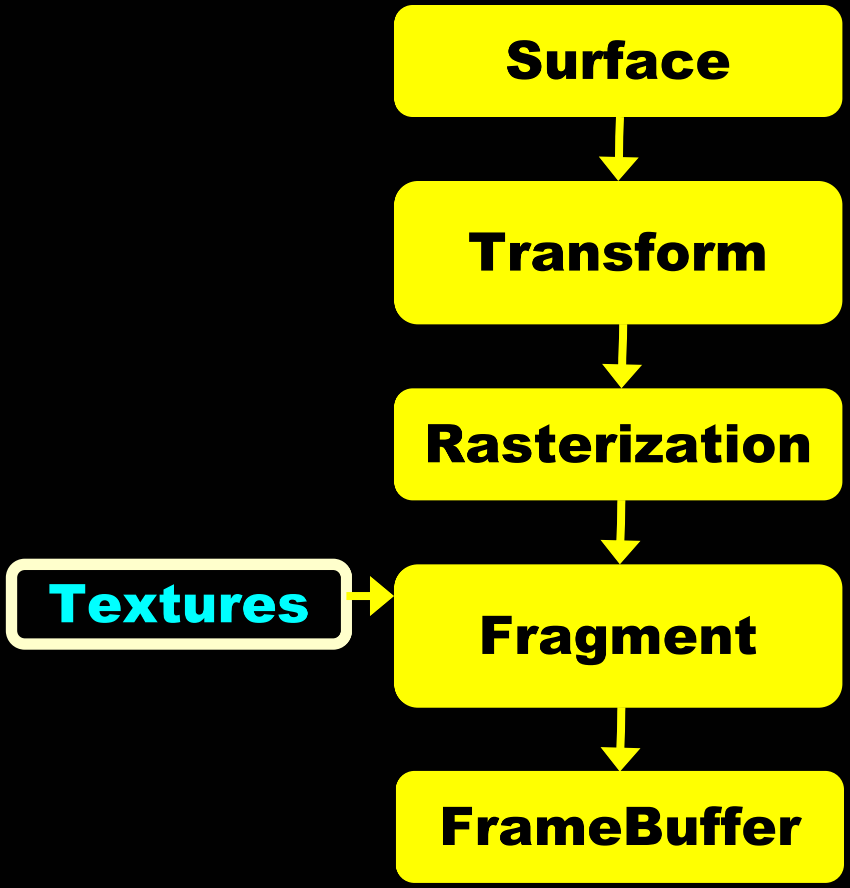
Graphics Hardware

- Input – 3D Scene
  - Typically triangles made up of vertices
  - 1D buffers of vertex data
  - Vertex and Fragment programs



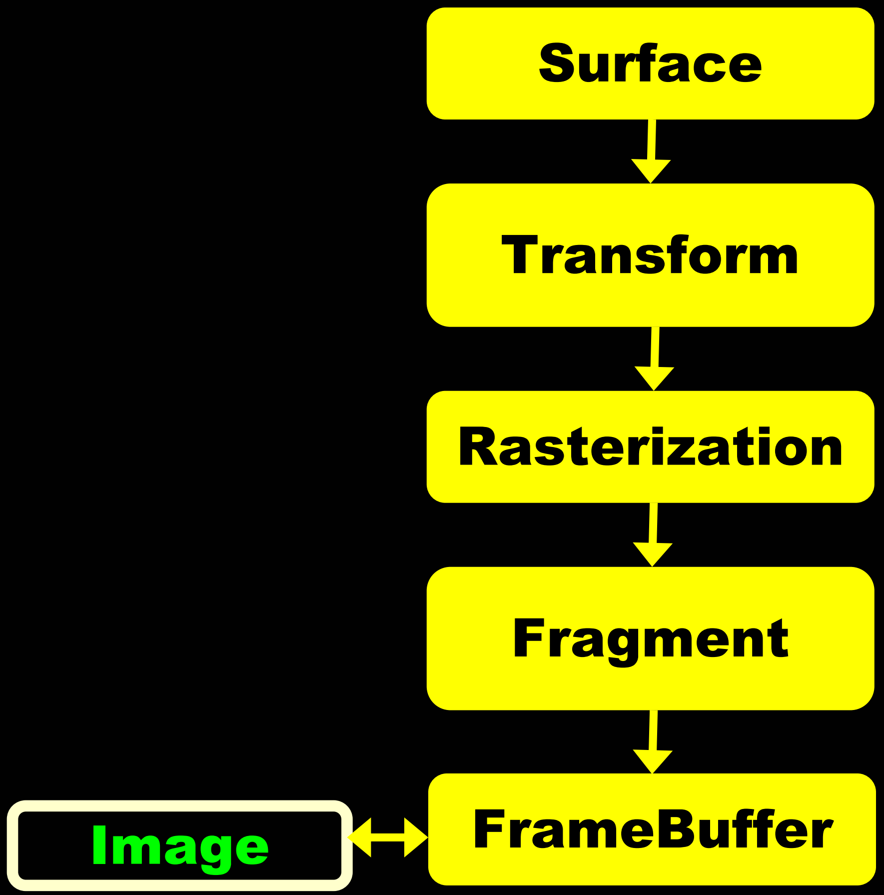
Graphics Hardware

- Input – 3D Scene
  - Typically triangles made up of vertices
  - 1D buffers of vertex data
  - Vertex and Fragment programs
  - Textures
    - Random memory access



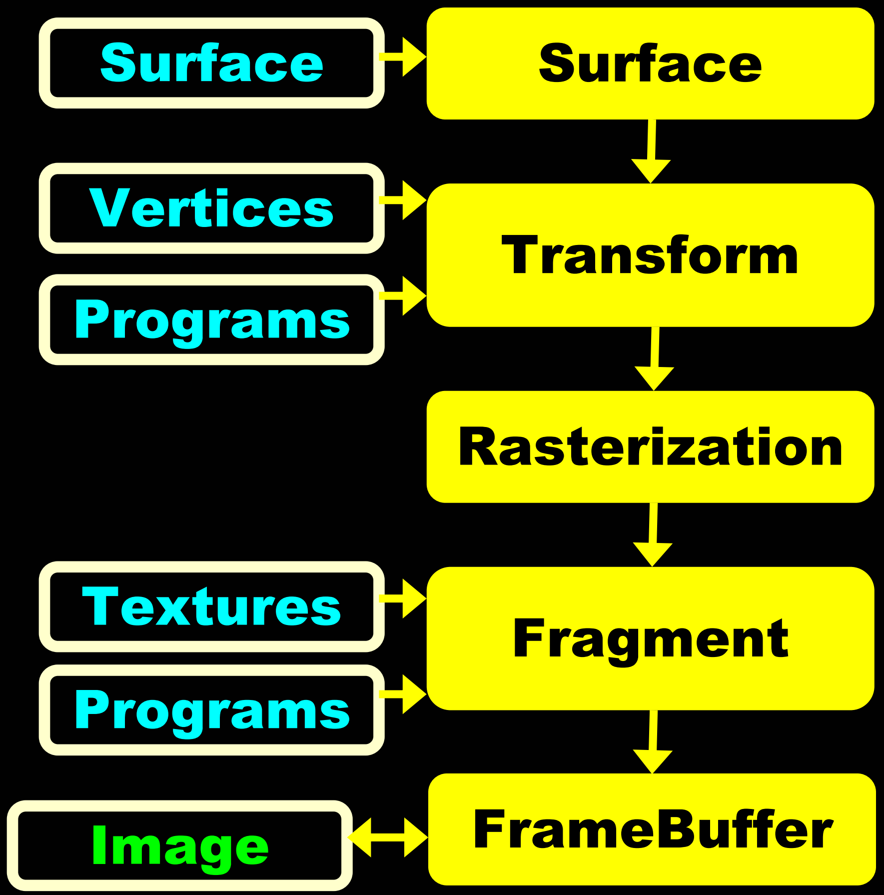
Graphics Hardware

- Input – 3D Scene
  - Typically triangles made up of vertices
  - 1D buffers of vertex data
  - Vertex and Fragment programs
  - Textures
    - Random memory access
- Output – 2D Image
  - Color, depth and stencil



Graphics Hardware

- Focus on programmable stages
  - Transform
  - Fragment

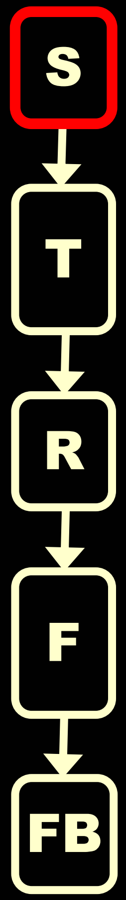


Graphics Hardware Evolution

Year	2000	2001	2002
ATI RADEON	7500 (R100)	8500 (R200)	9700 (R300)
DirectX	7	8	9
Stages			
Surface	CPU	CPU/GPU	CPU/GPU
Transform	State	VS 1.1 128 instrs	VS 2.0 256 instrs Control Flow
Rasterization	State	State	State
Fragment	State	12 tex/16 alu s3.12 Fixed point PS 1.4	32 tex/ 64 alu s16e7 Floating Point PS 2.0
FrameBuffer	State	State	State

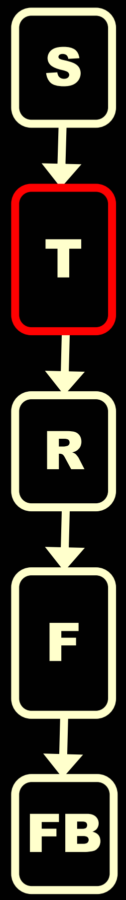
Surface Generation Stage

- Newest stage
  - DX8 NPatches
- State controlled
- Geometric calculations based on complex surfaces



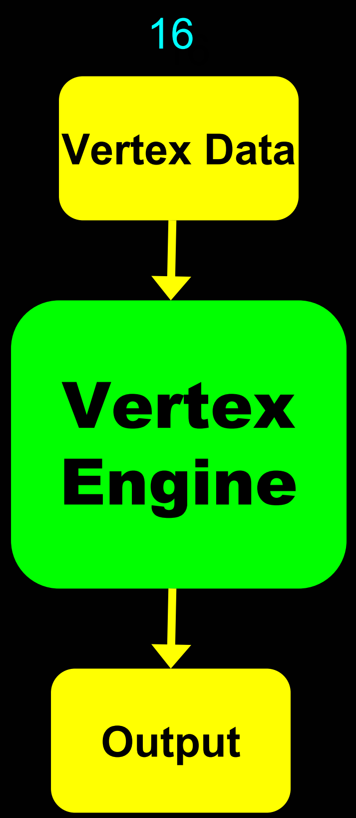
Transform Stage

- Includes:
  - ModelView Transformation
  - Vertex Lighting
  - Perspective Transformation
  - Tweening/Skinning
- Per-vertex operations
- Originally microcoded on DSPs or CPU
  - Controlled by fixed function state
- Programmable Vertex Engine
  - Lindholm et al., SIGGRAPH 2001



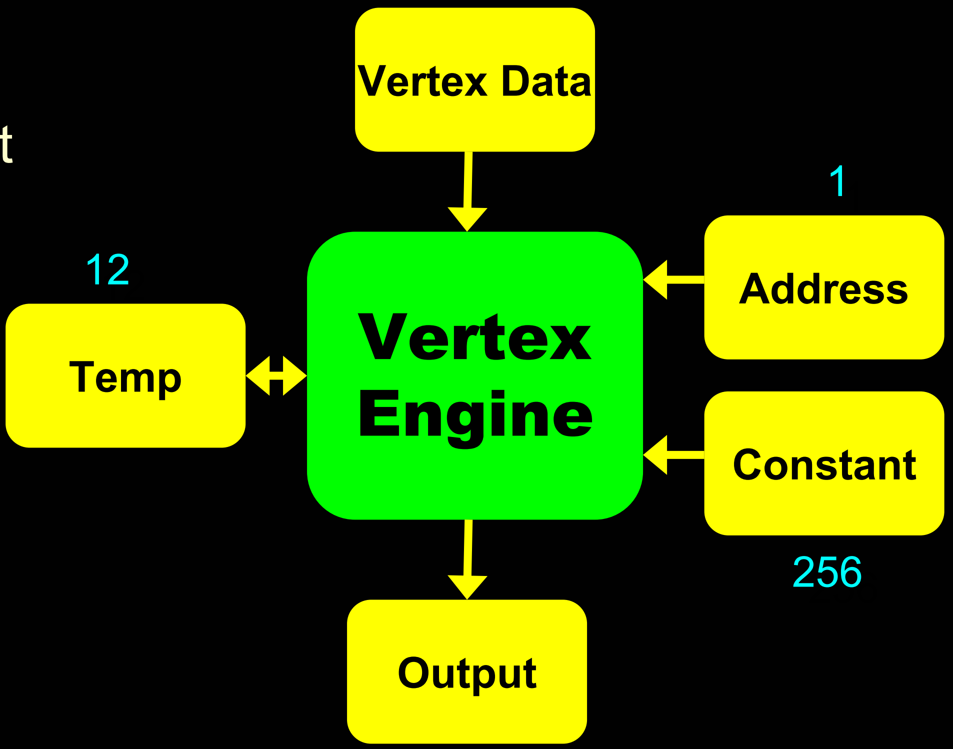
Vertex Engine

- 4 parallel vertex engines
- Input
  - Vertex stream
  - Constants
- Output
  - Position, Color, Tex Coords
- Program (Shader) has up to 256 instructions



Vertex Engine

- Registers
  - Four component floating point vectors
- 12 Read-write temp registers
- Output registers
  - **`position, color, fogcoord, pointsize, texcoord`**
- Vertex shader outputs are pixel shader inputs



Vertex Program Instructions

- Basic arithmetic operators
  - `ADD, MAD, MUL, SUB`
- Comparison
  - `MAX, MIN, SGE, SLT`
- Dot and cross Product
  - `DP3, DP4, DPH, XPD`
- Exponential functions
  - `EX2, EXP, LG2, LOG`
- Other
  - `ABS, ARL, DST, FLR, FRC, LIT, MOV, POW, RCP, RSQ, SWZ`

Register Modifiers

- Source swizzle selects which components to use
  - `iPosition.[xyzw][xyzw][xyzw][xyzw]`
  - e.g. `iPosition.yzxw`
- Destination mask to select individual component
  - `oColor.{x}{y}{z}{w}`
- Source negation
  - `-iNormal`

Simple Vertex Program

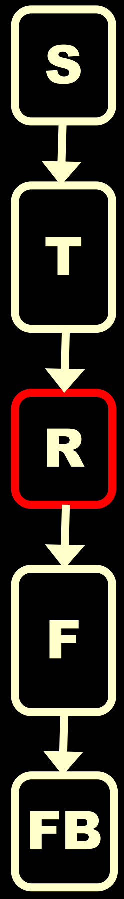
```
!!ARBvp1.0
ATTRIB iPos        = vertex.position;
PARAM  mvp[4]      = { state.matrix.mvp };
PARAM  ambientCol  = state.lightprod[0].ambient;
OUTPUT oPos        = result.position;
OUTPUT oColor      = result.color;

# Transform the vertex to clip coordinates.
DP4   oPos.x, mvp[0], iPos;
DP4   oPos.y, mvp[1], iPos;
DP4   oPos.z, mvp[2], iPos;
DP4   oPos.w, mvp[3], iPos;

# Write out a color.
MOV   oColor, ambientCol;
END
```

RADEON 9800 Vertex Shader

- DirectX 9.0 Vertex Shader 2.0
- Same arithmetic instructions
- Constant based control flow capabilities
- Loops, branches, subroutines
  - CALL, LOOP, ENDLOOP, JUMP, JNZ, LABEL, REPEAT, ENDREPEAT, RETURN
  - 16 Integer constants
  - 16 Boolean constants
  - Loop counter

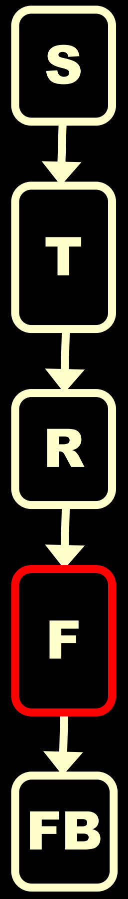
Rasterization Stage

- Includes:
  - Triangle Setup
  - Viewport Clipping
  - Viewport Transform
  - Rasterization
- Not programmable, some control through state
- High precision vertex parameter interpolators
  - Position, normal, color, tex coords



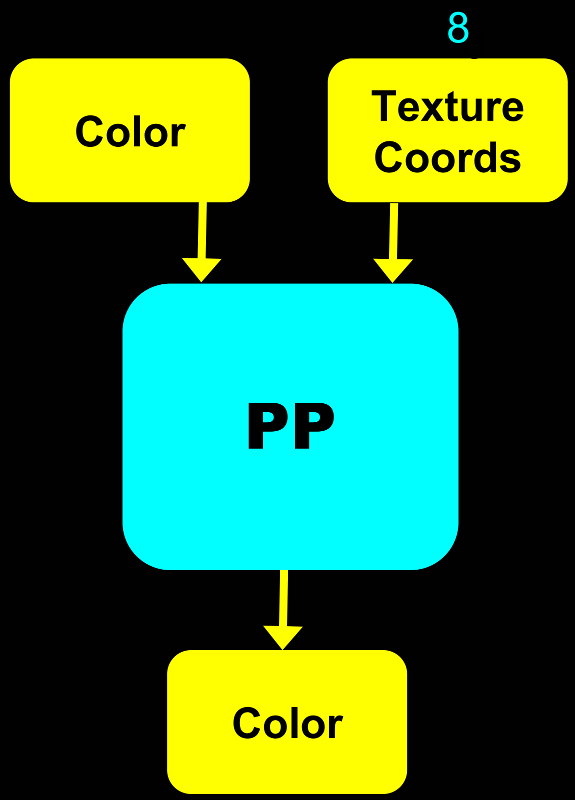
Fragment Stage

- Includes:
  - Texturing
    - arbitrary memory fetch (Gather)
    - fixed point filtering
      - Point, Linear, Bi-linear, Tri-linear, Anisotropic
  - Fragment (Per-Pixel) Lighting
- RADEON 9700 introduced floating point
  - 32 Texture and 64 ALU instructions
  - Co-issue vector and scalar instruction



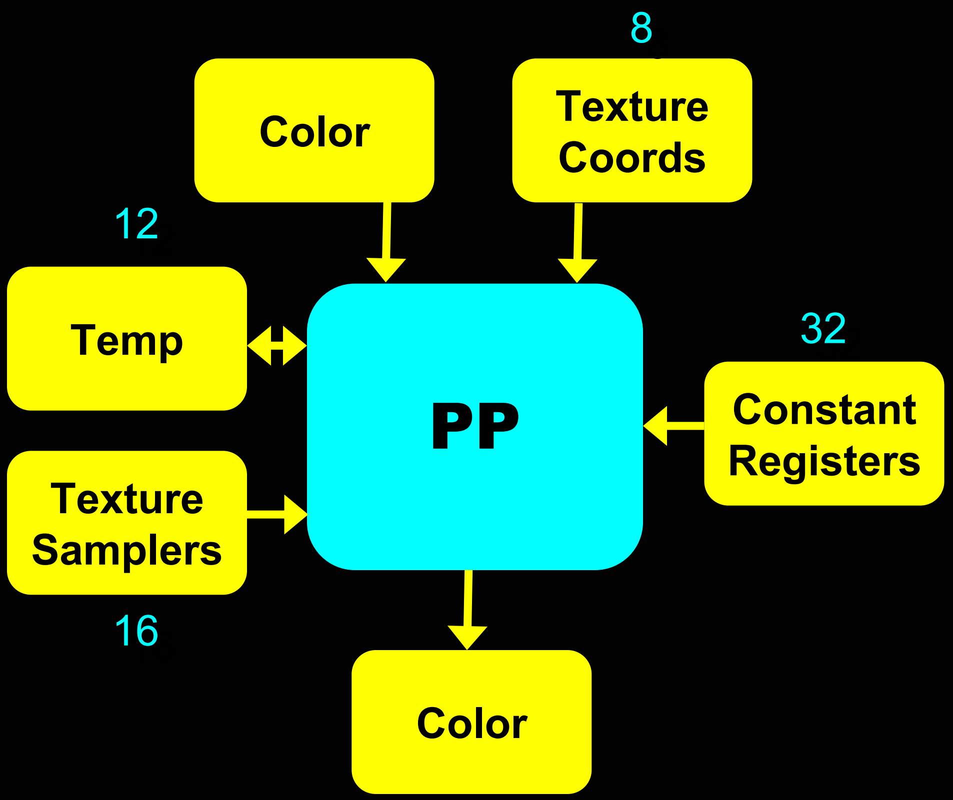
Pixel Pipeline

- 8 parallel floating point pixel pipelines
- Input
  - Color, Texcoords
- Output
  - Color, Depth
- Output surfaces
  - 16, 32 bit float
  - 16 bit fixed



Pixel Pipeline

- Registers
  - Four component
  - 24bit floating point
    - performance and precision
- 12 read-write temp registers
- 16 Texture images
- Texture instructions
  - **`KIL, TEX, TXB, TXP`**



Simple Fragment Program

```
!!ARBfp1.0
TEMP   temp;
ATTRIB tex0 = fragment.texcoord[0]; # input register
ATTRIB col0 = fragment.color;
PARAM  half = 0.5, 0.5, 0.5, 0.5;   # constant

OUTPUT out  = result.color;

#Fetch texture
TEX temp, tex0, texture[0], 2D;
#modulate and write out color
MUL out, col0, temp;
END
```

Computing the Mandelbrot set

- Test each point on the complex number plane
  - X dimension is the real component
  - Y dimension is the imaginary component
  - $Z' = Z^2 + C$
  - C = starting position on complex plane
  - Iterate until Z > 2

Mandelbrot main loop

```
MUL pos.xy, curr, curr;              # real component
ADD pos.x, pos.x, -pos.y;            # x² - y² + start.x
ADD pos.x, pos.x, start.x;
MUL pos.y, curr.x, curr.y;           # imaginary component
MAD pos.y, pos.y, two.x, start.y;
DP3 magnitude, pos, pos;             # calculate magnitude
SUB magnitude, magnitude, four;      # compare magnitude to 4
CMP escape.x, magnitude, 0, 1;
ADD pos.z, pos.z, escape.x;
MOV curr, pos;                       # ready next iteration
```

Mandelbrot main loop

```
MUL pos.xy, curr, curr;            # real component
ADD pos.x, pos.x, -pos.y;          # x^2 - y^2 + start.x
ADD pos.x, pos.x, start.x;
MUL pos.y, curr.x, curr.y;         # imaginary component
MAD pos.y, pos.y, two.x, start.y;
DP3 magnitude, pos, pos;           # calculate magnitude
```

Mandelbrot main loop

```
MUL pos.xy, curr, curr;          #
  real component
ADD pos.x, pos.x, -pos.y;        # x²
  - y² + start.x
ADD pos.x, pos.x, start.x;
MUL pos.y, curr.x, curr.y;       #
  imaginary component
MAD pos.y, pos.y, two.x, start.y;
DP3 magnitude, pos, pos;         #
  calculate magnitude
```

Mandelbrot main loop

```
MUL pos.xy, curr, curr;            # real component
ADD pos.x, pos.x, -pos.y;          # x^2 - y^2 + start.x
ADD pos.x, pos.x, start.x;
MUL pos.y, curr.x, curr.y;         # imaginary component
MAD pos.y, pos.y, two.x, start.y;
DP3 magnitude, pos, pos;           # calculate magnitude
```

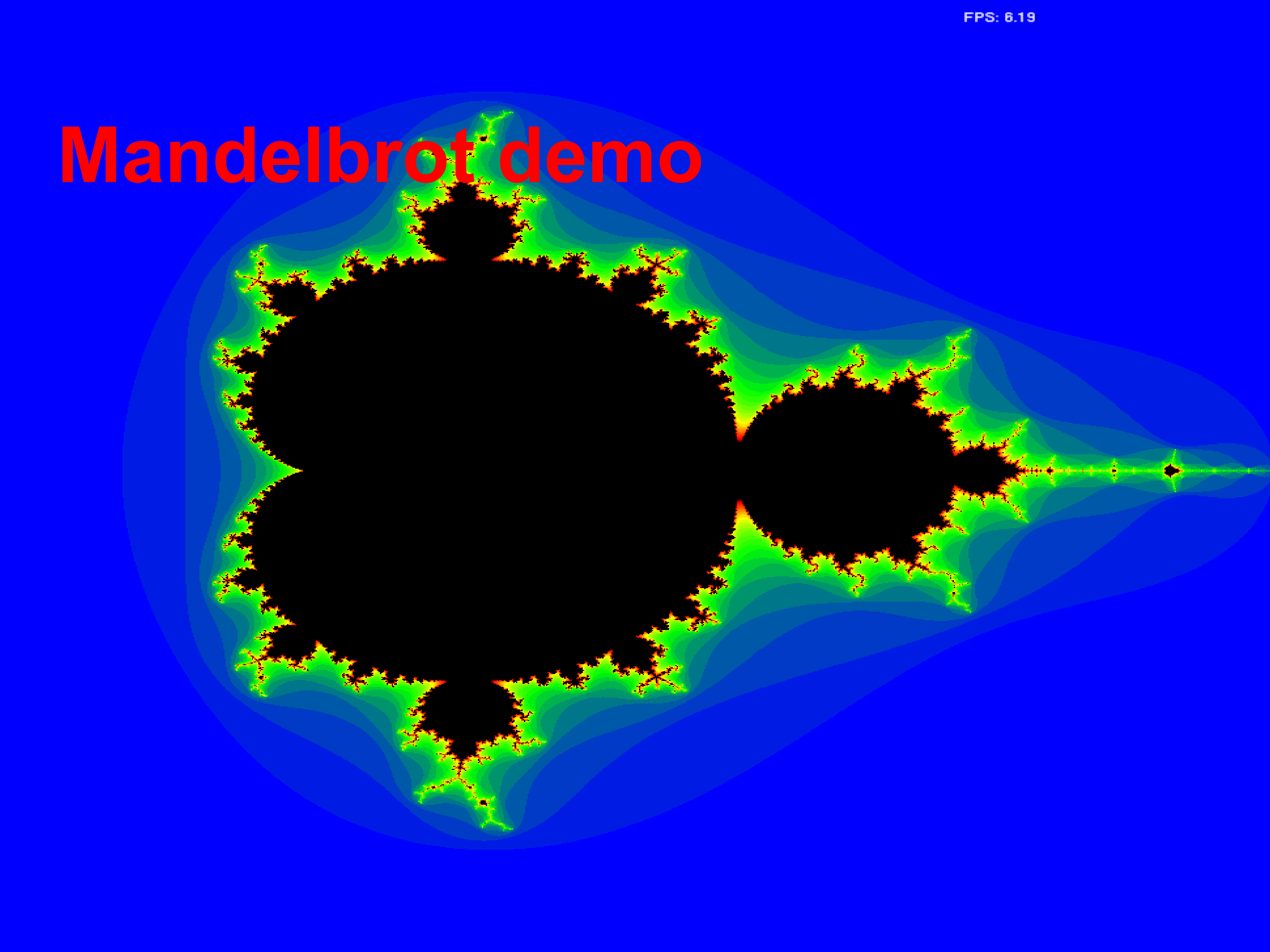
Mandelbrot main loop

```
MUL pos.xy, curr, curr;              #
  real component
ADD pos.x, pos.x, -pos.y;            # x^2
  - y^2 + start.x
ADD pos.x, pos.x, start.x;
MUL pos.y, curr.x, curr.y;           #
  imaginary component
MAD pos.y, pos.y, two.x, start.y;
DP3 magnitude, pos, pos;             #
  calculate magnitude
```

Mandelbrot multipass demo

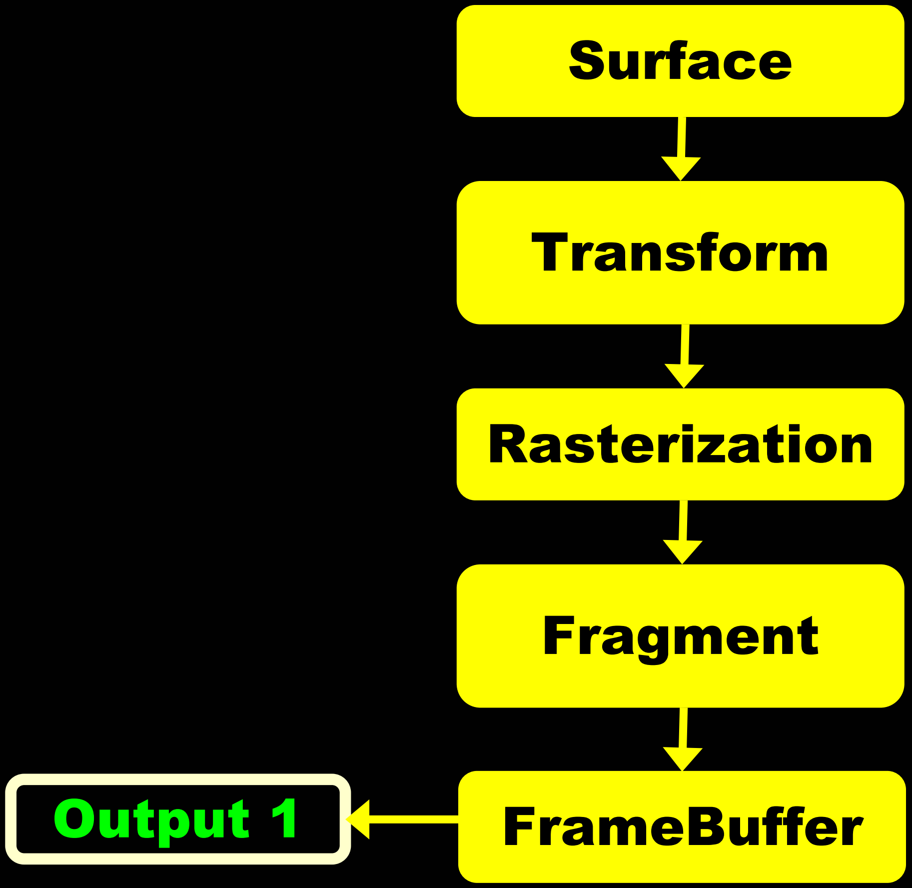
- Main loop is 11 instructions
- Need multiple iterations to get detail
- Multipass
  - Render to texture
  - Pass 1: Render output to framebuffer
  - Set framebuffer data as texture
  - Pass 2 to N: Read in previous result as texture
  - Pass N+1: Draw texture on screen

Mandelbrot demo



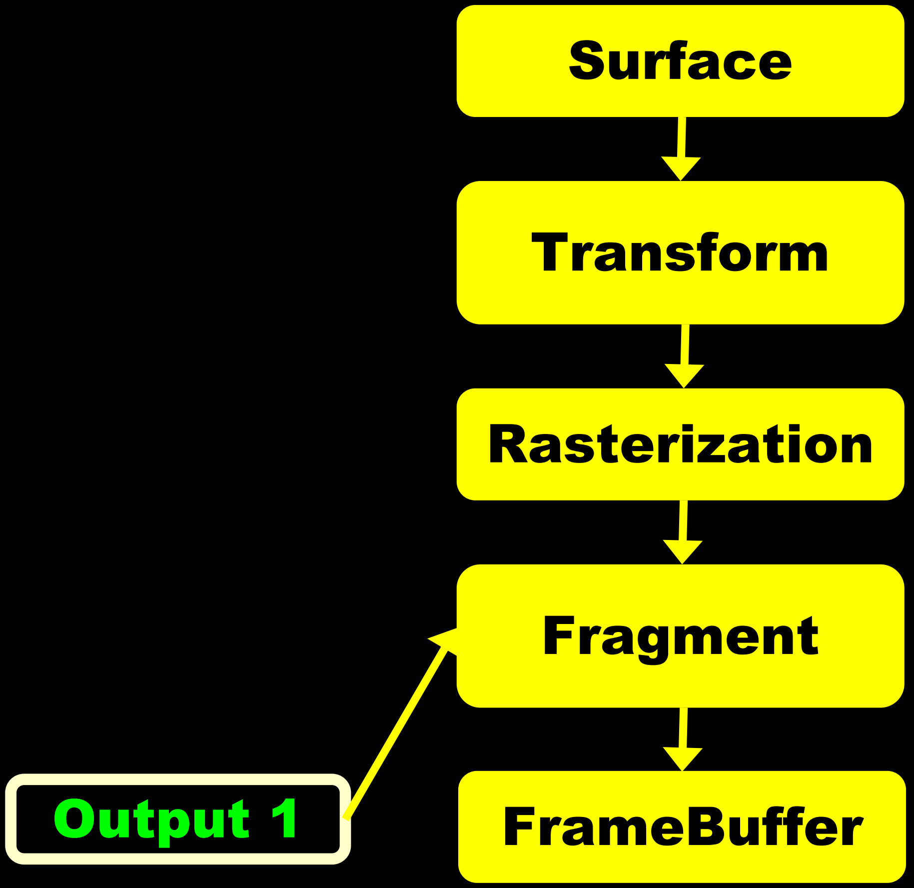
Render to texture

- Render data



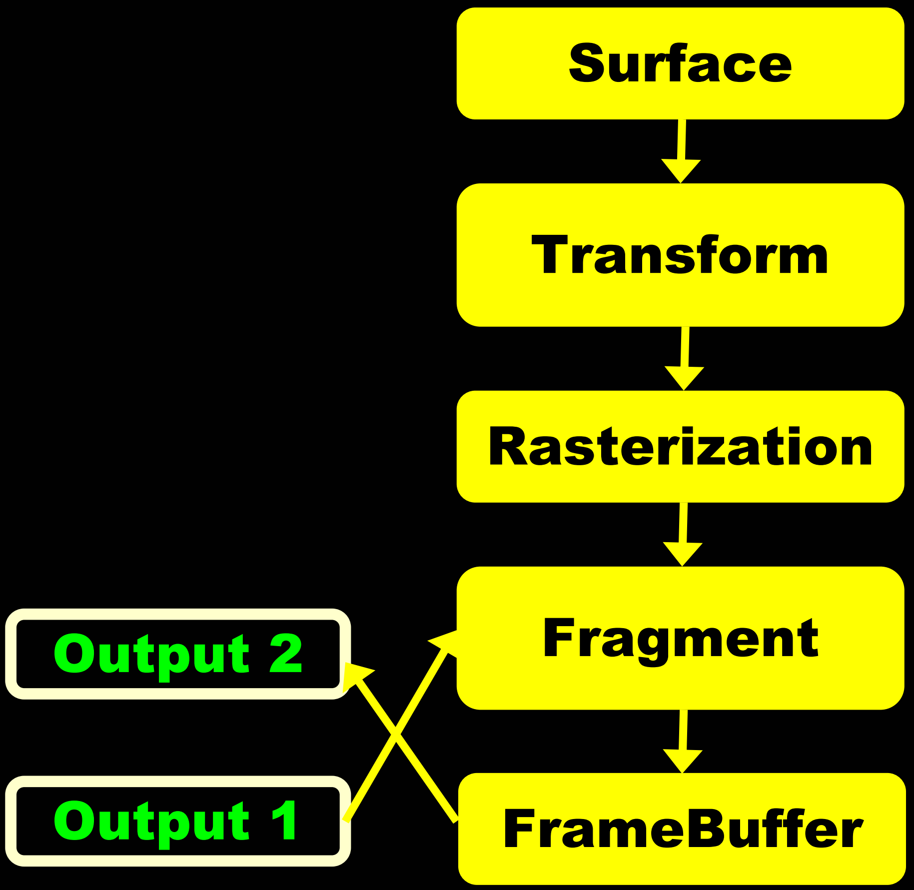
Render to texture

- Render data
- Set data as texture input to Fragment stage



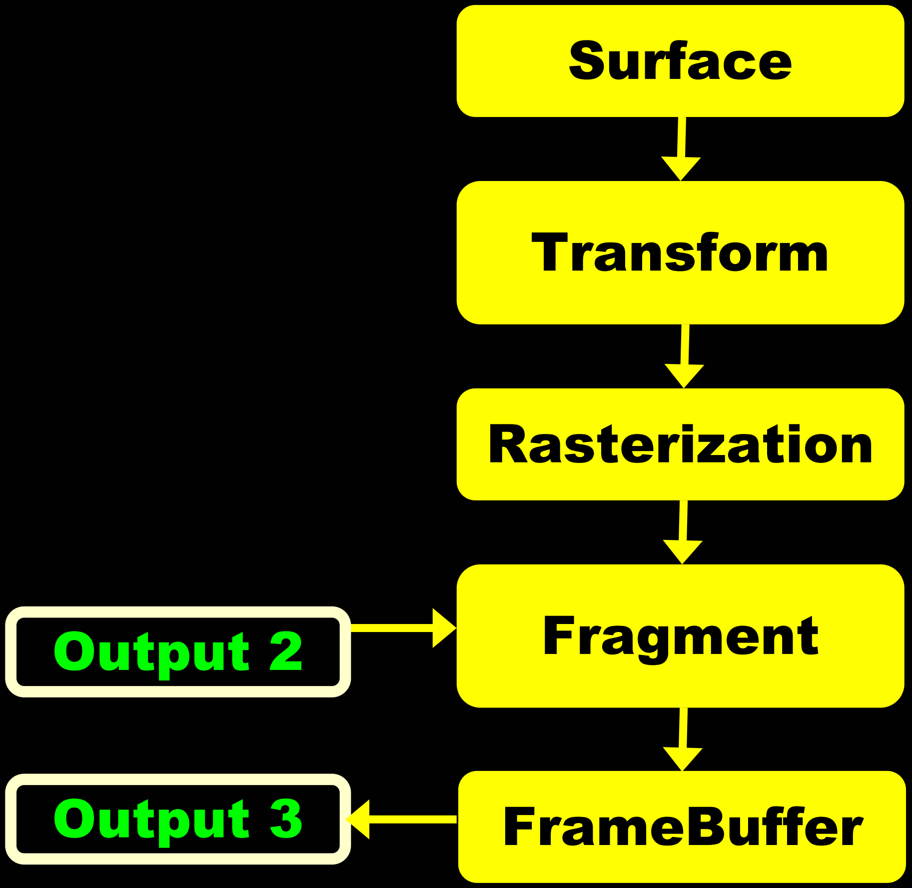
Render to texture

- Render data
- Set data as texture input to Fragment stage
- Render to second output



Render to texture

- Render data
- Set data as texture input to Fragment stage
- Render to second output
- Switch output and texture buffer
- Ping-pong buffers



Fast Fourier Transform

- Transform image into frequency domain
  - Complex weights for a summation of sine waves
- Separable Filter
- Cooley and Tukey 65, *decimation in time*
  - $\log_2$ (width) + $\log_2$ (height) + 2 rendering passes
  - Ping-pong between two floating point renderable textures

Fast Fourier Transform

- Horizontal scramble
  - dependent texture read using precalculated position bit invert
- $\log_2$(width) butterfly passes
  - texture read offset, beta
- Vertical scramble
- $\log_2$(height) butterfly passes
- More details
  - ShaderX2 chapter, “Advanced Image Processing with DirectX 9 Pixel Shaders”, Mitchell, Ansari, Hart
  - RNL seperable 50x50 gaussian blur filter
    - Jason Mitchell’s course notes from course 7. Real-Time Shading

FFT demo

Multiple Render Targets

- Multiple render targets
  - 4 possible output buffers
  - All same X, Y and bit resolution
  - Different formats
- ATI_draw_buffers extension
  - adds **`result.color[n]`** where n=0,1,2,3
    - `OUTPUT oColor1 = result.color[1];`

Render to texture

- pbuffers
- OpenGL ARB SuperBuffers
  - Adds **`GLmem`** type
    - can be used as
      - framebuffer
      - texture
      - vertex array

Displacement mapping using render to vertex array

- 2 Pass algorithm
  - 1st pass
    - Displacement fragment program
    - Output to SuperBuffer Vertex Array
      - using ATI_draw_buffers
      - Vertex, Normal, TexCoord
  - 2nd pass
    - Vertex program uses displaced Vertex Buffer
    - Fragment program does bump mapped lighting

Displacement Mapping Demo

Programming Options

- Low or High level
  - Assembler or High Level Shading languages
- Assembler
  - OpenGL Vertex Program and Fragment Program
  - DirectX Vertex Shader and Pixel Shader
- Shading Languages
  - the OpenGL Shading Language
  - DirectX9 HLSL
  - NVIDIA's Cg
- Tools
  - ATI's RenderMonkey

the OpenGL Shading Language

- C like language for programming GPUs
- Basically the same language for vertex and fragment shaders
- Inputs and outputs similar to Vertex Engine and Pixel Pipeline
  - different syntax e.g. **`gl_TexCoord0, gl_FragColor`**
- Type qualifiers
  - **`const      // compile-time constant`**
  - **`uniform    // constant across primitive`**
- Types
  - **`float, int, bool, vec4, ivec3`**

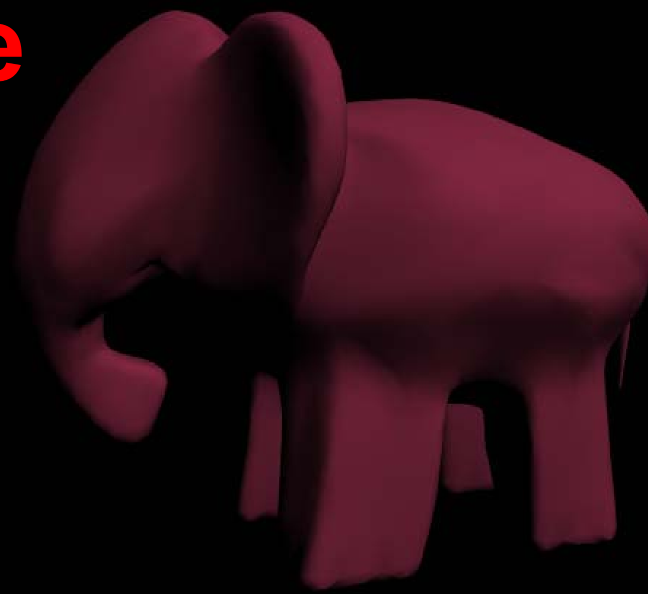
the OpenGL Shading Language

- Structures and arrays
- Operators work with 1-4 component types
- Flow control
- User defined functions
- Built-in functions
  - sin, cos, normalize, noise,

Simple GLSL example vertex shader

```
vec4  LightPos     = vec4 ( 0.4, 0.6, 0.6, 0.0 );
vec4  DiffuseColor = vec4( 0.50, 0.15, 0.25, 1.0 );
vec4  LightColor   = vec4( 0.8, 0.5, 0.8, 1.0 );

void main(void)
{
   vec3 eyeNormal;

   gl_Position  = gl_ModelViewProjectionMatrix * gl_Vertex;
   eyeNormal    = gl_NormalMatrix * gl_Normal;

   gl_TexCoord0 = vec4( eyeNormal, 0.0 );        // eye space normal
   gl_TexCoord1 = LightPos;
   gl_TexCoord2 = DiffuseColor;
   gl_TexCoord3 = LightColor;
}
```

Simple GLSL example fragment shader



```
const float Shininess = 50.0;
void main (void)
{
    vec3  NNormal;
    vec4  MyColor;
    float Intensity;

    NNormal   = normalize( vec3 ( gl_TexCoord0 ) );       // tc0 = eyeNormal
    Intensity = dot ( vec3 ( gl_TexCoord1 ), NNormal );   // diffuse calculation
                                                          // tc1 = light position
    Intensity = max ( Intensity, 0.0 );
    MyColor   = vec4( Intensity ) * gl_TexCoord2;         // tc2 = diffuse color
    NNormal   = abs ( NNormal );
    Intensity = dot ( NNormal, vec3( gl_TexCoord1 ) );    // light position
    Intensity = max ( Intensity, 0.0 );
    Intensity = pow ( Intensity, Shininess );
    MyColor  += vec4( Intensity ) * gl_TexCoord3;         // specular light color
    MyColor.a = gl_TexCoord2.a;                           // diffuse alpha
    gl_FragColor = MyColor;
}
```

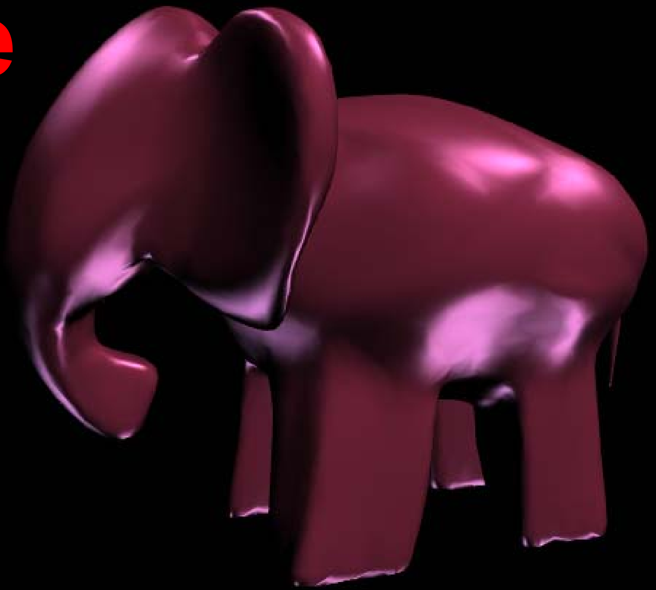
Simple GLSL example fragment shader

```
const float Shininess = 50.0;
void main (void)
{
    vec3  NNormal;
    vec4  MyColor;
    float Intensity;

    NNormal   = normalize( vec3 ( gl_TexCoord0 ) );      // tc0 = eyeNormal
    Intensity = dot ( vec3 ( gl_TexCoord1 ), NNormal ); // diffuse calculation
                                                         // tc1 = light position
    Intensity = max ( Intensity, 0.0 );
    MyColor   = vec4( Intensity ) * gl_TexCoord2;        // tc2 = diffuse color
    NNormal   = abs ( NNormal );
    Intensity = dot ( NNormal, vec3( gl_TexCoord1 ) );  // light position
    Intensity = max ( Intensity, 0.0 );
    Intensity = pow ( Intensity, Shininess );
    MyColor  += vec4( Intensity ) * gl_TexCoord3;        // specular light color
    MyColor.a = gl_TexCoord2.a;                          // diffuse alpha
    gl_FragColor = MyColor;
}
```

Simple GLSL example fragment shader

```
const float Shininess = 50.0;
void main (void)
{
    vec3  NNormal;
    vec4  MyColor;
    float Intensity;

    NNormal   = normalize( vec3 ( gl_TexCoord0 ) );      // tc0 = eyeNormal
    Intensity = dot ( vec3 ( gl_TexCoord1 ), NNormal );  // diffuse calculation
                                                          // tc1 = light position
    Intensity = max ( Intensity, 0.0 );
    MyColor   = vec4( Intensity ) * gl_TexCoord2;        // tc2 = diffuse color
    NNormal   = abs ( NNormal );
    Intensity = dot ( NNormal, vec3( gl_TexCoord1 ) );   // light position
    Intensity = max ( Intensity, 0.0 );
    Intensity = pow ( Intensity, Shininess );
    MyColor  += vec4( Intensity ) * gl_TexCoord3;        // specular light color
    MyColor.a = gl_TexCoord2.a;                          // diffuse alpha
    gl_FragColor = MyColor;
}
```

Simple GLSL example fragment shader



```
const float Shininess = 50.0;
void main (void)
{
    vec3  NNormal;
    vec4  MyColor;
    float Intensity;

    NNormal   = normalize( vec3 ( gl_TexCoord0 ) );        // tc0 = eyeNormal
    Intensity = dot ( vec3 ( gl_TexCoord1 ), NNormal );    // diffuse calculation
                                                           // tc1 = light position
    Intensity = max ( Intensity, 0.0 );
    MyColor   = vec4( Intensity ) * gl_TexCoord2;          // tc2 = diffuse color
    NNormal   = abs ( NNormal );
    Intensity = dot ( NNormal, vec3( gl_TexCoord1 ) );     // light position
    Intensity = max ( Intensity, 0.0 );
    Intensity = pow ( Intensity, Shininess );
    MyColor  += vec4( Intensity ) * gl_TexCoord3;          // specular light color
    MyColor.a = gl_TexCoord2.a;                            // diffuse alpha
    gl_FragColor = MyColor;
}
```

Computation on GPUs

- *Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware*
  - Hoff et al. 99
  - use polygon scan-conversion and depth comparison
- *Interactive Multi-Pass Programmable Shading*
  - Peercy00
  - Treat the OpenGL architecture as a general SIMD computer
- *Physically-Based Visual Simulation on Graphics Hardware*
  - Mark Harris et. al., GH02
  - Coupled map lattice stored in a texture
    - continuous values on a discrete lattice
  - simulations of convection, reaction-diffusion, and boiling

Computation on GPUs

- Simulation and Computation, GH03 session
  - *Simulation of Cloud Dynamics on Graphics Hardware*
    - Harris et al.
  - *A Multigrid Solver for Boundary Value Problems Using Programmable Graphics Hardware*
    - Goodnight et al.
  - *The FFT on a GPU*
    - Moreland et al.
- GPUs as Stream Processors, GH03 panel
  - *Data Parallel Computing on Graphics Hardware*, Ian Buck
  - "Brook" stream API
    - objective is to abstract computational functionality

Computation on GPUs

- SIGGRAPH2003 session
  - Thursday, 3.45-5.30
  - Bolz et. al., Sparse matrix solvers
  - Krüger et. al., Krüger, Westermann
    - *Linear Algebra Operators for GPU Implementation of Numerical Algorithms*
    - Framework for linear algebra operators
    - Navier-Stokes Demo

Summary

- GPU is
  - highly parallel
  - very programmable
    - low and high levels
  - increasing in performance and maintaining a low cost
    - driven by a demanding consumer application

Questions ?

- More information
  - www.ati.com/developer
  - devrel@ati.com
  - Course 22. The OpenGL Shading Language
    - Monday, Half Day, 1:45 - 5:30 pm
  - SuperBuffers and other GL extensions
    - Tuesday, exhibit hall C, 1 - 3pm.
  - www.opengl.org
- Acknowledgements
  - Evan Hart, Marwan Ansari, Bill Licea-Kane, Arcot Preetham, James Peercy