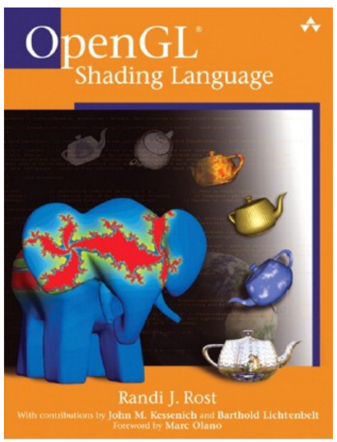
Introduction to GPU Programming with GLSL



Ricardo Marroquim

André Maximo

ISTI-CNR, Pisa

October, 2009
SIBGRAPI

summary

tutorial

Part I

Introduction and motivation (RM)

GPU architecture and pipeline (AM)

GLSL language :

- Hello World (AM)
- Basic Types (AM)
- Data Flow (RM)

OpenGL – GLSL integration (AM)

Part II

Examples:

- Cartoon Effect (AM)
- Texture Mapping (RM)
- Environment Mapping (RM)
- Phong Shading (AM)
- Spike Effect (AM)

GPGPU (RM)

- Particle System (RM)

Wrap-up (RM)

a word about parallelism

new reality

Commodity computers are now high performance machines

Computers don't increase in speed anymore, they just get wider

Parallel computing should be taught in 1st / 2nd semester of Computer Science and Engineering graduations

For example:

Merge Sort x Heap Sort

Both are O(n log n)

One parallel-friendly, one not

Students need to understand this early!

introduction

what is GPU?

Graphics Processing Unit

Graphics Processing Unit

Specialized hardware for graphics

Free CPU from the burden of rendering

Used everywhere:

- desktop, notebooks, mobiles, embedded system, etc.

introduction

why GPU programming?

Fixed functionality **limits** the programmer

Deep **changes** in graphics hardware

Outstanding **effects** are possible

Control over the processing inside the GPU

introduction

GPU history

90's: GPUs were black boxes (fixed functionalities)

2002: basic programmability

2004: GLSL (OpenGL 2.0), vertex and fragment shader

2006: geometry shader

2010?: tesselation shader (OpenGL 3.0)

motivation

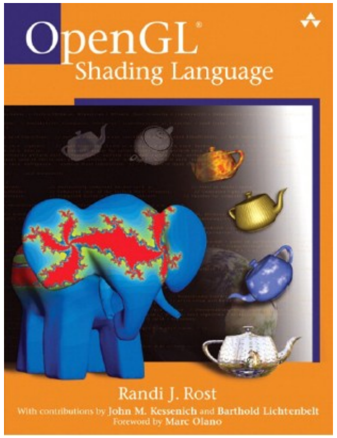
what is GLSL? OpenGL Shading Language

C/C++ like language

Coding short programs called shaders to run on the GPU

Used for different graphics card functionalities

High level shader language



(orange book)

motivation

why GLSL?

Based on OpenGL

OpenGL is the foundation of professional graphics

GLSL was introduced with OpenGL 2.0 in 2004

OpenGL 2.0 is the foundation of programmable, cross-platform and professional graphics

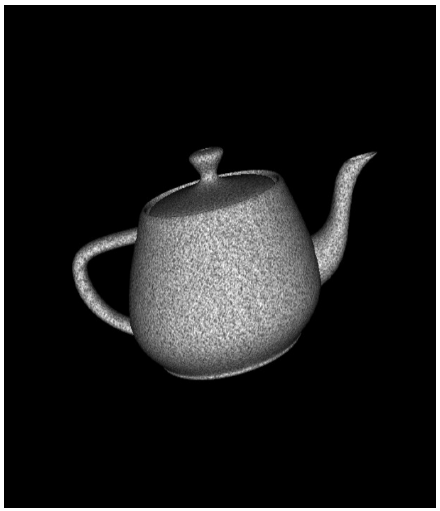
OpenGL / OpenGL 2.0 is an open standard

(open standard)

(+GLSL)

motivation

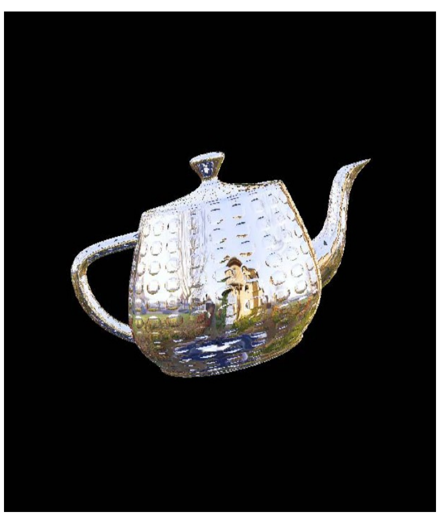
interesting new effects using shaders



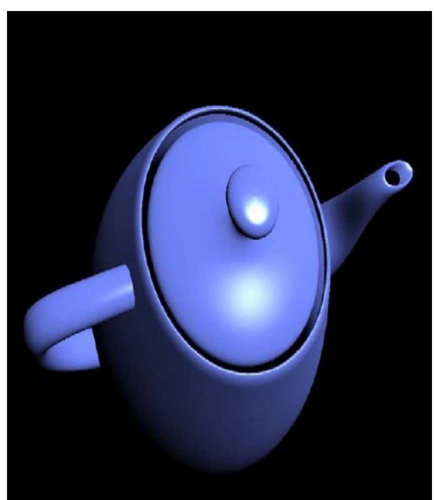
realistic materials

natural phenomena

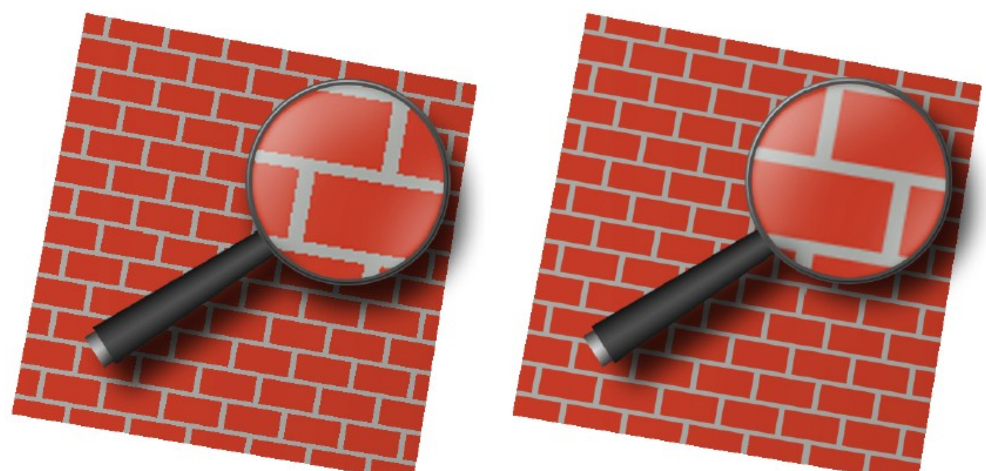
non-photorealistic rendering



environment map



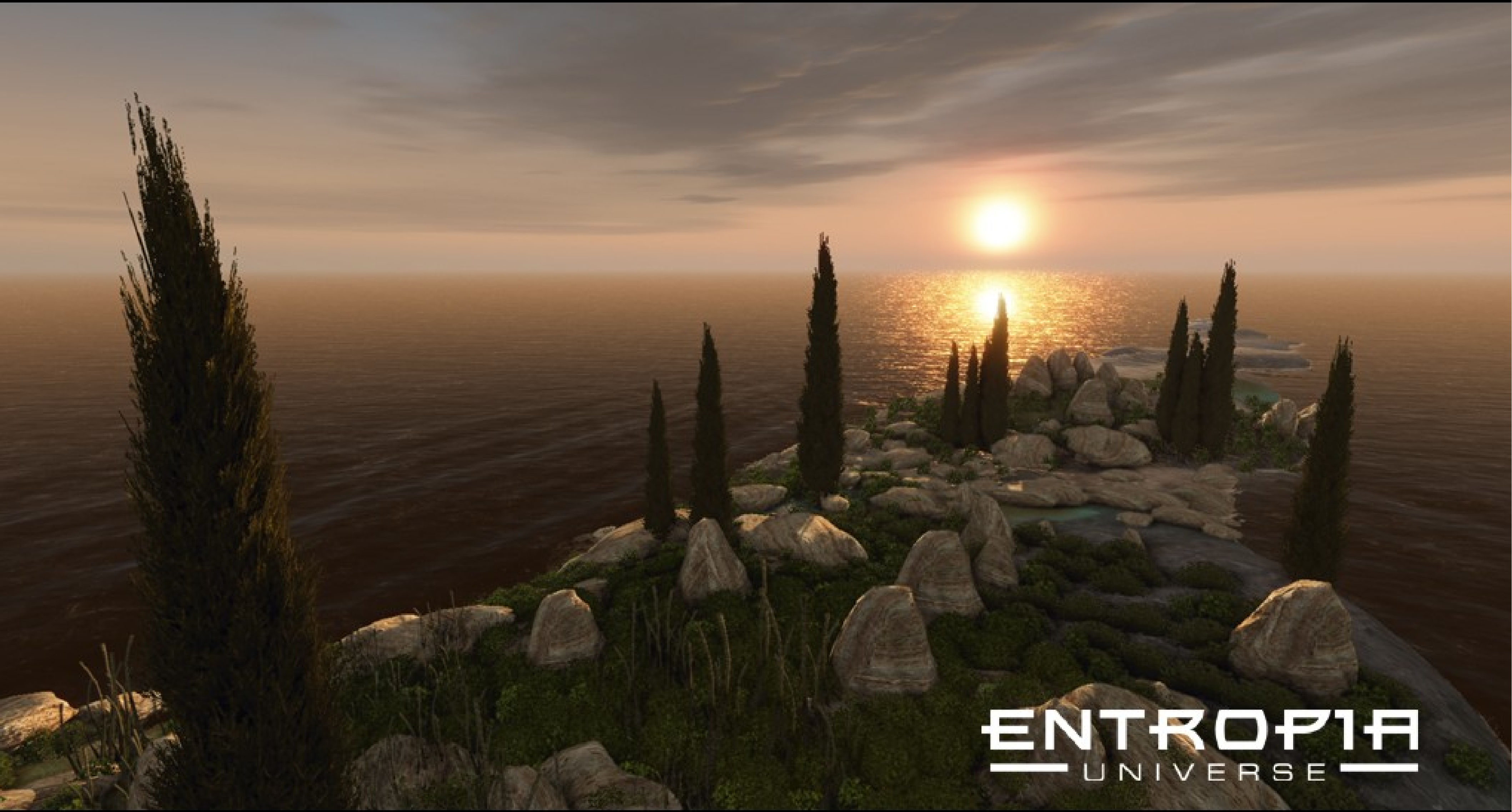
realistic surface



better anti-aliasing


CRYSIS
CRYTEK®
EA™


FARCRY 2


ENTROPIA
UNIVERSE

architecture

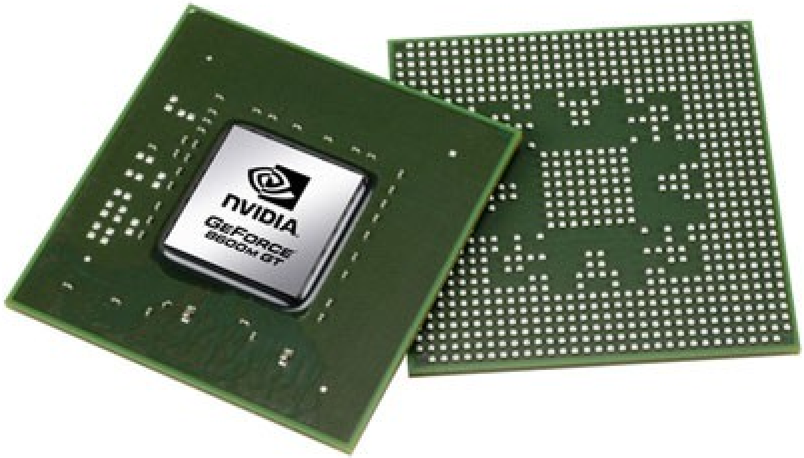
the GPU

High performance

High bandwidth

Arithmetic intensity

Throughput computing



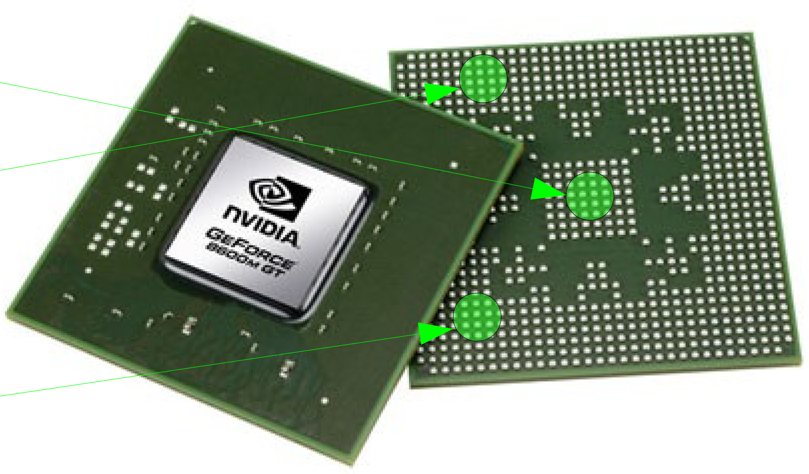
architecture

GPU x CPU

min(latency) – CPU (many levels of cache)

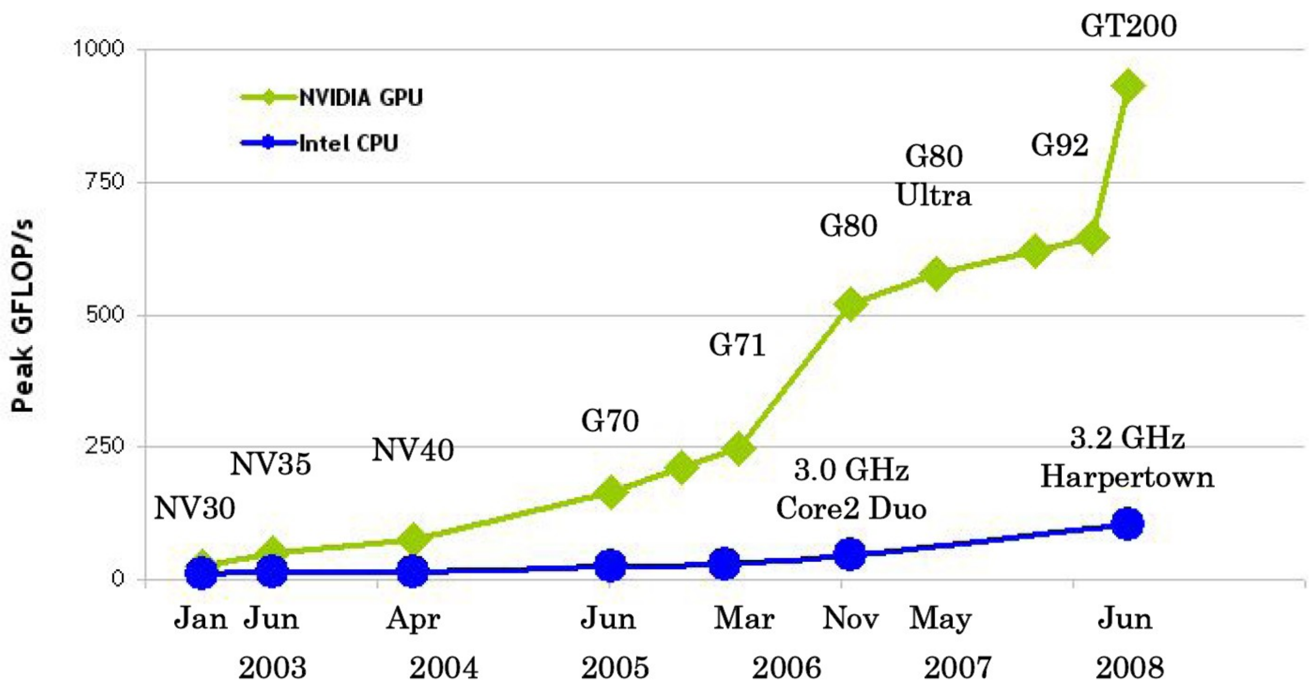
max(throughput) – GPU (stream programming model)

	GPU	CPU
control	↓	↑
data path	↑	↓
storage	↓	↑



architecture

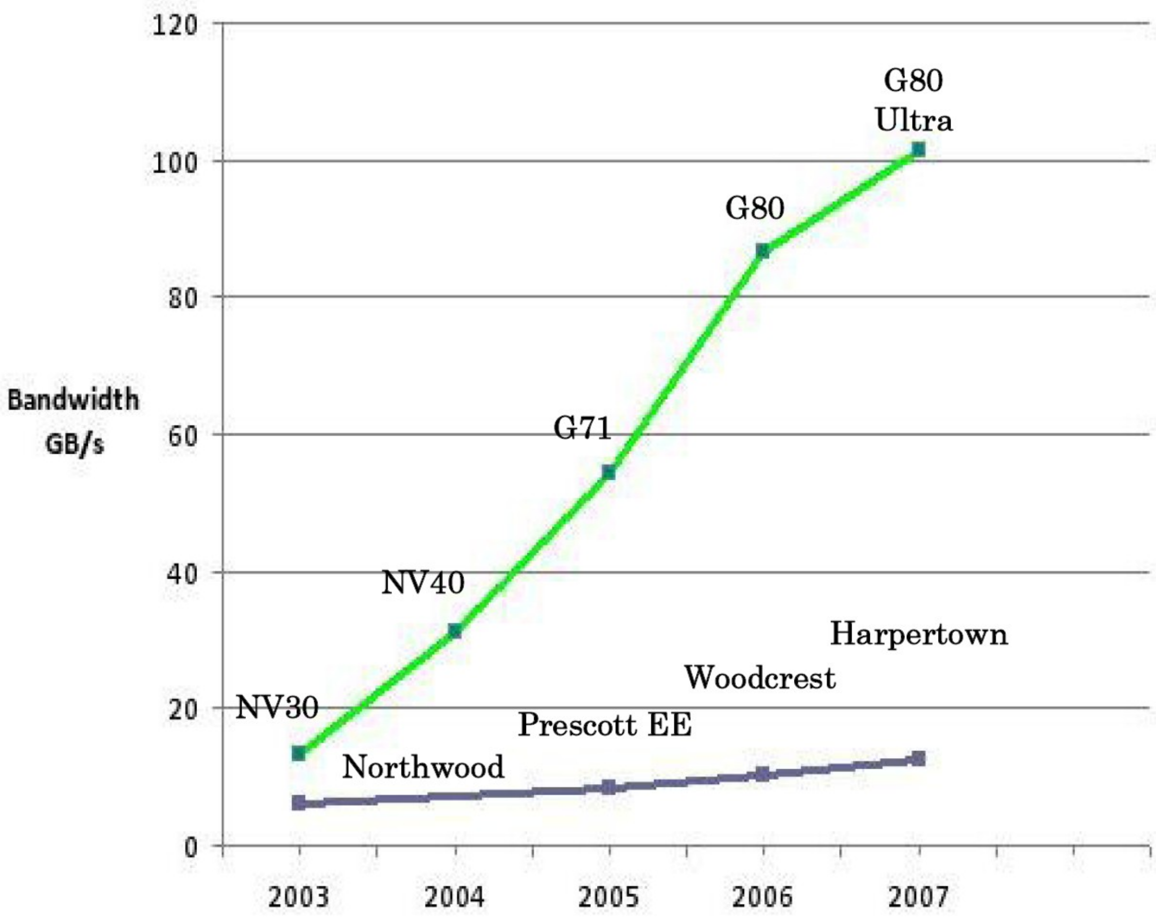
GPU x CPU – performance comparison



GT200 = GeForce GTX 280	G71 = GeForce 7900 GTX	NV35 = GeForce FX 5950 Ultra
G92 = GeForce 9800 GTX	G70 = GeForce 7800 GTX	NV30 = GeForce FX 5800
G80 = GeForce 8800 GTX	NV40 = GeForce 6800 Ultra	

architecture

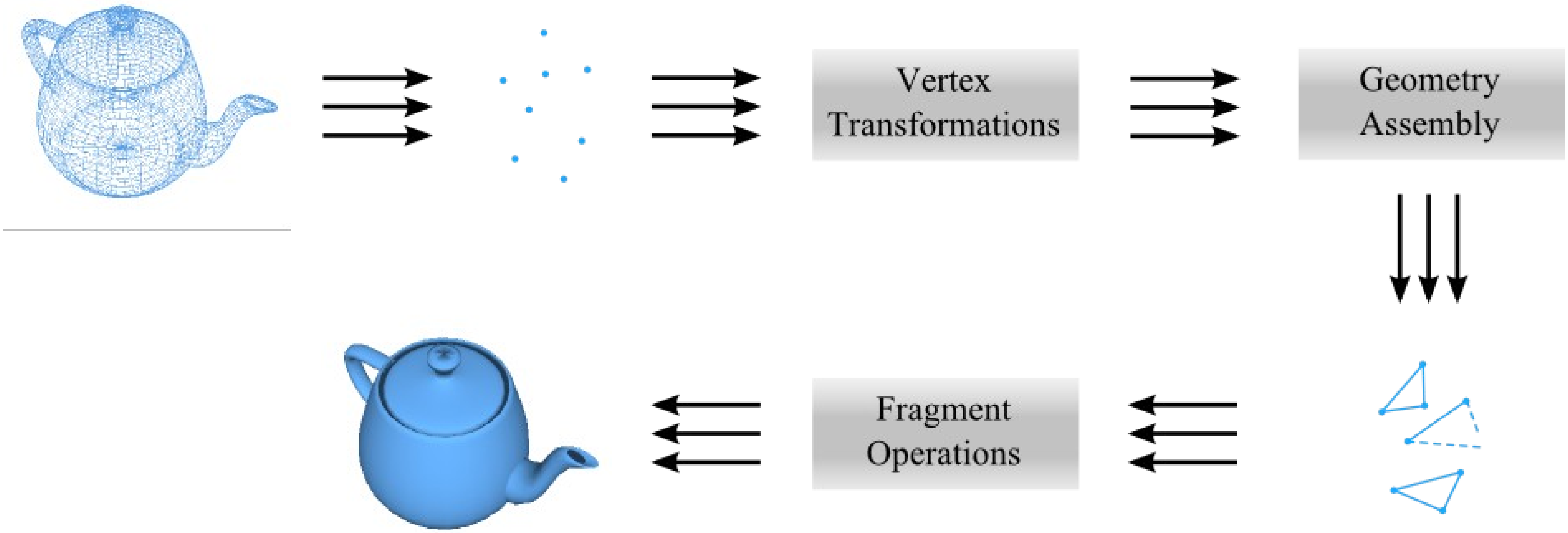
GPU x CPU – memory bandwidth comparison



architecture

OpenGL pipeline

Basic concepts

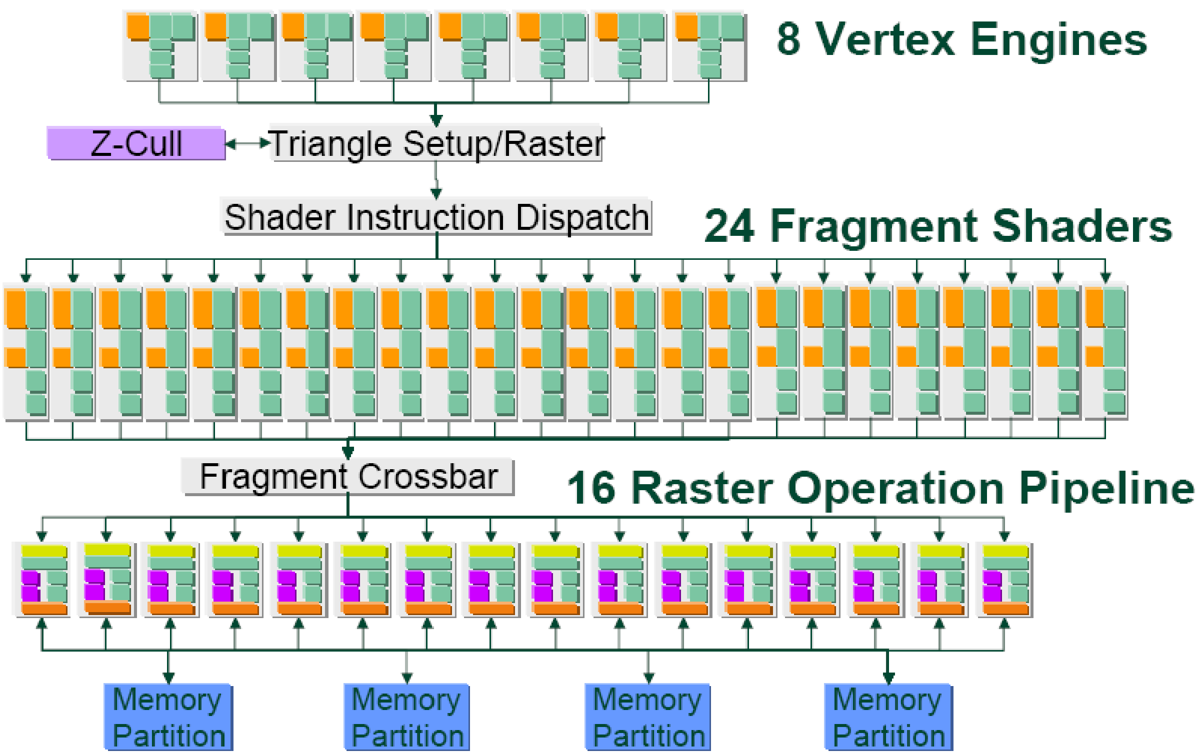


architecture

GPU pipeline – GeForce 7

One-way pipeline architecture

Different number of shader processors



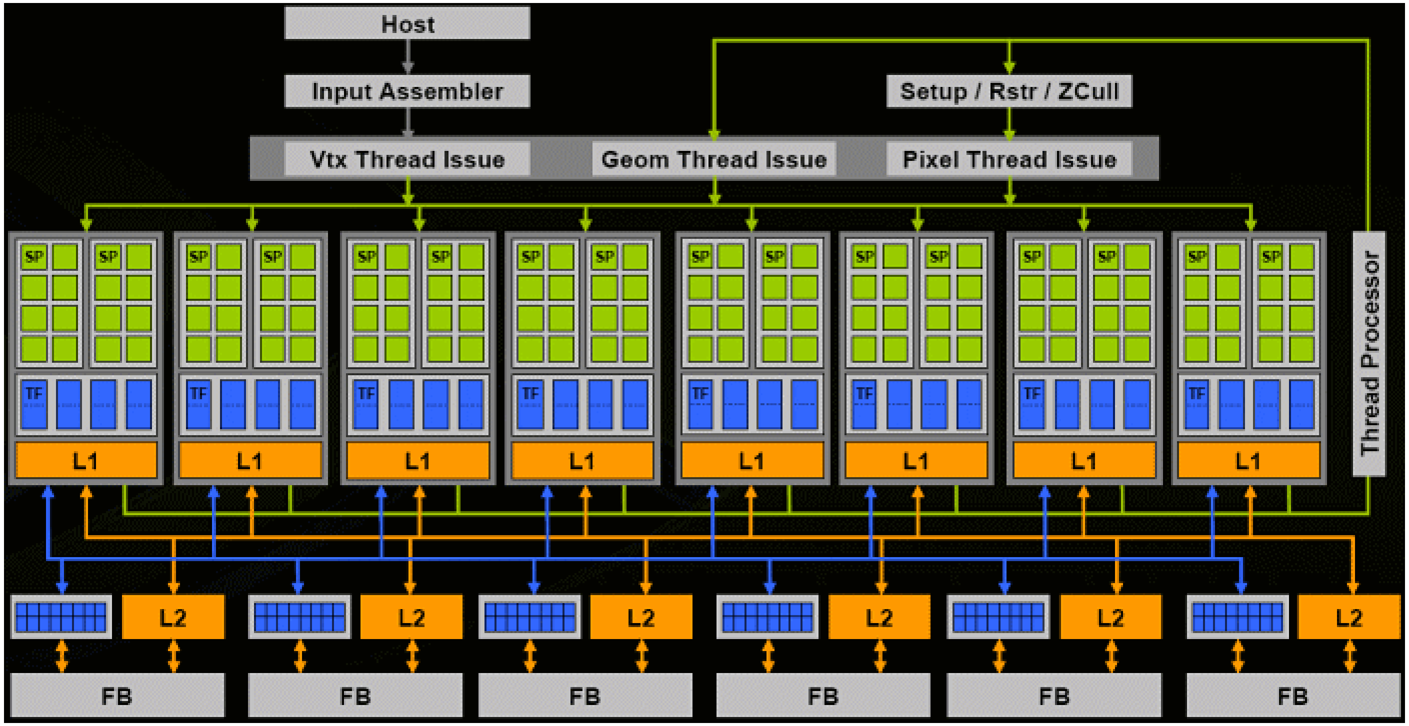
architecture

GPU pipeline – GeForce 8

Fixed number of shader processors

Unified shading architecture

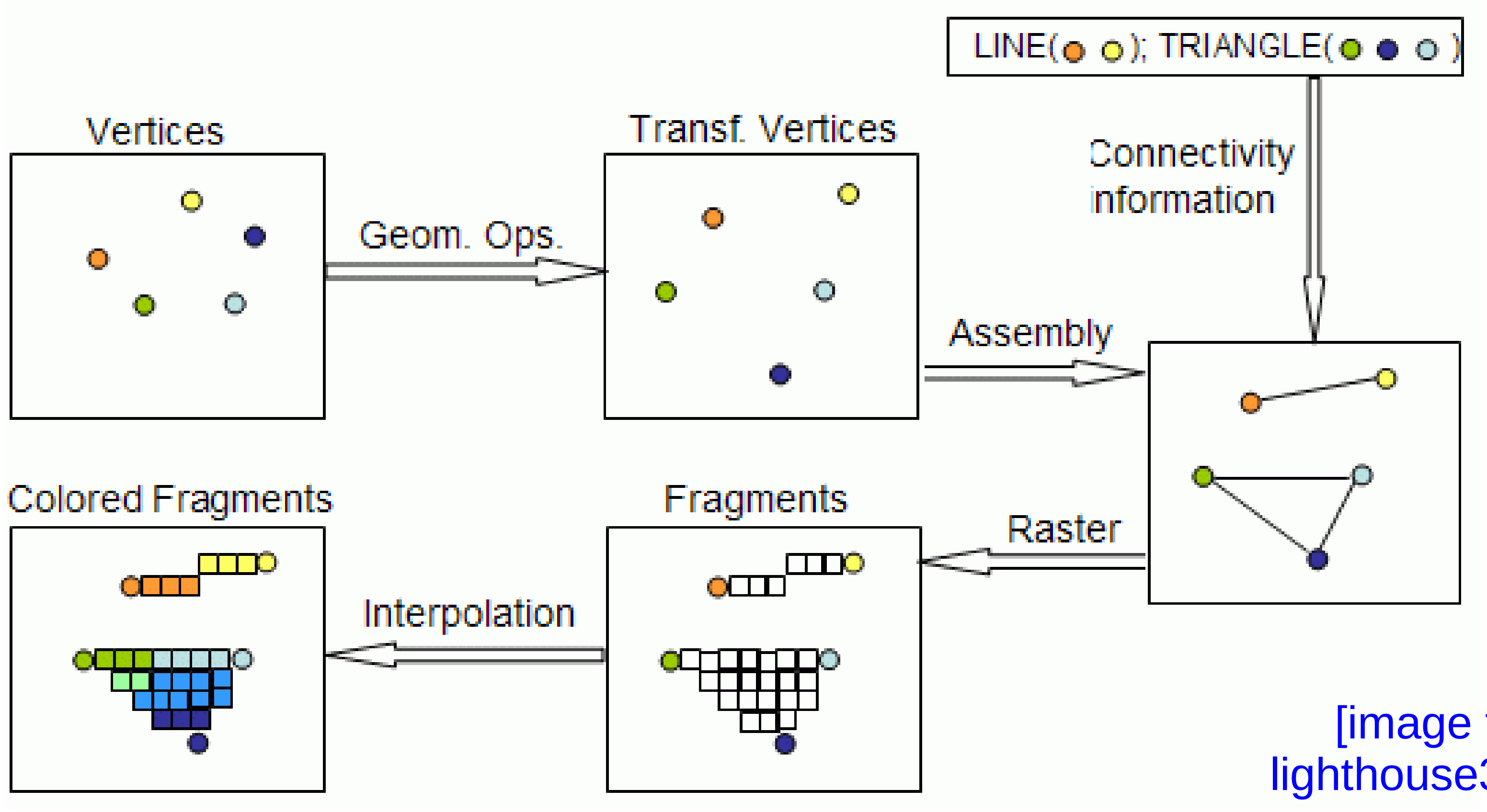
More programmability



architecture

OpenGL logical pipeline

Basic graphics pipeline

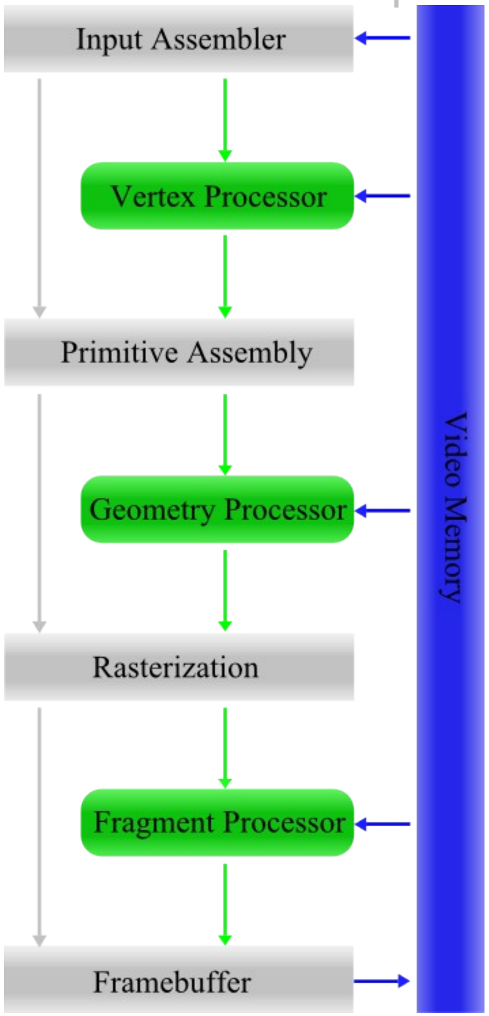


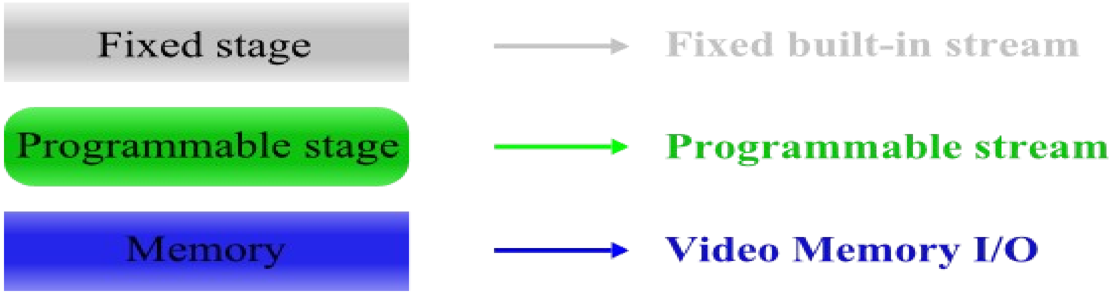
[image from lighthouse3d.com]

architecture

OpenGL logical pipeline

- Fixed functionalities
- Programmable functionalities
- Flexible memory access

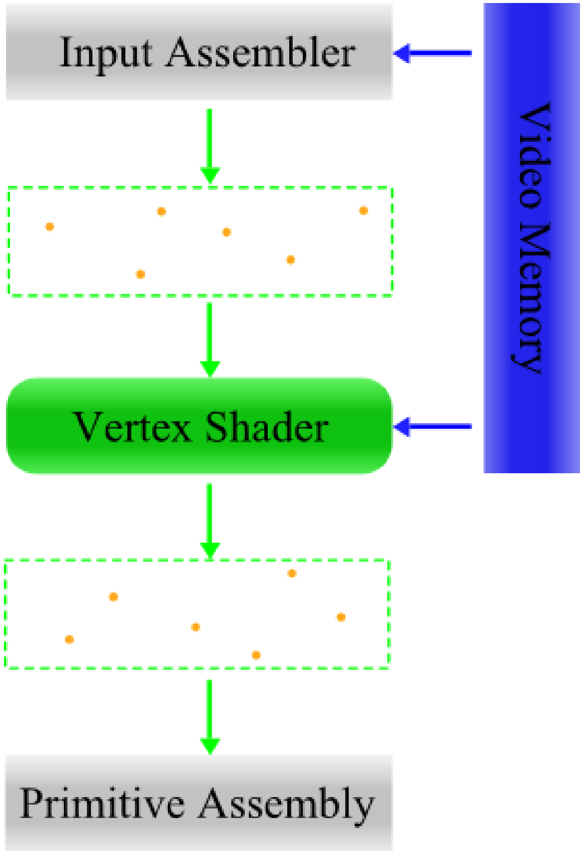




architecture

Vertex Shader

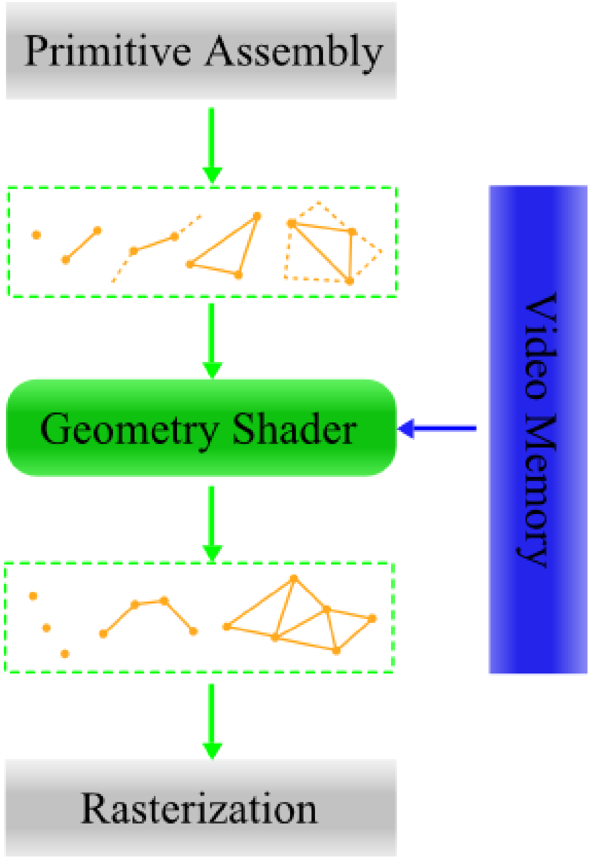
- Vertex transformation
- Once per vertex
- Input attributes
  - Normal
  - Texture coordinates
  - Colors
  - ...



architecture

Geometry Shader

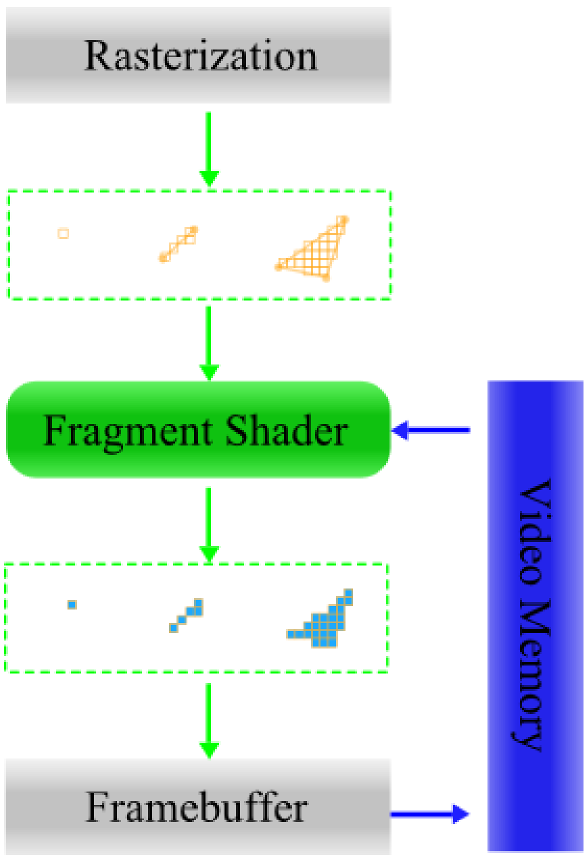
- Geometry composition
- Once per geometry
- Input primitives
  - Points, lines, triangles
  - Lines and triangles with adjacency
- Output primitives
  - Points, line strips or triangle strips
  - [0, n] primitives outputted



architecture

Fragment Shader

- Pre-pixel (or fragment) composition
- Once per fragment
- Operations on interpolated values
  - Vertex attributes
  - User-defined varying variables



language

other shading language

- ARB Assembly Language – Architecture Review Board
- nVidia (+ Microsoft) Cg – C for graphics
- Microsoft HLSL – High Level Shading Language
- RSL – RenderMan Shading Language
- ...

language

shader structure


```
/**
 *        :: Comments ::
 **/

:: Global definitions ::

void main(void) {

    :: Function body ::

}
```

language

hello world vertex shader

```
#version 120
// Vertex Shader Main
void main(void) {
  // Pass vertex color to next stage
  gl_FrontColor = gl_Color;
  // Transform vertex position before passing it
  gl_Position = gl_ModelViewProjectionMatrix
    * gl_Vertex;
}
```

language

hello world geometry shader

```
#extension GL_EXT_geometry_shader4: enable
// Geometry Shader Main
void main(void) {
  // Iterates over all vertices in the input
     primitive
  for (int i = 0; i < gl_VerticesIn; ++i) {
    // Pass color and position to next stage
    gl_FrontColor = gl_FrontColorIn[i];
    gl_Position = gl_PositionIn[i];
    // Done with this vertex
    EmitVertex();
  }
  // Done with the input primitive
  EndPrimitive();
}
```

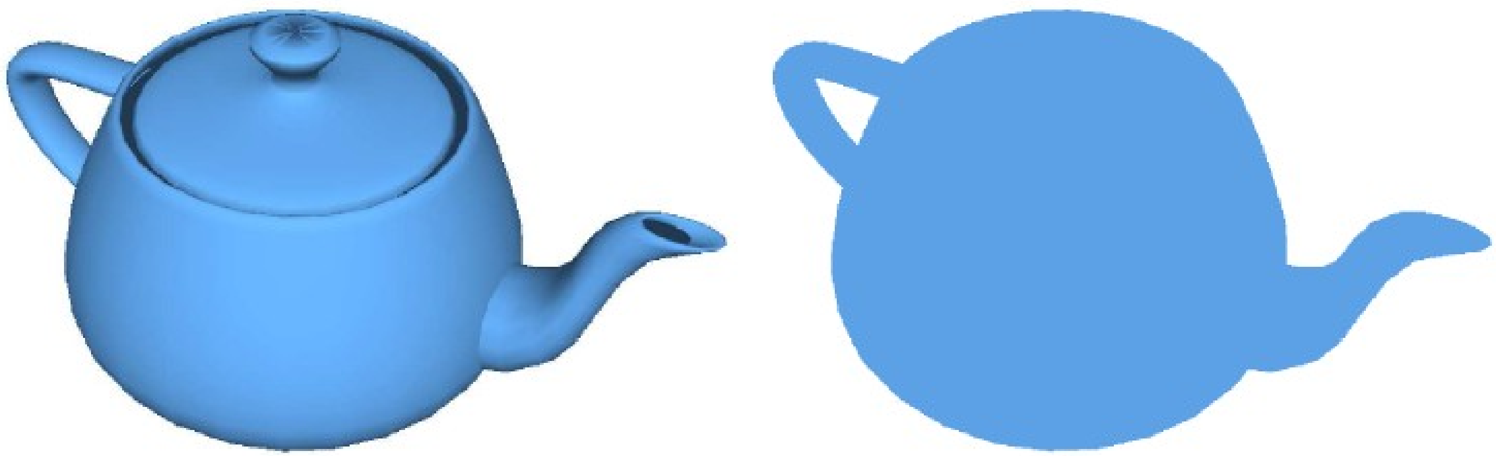
language

hello world fragment shader

```
// Fragment Shader Main
void main(void) {
  // Pass fragment color
  gl_FragColor = gl_Color;
}
```

language

hello world results



language

GLSL basic definitions

- Based on C Ansi with some C++ additions
- Vector types
  - Float, integers and booleans
  - 2-, 3- and 4- components
- Vector components can be swizzled
- Matrix types
  - Only floats
  - 2x2, 3x3 and 4x4 sizes
- Texture types
  - 1-, 2- and 3- dimensions
  - Cube mapping
  - Shadow mapping

DATA TYPES (4.1 p16)

float, vec2, vec3, vec4
int, ivec2, ivec3, ivec4
bool, bvec2, bvec3, bvec4
mat2, mat3, mat4
void
sampler1D, sampler2D, sampler3D
samplerCube
sampler1DShadow, sampler2DShadow

VECTOR COMPONENTS (5.5 p 30)

component names may not be mixed across sets

x, y, z, w
r, g, b, a
s, t, p, q

language

GLSL basic definitions

- Type qualifiers
  - Uniform
  - Attribute
  - Varying
- Function qualifiers
  - In, out and in-out
  - constant

DATA TYPE QUALIFIERS (4.3 p22)

global variable declarations:

uniform	input to Vertex and Fragment shader from OpenGL or application (READ-ONLY)
attribute	input per-vertex to Vertex shader from OpenGL or application (READ-ONLY)
varying	output from Vertex shader (READ/WRITE), interpolated, then input to Fragment shader (READ-ONLY)
const	compile-time constant (READ-ONLY)

function parameters:

in	value initialized on entry, not copied on return (default)
out	copied out on return, but not initialized
inout	value initialized on entry, and copied out on return
const	constant function input

language

GLSL basic definitions

- Built-in functions
    - Sin, cos, tan
    - Pow, exp, sqrt
    - Discard keyword
- Cease fragment processing

Angle and Trigonometry Functions (8.1 p51)

```
genType  sin( genType )
genType  cos( genType )
genType  tan( genType )
...
```

Exponential Functions (8.2 p52)

```
genType  pow( genType, genType )
genType  exp( genType )
genType  log( genType )
genType  exp2( genType )
genType  log2( genType )
genType  sqrt( genType )
genType  inversesqrt( genType )
...
        genType = float | vec2 | vec3 | vec4
```

language

GLSL basic definitions

- Built-in uniform variables
  - Modelview and projection matrices
  - Texture and normal matrices
- Light and material parameters
- Fog and point parameters

BUILT-IN UNIFORMs (7.5 p45) *access=RO*

```
uniform mat4  gl_ModelViewMatrix;
uniform mat4  gl_ModelViewProjectionMatrix;
uniform mat4  gl_ProjectionMatrix;
uniform mat4  gl_TextureMatrix[gl_MaxTextureCoords];

uniform mat4  gl_ModelViewMatrixInverse;
uniform mat4  gl_ModelViewProjectionMatrixInverse;
uniform mat4  gl_ProjectionMatrixInverse;
uniform mat4  gl_TextureMatrixInverse[gl_MaxTextureCoords];

uniform mat4  gl_ModelViewMatrixTranspose;
uniform mat4  gl_ModelViewProjectionMatrixTranspose;
uniform mat4  gl_ProjectionMatrixTranspose;
uniform mat4  gl_TextureMatrixTranspose[gl_MaxTextureCoords];
...
```

language

GLSL additional definitions (Shader Model 4.0)

- Additional vector types
  - Unsigned integers
- Additional texture types
  - Array of textures
  - Buffer textures
- Additional type qualifiers
  - Flat varying
  - Non perspective varying
  - Centroid varying

DATA TYPES

unsigned int, uvec2, uvec3, uvec4
sampler1DArray, sampler2DArray
sampler1DArrayShadow, ...
isampler1D, isampler2D, isampler3D, ...
isampler1DArray, isampler2DArray
usampler1D, usampler2D, usampler3D, ...
usampler1DArray, usampler2DArray
samplerBuffer, isamplerBuffer, usamplerBuffer

DATA TYPE QUALIFIERS

flat varying
noperspective varying
centroid varying

language

GLSL additional definitions (Shader Model 4.0)

Additional built-in functions

Modulo

Bit-wise operations

Texture access

BUILT-IN FUNCTIONS

% : *modulo*

&, |, ^, ~, << , >> : *bit-wise operations*

For texture access:

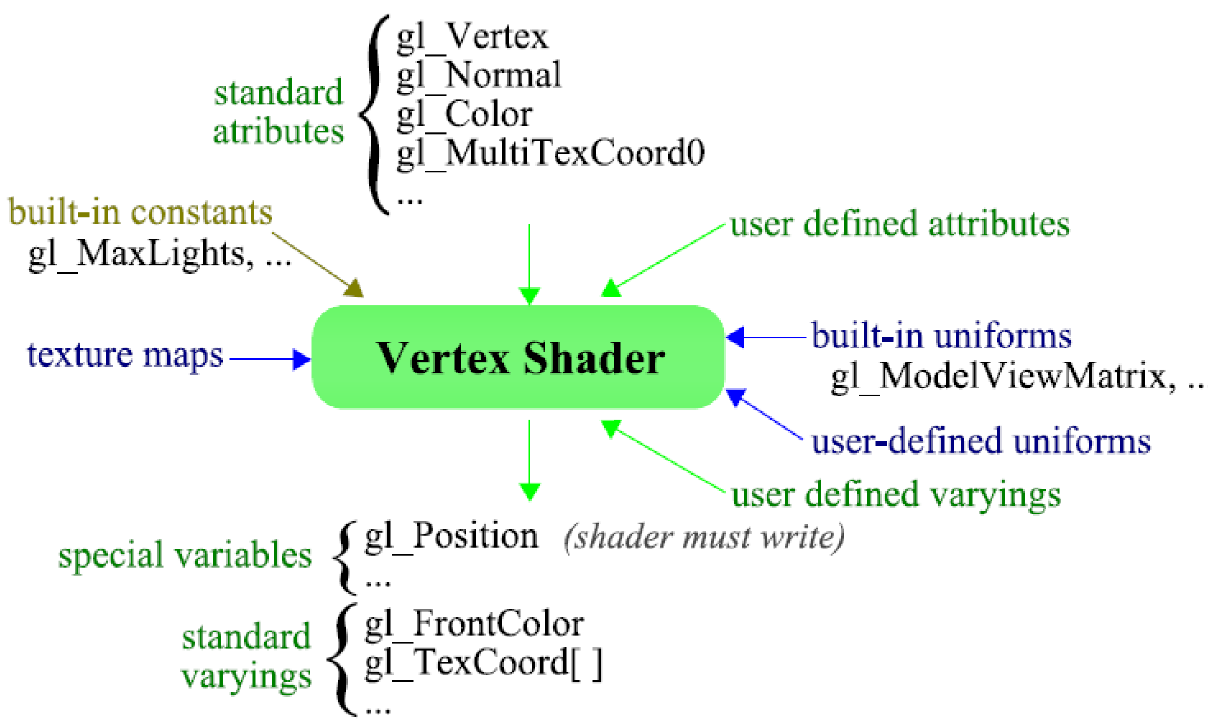
vec4 texture1D(sampler1D sampler, float coord [, float bias])
ivec4 texture1D(isampler1D sampler, float coord [, float bias])
uvec4 texture1D(usampler1D sampler, float coord [, float bias])

More ? cf. EXT_gpu_shader4, g80specs.pdf p117-124

language

data flow

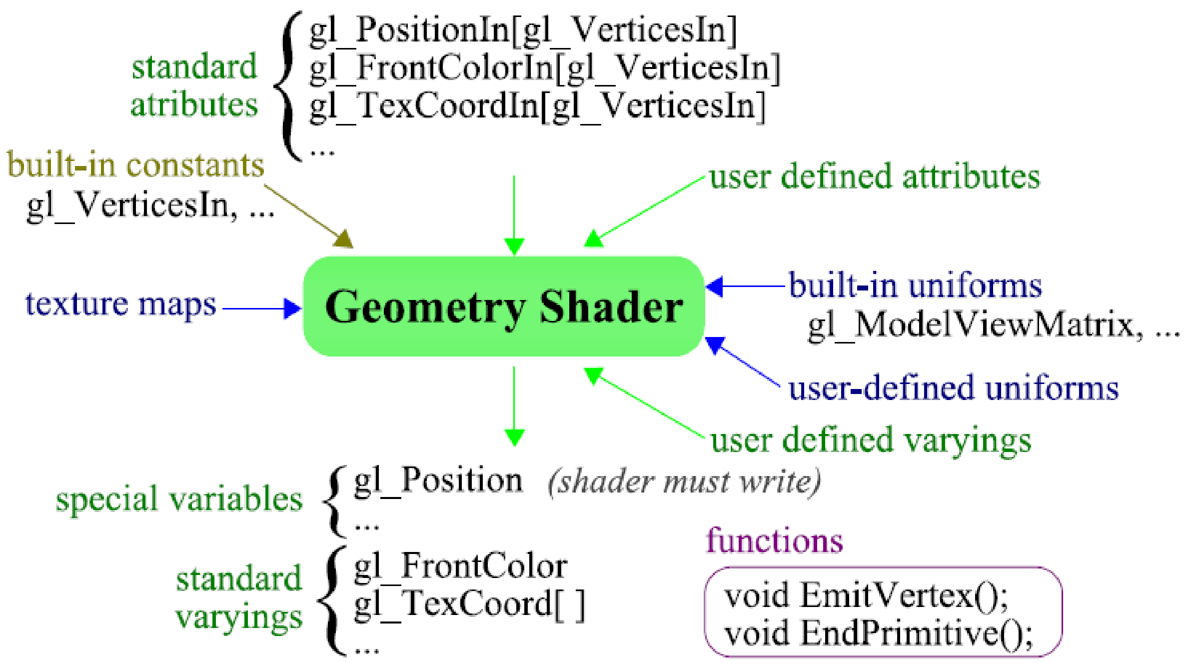
Input/Output summary of the vertex shader



language

data flow

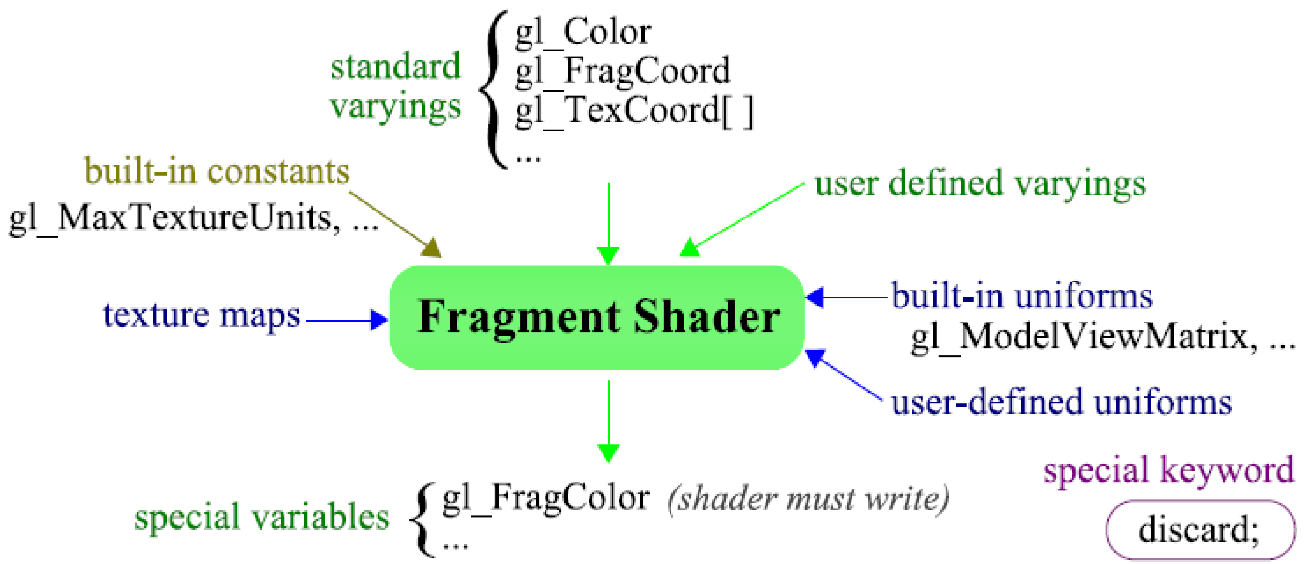
Input/Output summary of the geometry shader



language

data flow

Input/Output summary of the fragment shader



language

OpenGL – GLSL integration


```
#include <GL/glut.h>
// C Main function
int main( int argc, char** argv ) {
  // GLUT Initialization
  glutInit( &argc, argv );
  glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGBA );
  glutInitWindowSize( 512, 512 );
  // Create OpenGL Window
  glutCreateWindow( "Simple Window" );
  init(); // non-GLUT initializations
  // Register callbacks
  glutReshapeFunc( reshape );
  glutDisplayFunc( display );
  glutKeyboardFunc( keyboard );
  glutMouseFunc( mouse );
  // Event Loop
  glutMainLoop();
  return 0;
}
/// The result is a window with 512x512 pixels
```

language

OpenGL – GLSL integration


```
// OpenGL initialization calls for shaders
void initShader() {
  // Vertex Shader code source
  const GLchar* vsSource = {
  "#version 120\n"
  "void main(void) {\n"
  "     gl_FrontColor = gl_Color;\n"
  "     gl_Position = gl_ModelViewProjectionMatrix
                   * gl_Vertex;\n"
  "}"
  }
  // Create program and vertex shader objects
  programObject = glCreateProgram();
  vtxShader = glCreateShader( GL_VERTEX_SHADER );
  // Assign the vertex shader source code
  glShaderSource( vtxShader, 1, &vsSource, NULL );
  // Compile the vertex shader
  glCompileShader( vtxShader );
  // Attach vertex shader to the GPU program
  glAttachShader( programObject, vtxShader );
  // Create an executable to run on the GPU
  glLinkProgram( programObject );
  // Install vertex shader as part of the pipeline
  glUseProgram( programObject );
}
/// The result is a vertex shader acting as a
/// simplified version of the fixed functionality
```

summary

tutorial

Part I

Introduction and motivation (RM)

GPU architecture and pipeline (AM)

GLSL language :

- Hello World (AM)
- Basic Types (AM)
- Data Flow (RM)

OpenGL – GLSL integration (AM)

Part II

Examples:

- Cartoon Effect (AM)
- Texture Mapping (RM)
- Environment Mapping (RM)
- Phong Shading (AM)
- Spike Effect (AM)

GPGPU (RM)

- Particle System (RM)

Wrap-up (RM)

cartoon effect

vertex shader

```
// Output vertex normal to fragment shader
varying out vec3 normal;
void main(void) {
  // Compute normal per-vertex
  normal = normalize(gl_NormalMatrix * gl_Normal);
  gl_FrontColor = gl_Color;
  // Transform position using built-in function
  gl_Position = ftransform();
}
```

cartoon effect

fragment shader

```
// Input vertex normal from vertex shader
varying in vec3 normal;
void main(void) {
  // Compute light direction
  vec3 ld = normalize( vec3(
    gl_LightSource[0].position) );
  // Compute light intensity to the surface
  float ity = dot( ld, normal );
  // Weight the final color in four cases,
  // depending on the light intensity
  vec4 fc;
  if (ity > 0.95) fc = 1.00 * gl_Color;
  else if (ity > 0.50) fc = 0.50 * gl_Color;
  else if (ity > 0.25) fc = 0.25 * gl_Color;
  else fc = 0.10 * gl_Color;
  // Output the final color
  gl_FragColor = fc;
}
```

cartoon effect

results

texture mapping

map image onto polygon model

increase realism

avoid detailed meshes -> gain performance

texture mapping

texture coordinates

```
//Draw a textured square
glBegin(GL_QUADS);
glTexCoord2f(0.0, 1.0);
glVertex3f(-0.5, -0.5, 0.0);
glTexCoord2f(1.0, 1.0);
glVertex3f( 0.5, -0.5, 0.0);
glTexCoord2f(1.0, 0.0);
glVertex3f( 0.5,  0.5, 0.0);
glTexCoord2f(0.0, 0.0);
glVertex3f(-0.5,  0.5, 0.0);
glEnd();
```

texture mapping

vertex shader

```
void main(void) {
  // Pass texture coordinate to next stage
  gl_TexCoord[0] = gl_TextureMatrix[0]
    * gl_MultiTexCoord0;
  // Pass color and transformed position
  gl_FrontColor = gl_Color;
  gl_Position = ftransform();
}
```

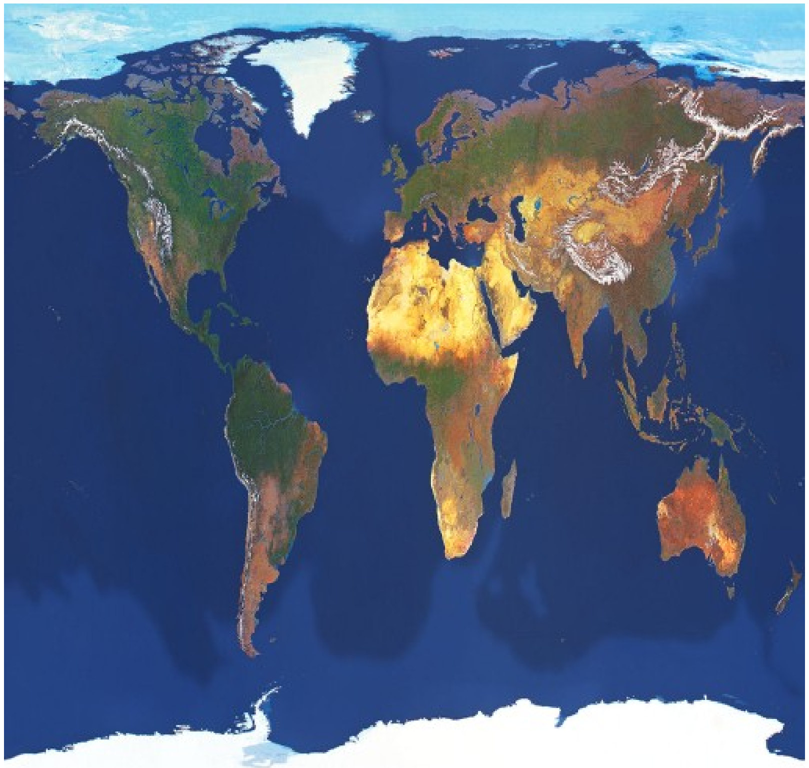
texture mapping

fragment shader

```
// User-defined uniform to access texture
uniform sampler2D texture;
void main(void) {
  // Read a texture element from a texture
  vec4 texel = texture2D( texture,
    gl_TexCoord[0].st );
  // Output the texture element as color
  gl_FragColor = texel;
}
```

texture mapping

results



phong shading

vertex shader

```
// Output vertex normal and position
varying out vec3 normal, vert;
void main(void) {
  // Store normal per-vertex to fragment shader
  normal = normalize( gl_NormalMatrix
    * gl_Normal );
  // Compute vertex position in model-view space
  // to be used in the fragment shader
  vert = vec3( gl_ModelViewMatrix * gl_Vertex );
  // Pass color
  gl_FrontColor = gl_Color;
  // Pass transformed position
  gl_Position = ftransform();
}
```

phong shading

fragment shader

```
// Additional input from vertex shader:
// vertex normal and position
varying in vec3 normal, vert;
void main(void) {
  // Compute light and eye direction
  vec3 lp = gl_LightSource[0].position.xyz;
  vec3 ld = normalize( lp − vert );
  vec3 ed = normalize( − vert );
  // Compute reflection vector based on
  // light direction and normal
  vec3 r = normalize( −reflect( ld, normal ) );
  // Compute light parameters per fragment
  vec4 la = gl_FrontLightProduct[0].ambient;
  vec4 lf = gl_FrontLightProduct[0].diffuse
    ∗ max( dot(normal, ld), 0.0 );
  vec4 ls = gl_FrontLightProduct[0].specular
    ∗ pow( max( dot(r, ed), 0.0 ),
           gl_FrontMaterial.shininess );
  // Use light parameters to compute final color
  gl_FragColor = gl_FrontLightModelProduct.
    sceneColor + la + lf + ls;
}
```

phong shading

results

environment mapping

simulate reflection of ambient on an object

map photo onto sphere (use your favorite image editor)

use mapped photo as texture for look up

environment mapping

vertex shader

```
// Output reflection vector per-vertex
varying out vec3 r;
void main(void) {
  // Pass texture coordinate
  gl_TexCoord[0] = gl_MultiTexCoord0;
  // Compute vertex position in model-view space
  vec3 v = normalize( vec3( gl_ModelViewMatrix
    * gl_Vertex ) );
  // Compute vertex normal
  vec3 n = normalize(gl_NormalMatrix*gl_Normal);
  // Compute reflection vector
  r = reflect( u, n );
  // Pass transformed position
  gl_Position = ftransform();
}
```

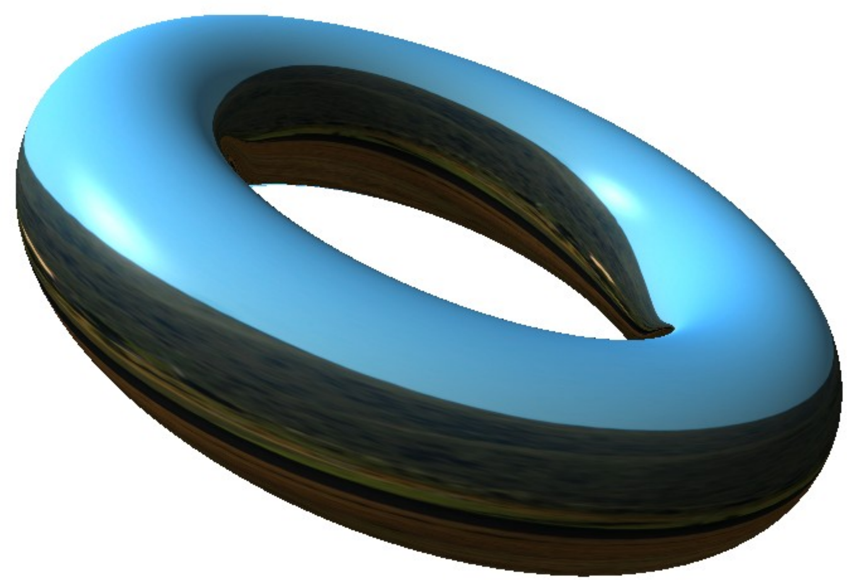
environment mapping

fragment shader

```
// Input reflection vector from vertex shader
varying in vec3 r;
// Texture id to access environment map
uniform sampler2D envMapTex;
void main(void) {
  // Compute texture coordinate using the
  // interpolated reflection vector
  float m = 2.0 * sqrt( r.x*r.x + r.y*r.y
    + (r.z+1.0)*(r.z+1.0) );
  vec2 coord = vec2(r.x/m + 0.5, r.y/m + 0.5);
  // Read corresponding texture element
  vec4 texel = texture2D(envMapTex, coord.st);
  // Output texture element as fragment color
  gl_FragColor = texel;
}
```

environment mapping

results



spike effect

vertex shader

```
void main(void) {
  gl_Position = gl_Vertex; // Pass-thru vertex
}
```

spike effect

geometry shader I

```
varying out vec3 normal, vert; // Output to FS
void main() {
  // Store original triangle's vertices
  vec4 v[3];
  for (int i=0; i<3; ++i)
    v[i] = gl_PositionIn[i];
  // Compute triangle's centroid
  vec3 c = ( v[0] + v[1] + v[2] ).xyz / 3.0;
  // Compute original triangle's normal
  vec3 v01 = ( v[1] - v[0] ).xyz;
  vec3 v02 = ( v[2] - v[0] ).xyz;
  vec3 tn = -cross( v01, v02 );
  // Compute middle vertex position
  vec3 mp = c + 0.5 * tn;
  // Generate 3 triangles using middle vertex
  for (int i = 0; i < gl_VerticesIn; ++i) {
    // Compute triangle's normal
    v01 = ( v[(i+1)%3] - v[i] ).xyz;
    v02 = mp - v[i].xyz;
    tn = -cross( v01, v02 );
```

spike effect

geometry shader II


```
    // Compute and send first vertex
    gl_Position = gl_ModelViewProjectionMatrix
      * v[i];
    normal = normalize(tn);
    vert = vec3( gl_ModelViewMatrix * v[i] );
    EmitVertex();
    // Compute and send second vertex
    gl_Position = gl_ModelViewProjectionMatrix
      * v[(i+1)%3];
    normal = normalize(tn);
    vert = vec3(gl_ModelViewMatrix * v[(i+1)%3]);
    EmitVertex();
    // Compute and send third vertex
    gl_Position = gl_ModelViewProjectionMatrix
      * vec4( mp, 1.0 );
    normal = normalize(tn);
    vert = vec3(gl_ModelViewMatrix*vec4(mp,1.0));
    EmitVertex();
    // Finish this triangle
    EndPrimitive();
  }
}
```

spike effect

results

GPGPU

General Purpose Computation on Graphics Processing Unit

- use the GPU as a parallel machine
- auxiliary computation for graphics
- non-graphics stuff
- scientific computation
- mask non-graphics primitives:
  - pixels and vertices may represent any kind of information

GPGPU

General Purpose Computation on Graphics Processing Unit

example: particle simulation

first pass: compute positions using basic kinematics

no rendering

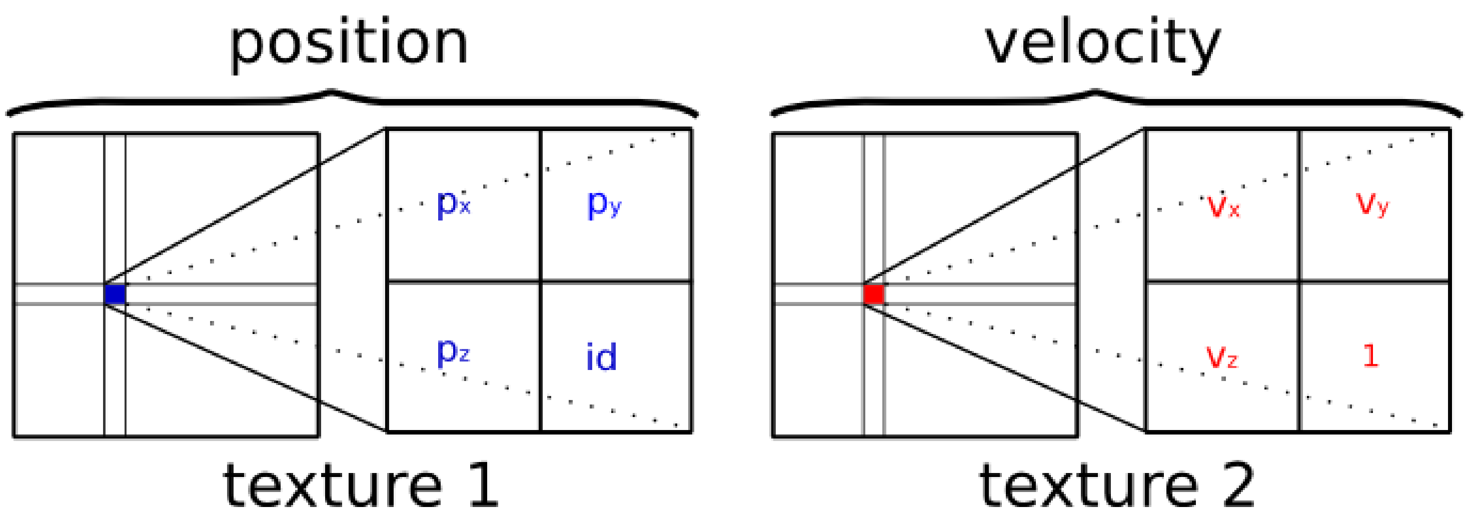
second pass: display particles

rendering

particle simulation

first pass

- basic kinematics
- store information using textures
    - each texel -> one particle
    - two textures: one for position, one for velocity
- do not render -> write output to texture

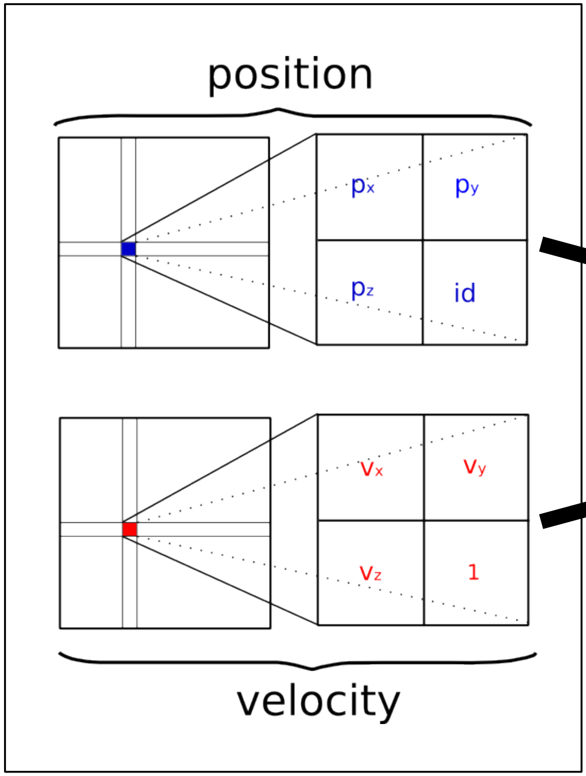


particle simulation

first pass

compute positions using basic kinematics

input: current

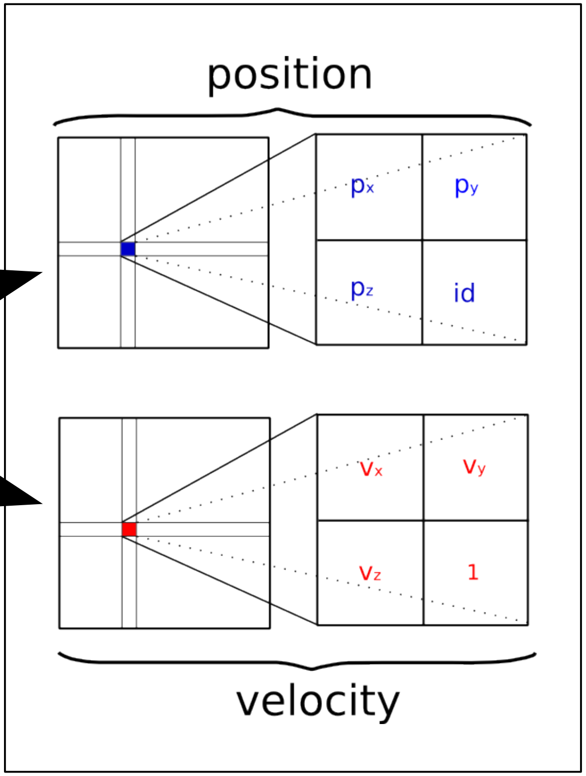


shaders

$$p_{new} = p_{cur} + v_{cur} * t$$

$$v_{new} = v_{cur} + a_{cur} * t$$

output: new



acceleration: simulate gravity

particle simulation

first pass

```
glViewport(0, 0, tex_size, tex_size);
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();
gluOrtho2D(0.0, tex_size, 0.0, tex_size);

glBegin(GL_QUADS);
glTexCoord2f(0.0, 0.0);
glVertex2f(0.0, 0.0);
glTexCoord2f(1.0, 0.0);
glVertex2f( tex_size, 0.0);
glTexCoord2f(1.0, 1.0);
glVertex2f( tex_size, tex_size);
glTexCoord2f(0.0, 1.0);
glVertex2f(0.0, tex_size);
glEnd();
```

how to access one texture position per fragment shader pass?

draw a Quad the size of the screen

one texel = one pixel = one fragment pass

particle simulation

vertex shader

```
void main(void) {
  gl_TexCoord[0] = gl_MultiTexCoord0 *
       gl_TextureMatrix[0];
  gl_Position = gl_ProjectionMatrix * gl_Vertex;
}
```

setup the texture coordinate

do not transform using modelview-> keep Quad aligned with screen

particle simulation

fragment shader


```
uniform sampler2D positionTex;
uniform sampler2D velocityTex;
uniform float time_step;
uniform vec3 gravity;

void main(void) {
  // retrieve data from textures
  vec3 position = texture2D(positionTex,
      gl_TexCoord[0].st).xyz;
  vec3 velocity = texture2D(velocityTex,
      gl_TexCoord[0].st).xyz;

  // update particle
  position.xyz = position.xyz + velocity.xyz*
      time_step;
  velocity.xyz = velocity.xyz + gravity.xyz*
      time_step;
  position = clamp (position, -1.0, 1.0);

  // write output to two textures
  gl_FragData[0] = vec4(position.xyz, 1.0);
  gl_FragData[1] = vec4(velocity.xyz, 1.0);
}
```

Attention: read/write to same texture is not specified by documentation! we are using it here for the sake of simplicity and because we are sure each particle only reads/writes from/to a single texture position (no conflict)

particle simulation

first pass

how to write to a texture?

usually writes to the render buffer

Framebuffer extension lets you write elsewhere

1. Create texture
2. Create framebuffer
3. Bind texture to framebuffer
4. Before rendering quad, specify draw buffer
5. Read back information from framebuffer

particle simulation

create texture

```
GLfloat *tex_data = new GLfloat[4*numParticles];

srand ( time(NULL) );

for (int i = 0; i < numParticles; ++i) {
  tex_data[4*i + 0] = 0.0;
  tex_data[4*i + 1] = 0.0;
  tex_data[4*i + 2] = 0.0;
  tex_data[4*i + 3] = i/(GLfloat)numParticles;
}

glGenTextures(1, &tex_position);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, tex_position);
glTexParameteri(GL_TEXTURE_2D,
    GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,
    GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
    GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
    GL_CLAMP);
glTexImage2D(GL_TEXTURE_2D, 0, TEXTURE_TYPE,
    tex_size, tex_size, 0, GL_RGBA, GL_FLOAT, &
    tex_data[0]);
```

particle simulation

bind texture to framebuffer

```
//Create framebuffer object
glGenFramebuffersEXT(1, &fbo);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo);

//Bind position and velocity textures to fbo
glBindTexture(GL_TEXTURE_2D, tex_position);
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
    GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D,
    tex_position, 0);

glBindTexture(GL_TEXTURE_2D, tex_velocity);
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
    GL_COLOR_ATTACHMENT1_EXT, GL_TEXTURE_2D,
    tex_velocity, 0);

//Disable fbo
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
```

particle simulation

render to texture


```
//Enable fbo and texture
glEnable(GL_TEXTURE_2D);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo);
//Specify target buffers for rendering
GLenum drawBufs[] = { GL_COLOR_ATTACHMENT0,
    GL_COLOR_ATTACHMENT1 };
glDrawBuffers(2, drawBufs);

//Enable shader
computeShader.use();
computeShader.set_uniform("positionTex", 0);
computeShader.set_uniform("velocityTex", 1);
computeShader.set_uniform("time_step", (GLfloat)
    time_step);
computeShader.set_uniform("gravity", gravity[0],
    gravity[1], gravity[2]);

//Draw textured quad
glBegin(GL_QUADS);
...
glEnd();

//Disable fbo, texture and shader
computeShader.use(0);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
glDisable(GL_TEXTURE_2D);
```

particle simulation

second pass

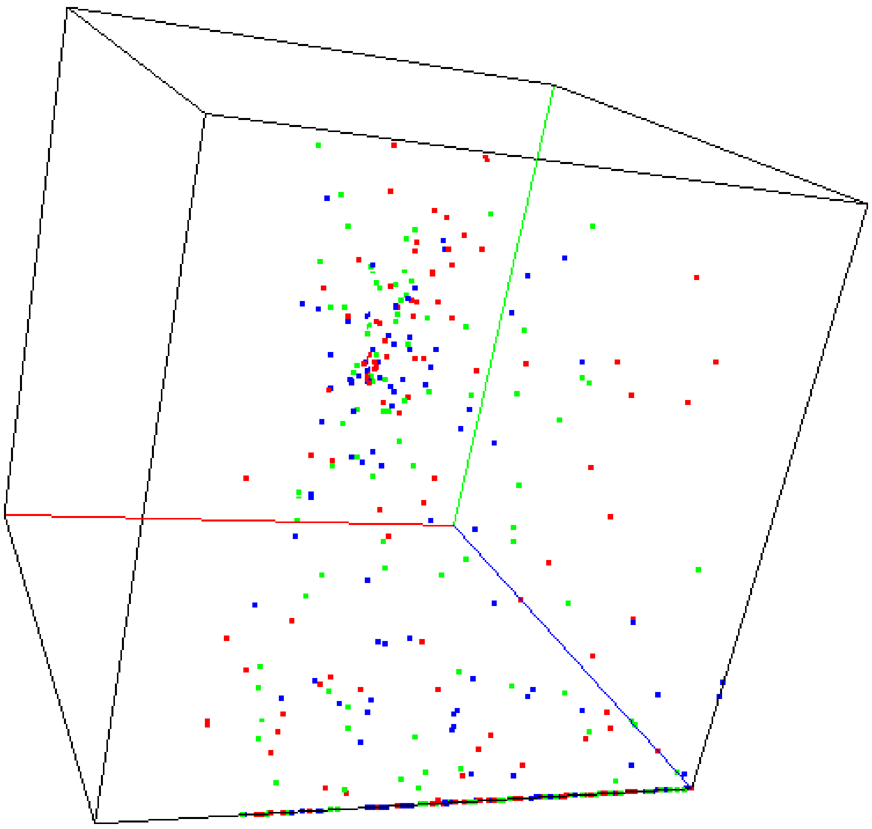
read current particle position

```
//Setup fbo and textures
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, tex_position);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, tex_velocity);

//Read back positions
GLfloat *tex_data = new GLfloat[4*numParticles];
glReadBuffer(GL_COLOR_ATTACHMENT0);
glReadPixels(0, 0, tex_size, tex_size, GL_RGBA,
    GL_FLOAT, &tex_data[0]);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);

//Render points
glPointSize(point_size);
glBegin(GL_POINTS);
for (int i = 0; i < numParticles; ++i) {
  glVertex3f( tex_data[4*i + 0], tex_data[4*i +
      1], tex_data[4*i + 2]);
}
glEnd();
```

Not always necessary to read back information: use as input for next step

particle simulation

results



extra in code: reset particles and rotate gravity vector

wrap-up

important points about shader programming:

- understand:
  - graphics pipeline
  - parallel programming (streaming)
- watch out:
  - shader too long: rethink your strategy
  - too many texture fetches: slow
  - try to keep things inside the GPU
- internet: **lots** of examples
- above all: **experiment!**

wrap-up

so, why use shaders?

what's coming up?

- OpenGL 3.2, GLSL 1.5
- hull and domain shader
- nVidia 300 series(?), Intel Larrabee(?)

will GPGPU languages (CUDA, OpenCL) substitute shaders?

do I really need to know shader programming if I want to write graphics applications?

wrap-up

important links

- **codes for all presented demos**
- **last version of this presentation**
- **survey paper**
- **cool and useful links**

Tutorial project: htttp://www.lcg.ufrj.br/Projectos/glsl_tutorial

Sample codes: http://code.google.com/p/glsl-intro-shaders

AM page: http://www.lcg.ufrj.br/Members/andream

RM page: http://www.lcg.ufrj.br/Member/ricardo

AM email: andmax@gmail.com

RM email: ricardo.marroquim@gmail.com

any further questions?
we will be very happy to help, do not hesitate to contact us!