Shaders

Slide credit to Prof. Zwicker

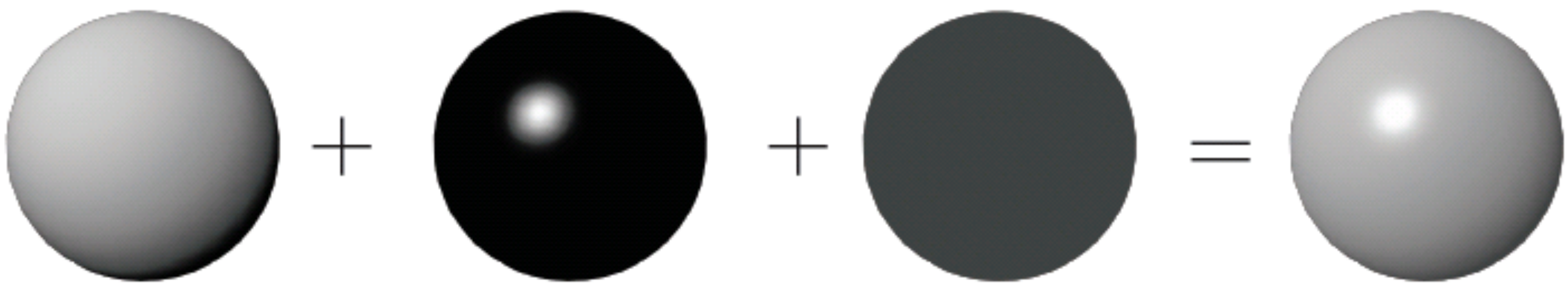
Today

- Shader programming

Complete model

- Blinn model with several light sources i

$$c = \sum_i c_{l_i}\left(k_d\left(\mathbf{L}_i \cdot \mathbf{n}\right) + k_s\left(\mathbf{h}_i \cdot \mathbf{n}\right)^s\right) + k_a c_a$$

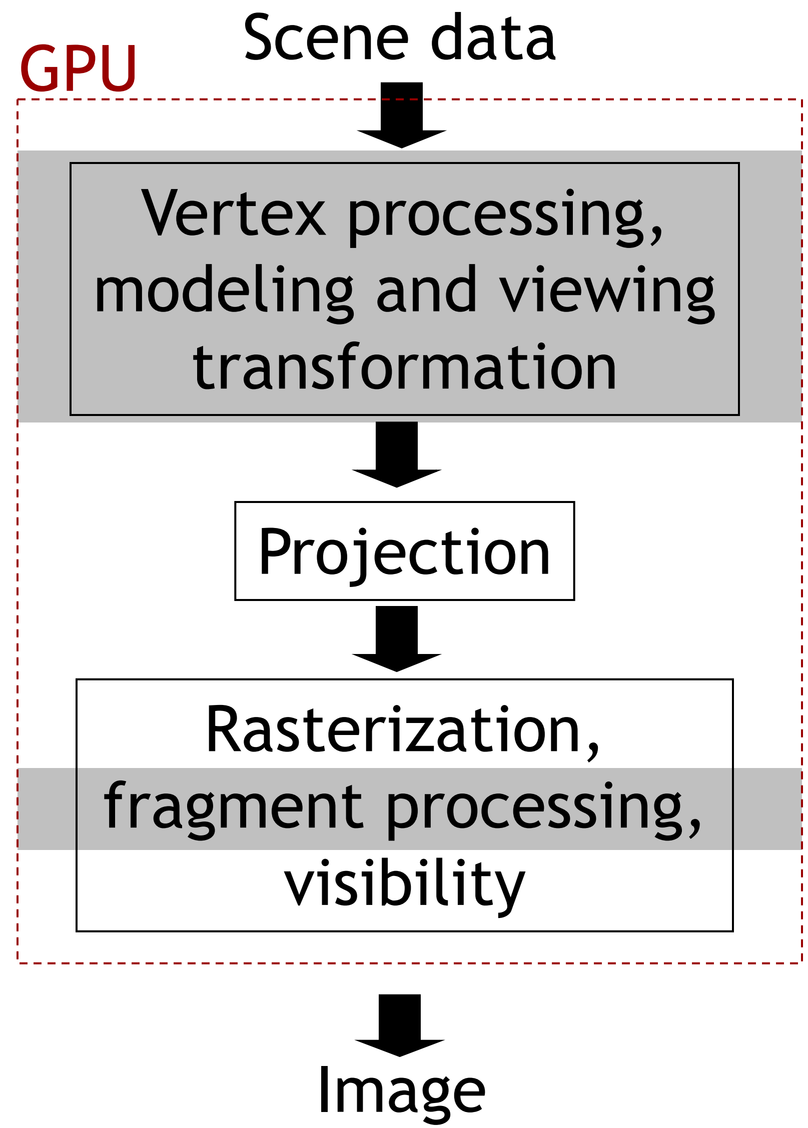


diffuse specular ambient

How is this implemented on the graphics processor (GPU)? Shader programming!

Programmable pipeline

- Functionality in parts (grey boxes) of the GPU pipeline specified by user programs
- Called shaders, or shader programs, executed on GPU
- Not all functionality in the pipeline is programmable



Shader programs

- Written in a shading language
- Examples
  - Cg, early shading language by NVidia
  - OpenGL's shading language GLSL
    http://en.wikipedia.org/wiki/GLSL
  - DirectX' shading language HLSL (high level shading language)
    http://en.wikipedia.org/wiki/High_Level_Shader_Language
  - RenderMan shading language (film production)
  - All similar to C, with specialties
- Recent, quickly changing technology
- Driven by more and more flexible GPUs

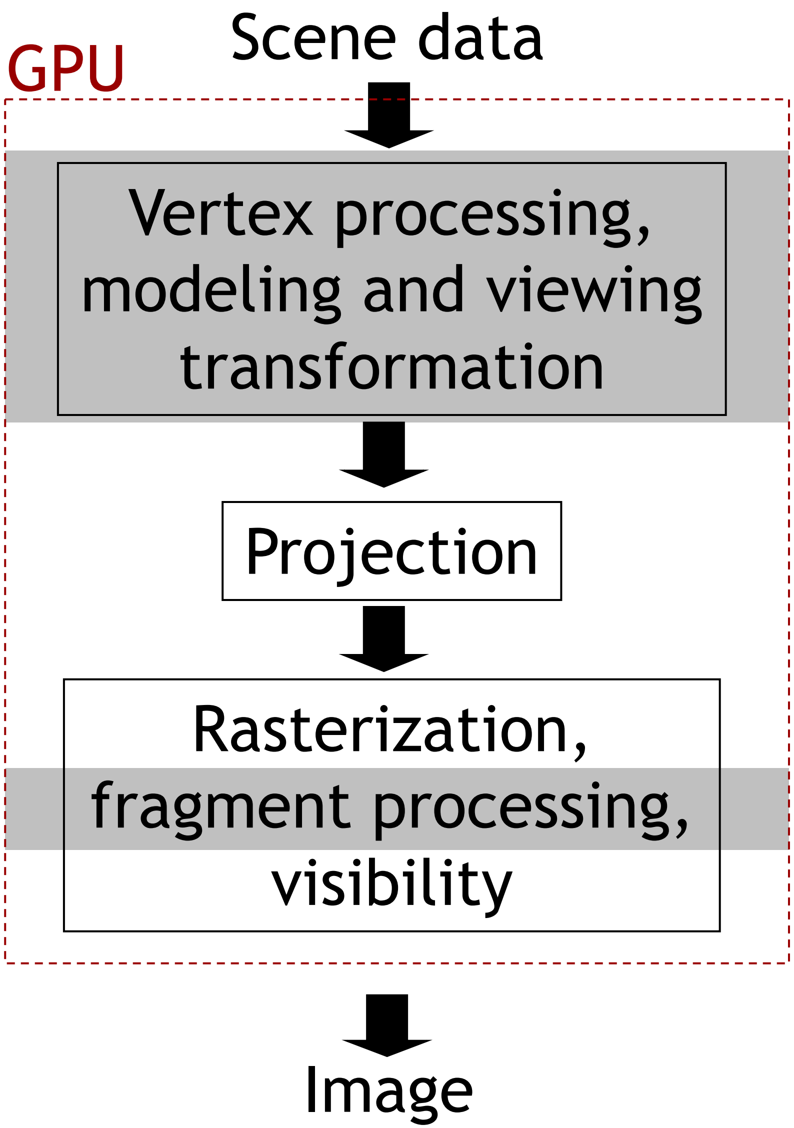
Programmable pipeline (2006)

Two types of shader programs

1. Vertex program

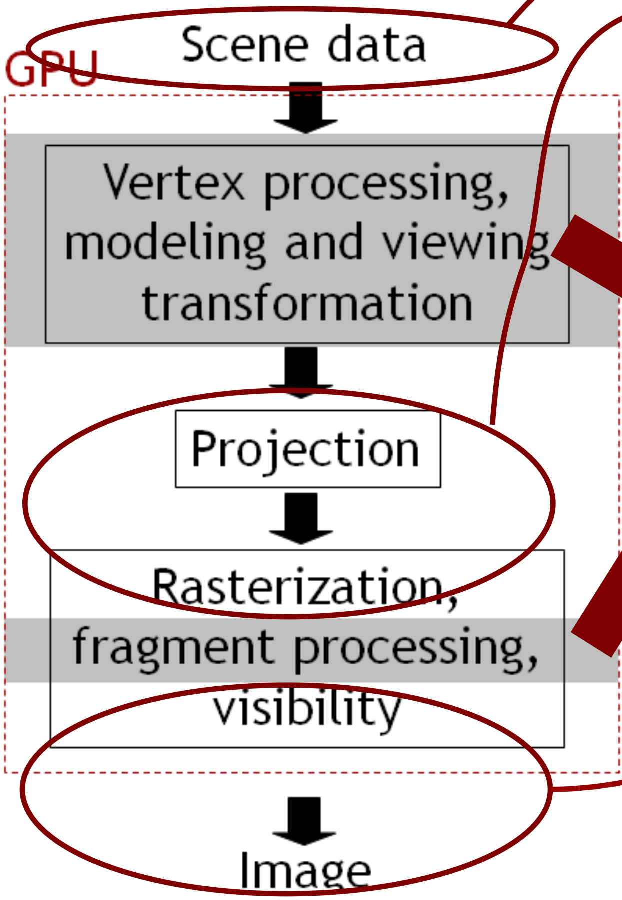
2. Fragment program

(fragment: pixel location inside a triangle and interpolated data)

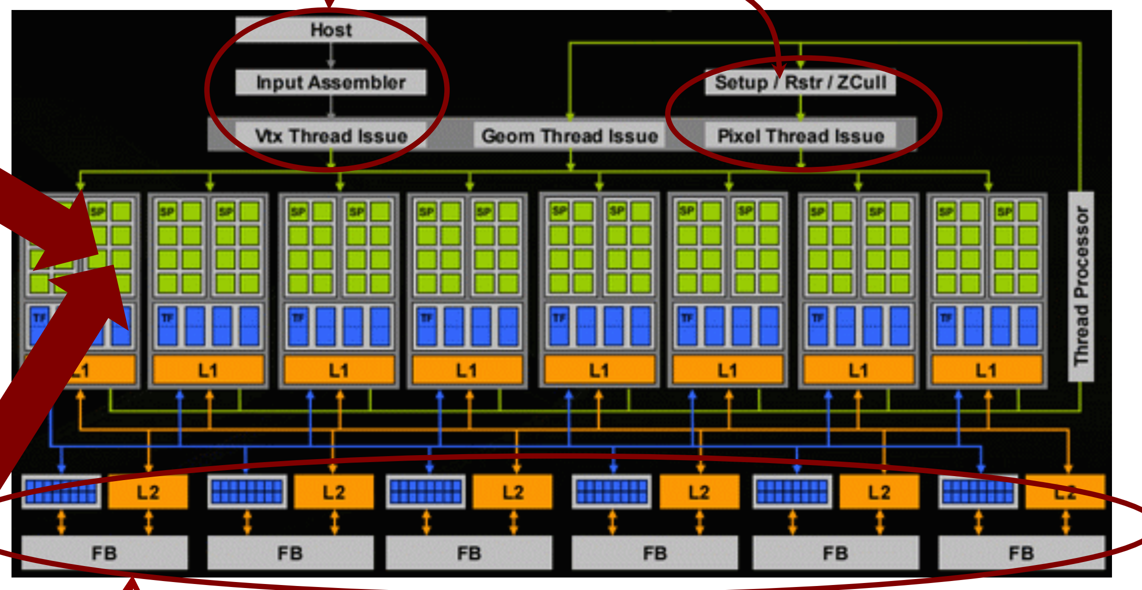


GPU architecture (2006)

Pipeline



GPU Architecture



NVidia NV80 (GeForce 8800 Series)

128 functional units, "stream processors"

http://arstechnica.com/news.ars/post/20061108-8182.html

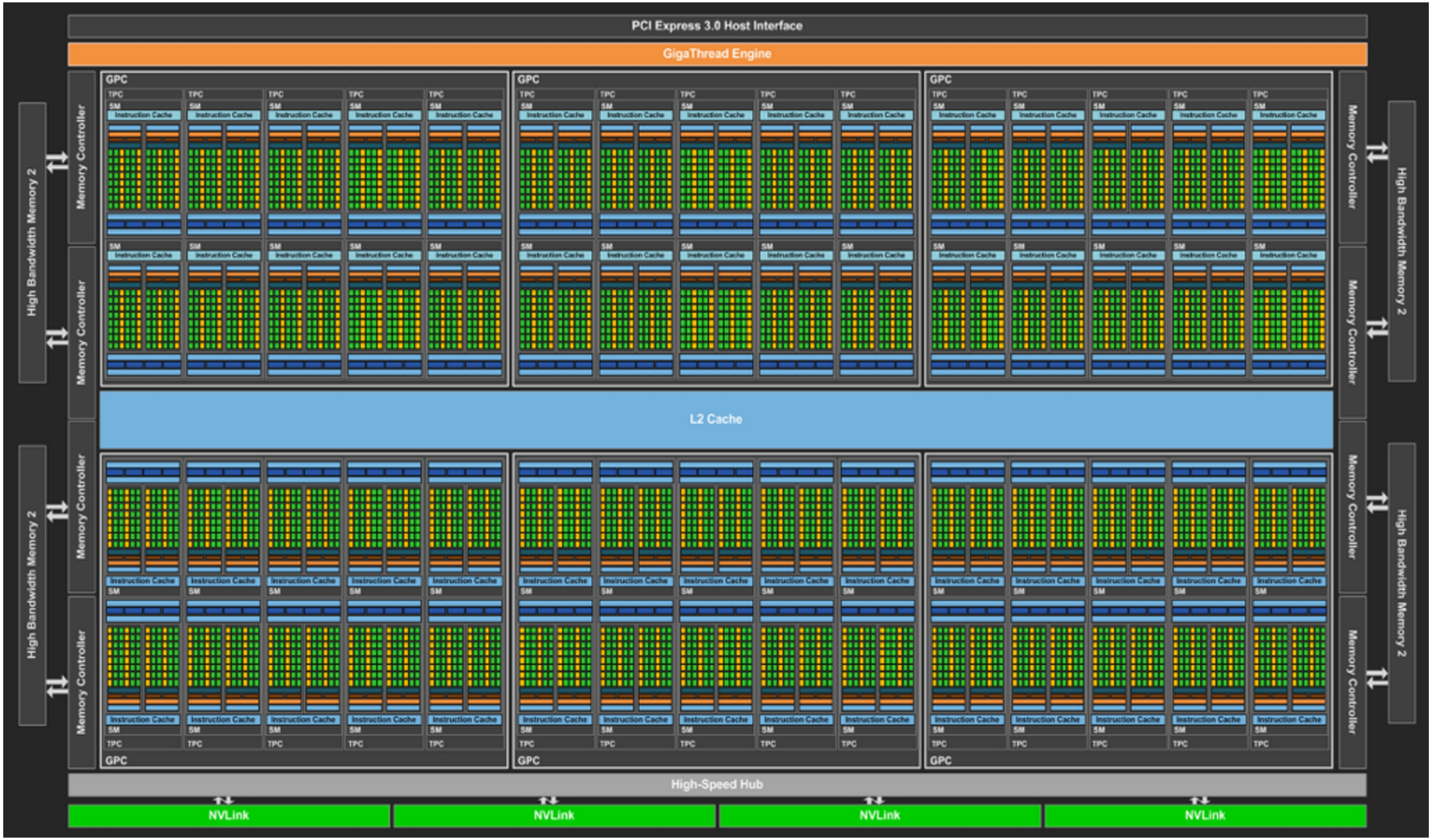
GPU architecture (2014)

- Similar, but more processors (2048 ■)

http://hexus.net/tech/reviews/graphics/74849-nvidia-geforce-gtx-980-28nm-maxwell/

GPU architecture (2016)

- Similar, but even more processors (3840 ■)



https://devblogs.nvidia.com/parallelforall/inside-pascal/

Parallelism

- Task parallelism
  http://en.wikipedia.org/wiki/Task_parallelism
  - Processor performs different threads (sequences of instructions) simultaneously
  - Multi-core CPUs
- Data parallelism
  http://en.wikipedia.org/wiki/Data_parallelism
  - Processors performs same thread of instructions on different data elements
  - Single Instruction Multiple Data (SIMD)
  - GPUs

Parallelism

- GPUs up to now exploit mostly data parallelism
  - Perform same thread of operations (same shader) on multiple vertices and pixels independently
  - Massive parallelism (same operation on many vertices, pixels) enables massive number of operations per second
  - Currently: hundreds of parallel operations at several hundred megahertz
- Detailed description of Nvidia „Kepler“ architecture
  http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf

Still fixed functionality (2014)

- “Hardcoded in hardware”
- Projective division
- Rasterization
  - I.e., determine which pixels lie inside triangle
  - Vertex attribute interpolation (color, texture coords.)
- Access to framebuffer
  - Z-buffering
  - Texture filtering
  - Framebuffer blending

Shader programming

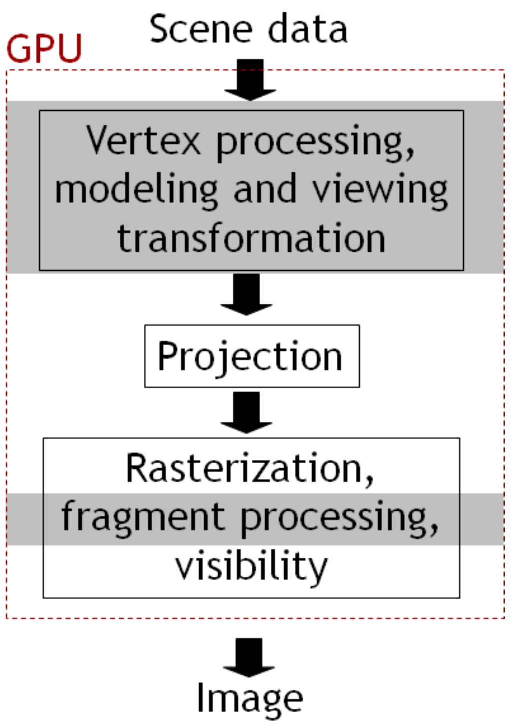
- Each shader (vertex or fragment) is a separate piece of code in a shading language (e.g. GLSL)
- Vertex shader
  - Executed automatically for each vertex and its attributes (color, normal, texture coordinates) flowing down the pipeline
  - Type and number of output variables of vertex shader are user defined
  - Every vertex produces same type of output
  - Output interpolated automatically at each fragment and accessible as input to fragment shader
- Fragment shader
  - Executed automatically for each fragment (pixel) being touched by rasterizer
  - Output (fragment color) is written to framebuffer

Shader programming

- Shaders are activated/deactivated by host program (Java, C++, …)
  - Can have different shaders to render different parts of a scene
- Shader programs can use additional variables set by user
  - Modelview and projection matrices
  - Light sources
  - Textures
  - Etc.
- Variables are passed by host (Java, C++) program to shader
  - In jrtr via jogl, see class jrtr.GLRenderContext
- Learn OpenGL details from example code, then (advanced) reference books, e.g. http://www.opengl.org/registry/doc/glspec40.core.20100311.pdf

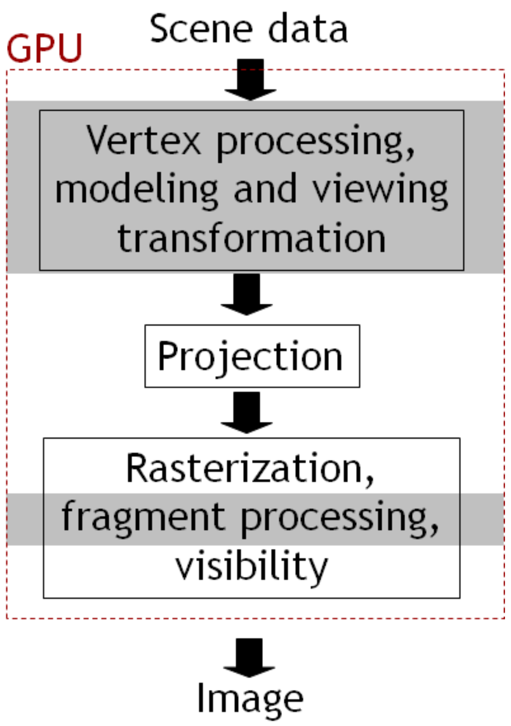
Vertex programs

- Executed once for every vertex
  - Or: “every vertex is processed by same vertex program that is currently active”
- Implements functionality for
  - Modelview, projection transformation (required!)
  - Per-vertex shading
- Vertex shader often used for animation
  - Characters
  - Particle systems



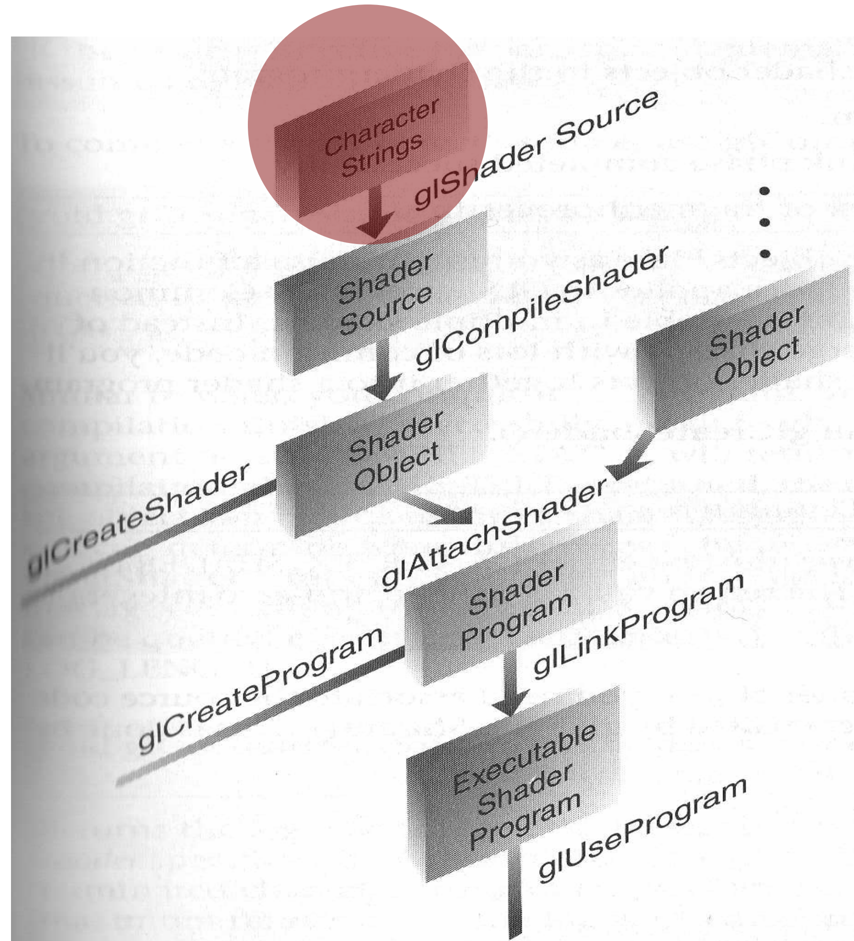
Fragment programs

- Executed once for every fragment
  - Or: “Every fragment is processed by same fragment program that is currently active”
- Implements functionality for
  - Output color to framebuffer
  - Texturing
  - Per-pixel shading
  - Bump mapping
  - Shadows
  - Etc.



Creating shaders in OpenGL

- Sequence of OpenGL API calls to load, compile, link, activate shaders
  - Mostly taken care of in `Shader.java`
- Input is a string that contains shader program
  - String usually read from file
  - Separate files for fragment and vertex shaders

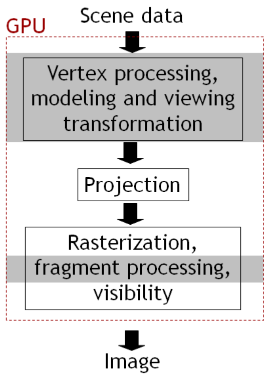


[OpenGL programming guide]

Creating shaders in OpenGL

- You can switch between different shaders during runtime of your application
  - Setup several shaders as shown before
  - Call `glUseProgram(s)` whenever you want to render using a certain shader `s`
  - Shader is active until you call `glUseProgram` with a different shader
- In `jrtr`, this functionality is encapsulated in the `Shader` class

Vertex programs



Vertices with attributes
storage classifier `in`

Coordinates in object space,
additional vertex attributes

From application

Uniform parameters
storage classifier `uniform`

OpenGL state,
application specified
parameters

Vertex
program

To rasterizer

Output
storage classifier `out`

Transformed vertices,
processed vertex attributes

"Hello world" vertex program

- `main()` function is executed for every vertex
- Three storage classifiers: **in**, **out**, **uniform**

```
in vec4 position;           // position, vertex attribute
uniform mat4 projection; // projection matrix, set by host (Java)
uniform mat4 modelview;   // modelview matrix, set by  host (Java)

void main()
{
  gl_Position =            // required, predefined output variable
  projection *             // apply projection matrix
  modelview *              // apply modelview matrix
  position;                // vertex position
}
```

Vertex attributes

- "Data that flows down the pipeline with each vertex"
- Per-vertex data that your application sends to rendering pipeline
- E.g., vertex position, color, normal, texture coordinates
- Declared using `in` storage classifier in your shader code
  - Read-only

Vertex attributes

- Application needs to tell OpenGL which vertex attributes are mapped to which `in` variables
- In host (Java) program, sequence of calls

```
glGenBuffers  // Get reference to OpenGL buffer object
glBindBuffer  // Activate buffer object
glBufferData  // Write data into buffer
glGetAttribLocation   // Get reference of uniform variable
                      //   in shader
glVertexAttribPointer // Link buffer object with uniform
                      // shader variable
glEnableVertexAttribArray // Enable the link
```

- Details see `GLRenderContext.draw`
  - No need to modify it

Uniform parameters

- Parameters that are set by the application, but do not change on a per-vertex basis!
  - Transformation matrices, parameters of light sources, textures
- Will be the same for each vertex until application changes it again
- Declared using `uniform` storage classifier in vertex shader
  - Read-only

Uniform parameters

- To set parameters, use `glGetUniformLocation, glUniform*` in application
  - After shader is active, before rendering
- Example
  - In shader declare
    ```
    uniform float a;
    ```
  - In application, set `a` using
    ```
    GLuint p;
    //… initialize program p
    int i=glGetUniformLocation(p,"a");
    glUniform1f(i, 1.f);
    ```

Output variables

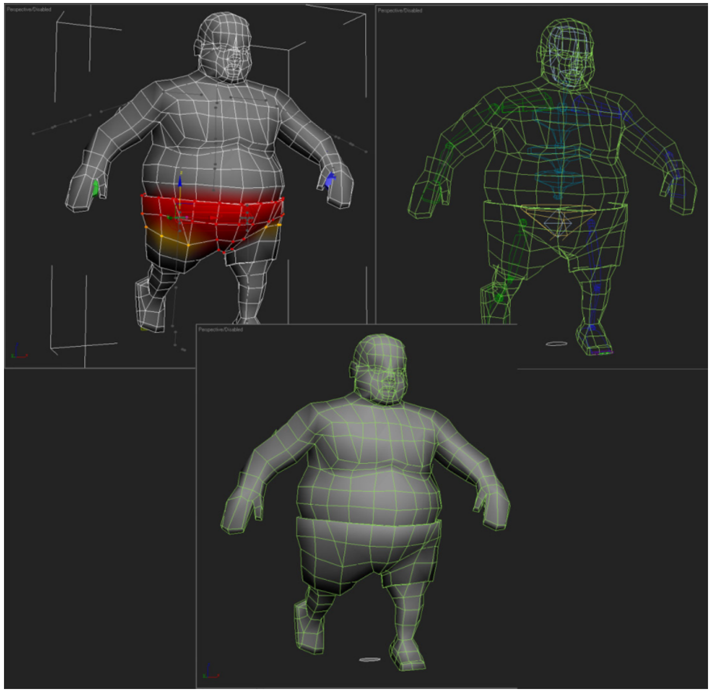
- Required, predefined output variable: homogeneous vertex coordinates
  `vec4 gl_Position`
- Additional user defined outputs
  - Mechanism to send data to the fragment shader
  - Will be interpolated during rasterization
  - Interpolated values accessible in fragment shader (using same variable names)
- Storage classifier **`out`**

Limitations (2014)

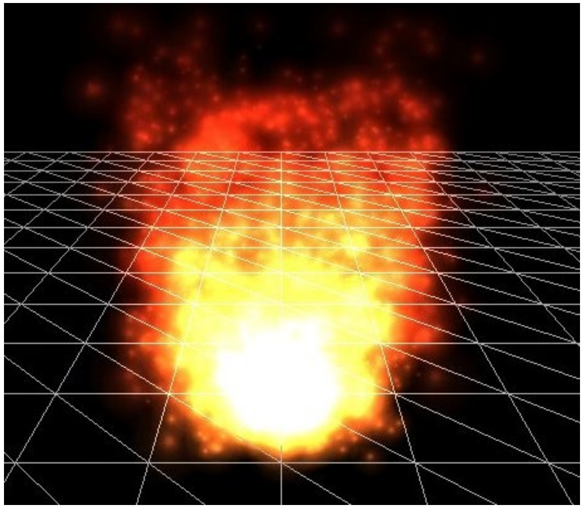
- Cannot write data to any memory accessible directly by application (Java, C++, etc.)
- Cannot pass data between vertices
  - Each vertex is independent
- One vertex in, one vertex out
  - Cannot generate new geometry
  - Note: “Geometry shaders” (not discussed here) can do this

Examples

- Animation
  - Offload as much as possible to the GPU
- Character skinning
- Particle systems
- Water

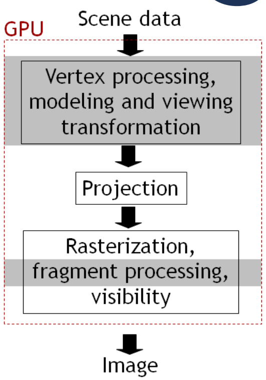


Character skinning



http://www.youtube.com/watch?v=on4H3s-W0NY

Fragment programs



Fragment data
storage classifier `in`

Interpolated vertex attributes, additional fragment attributes

From rasterizer

Uniform parameters
storage classifier `uniform`

OpenGL state, application specified parameters

Fragment program

To fixed framebuffer access functionality (z-buffering, etc.)

Output
storage classifier `out`

Fragment color, depth

Fragment data

- Change for each execution of the fragment program
- Interpolated from vertex output during rasterization
  - Fragment color, texture coordinates, etc.
- Declared as **`in`** variables
  - Need to have same variable name as output (declared as **`out`**) of vertex shader

Uniform parameters

- Work same as in vertex shader
- Typically transformation matrices, parameters of light sources, textures
- Pass from host application via `glGetUniformLocation, glUniform*`

Output variables

- Typically fragment color
- Declared as **`out`**
- Will be written to frame buffer (i.e., output image) automatically

“Hello world” fragment program

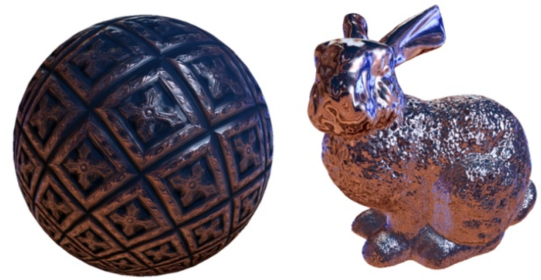
- `main()` function is executed for every fragment
- Draws everything in bluish color

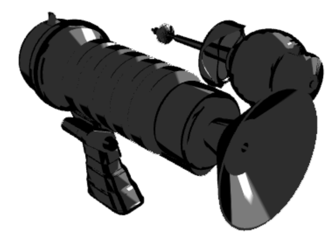
```
out vec4 fragColor;

void main()
{
  fragColor = vec4(0.4,0.4,0.8,1.0);
}
```

Examples

- Per pixel shading as discussed in class
- Bump mapping
- Displacement mapping
- Realistic reflection models
- Cartoon shading
- Shadows
- Etc.
- Most often, vertex and fragment shader work together to achieve desired effect





Limitations (2014)

- Cannot read framebuffer
  - Current pixel color, depth, etc.
- Can only write to framebuffer pixel that corresponds to fragment being processed
  - No random write access to framebuffer
- Number of variables passed from vertex to fragment shader is limited
- Number of application defined uniform parameters is limited

GLSL built in functions and data types

- See OpenGL/GLSL quick reference card
  http://www.khronos.org/files/opengl-quick-reference-card.pdf
- Matrices, vectors, textures
- Matrix, vector operations
- Trigonometric functions
- Geometric functions on vectors
- Texture lookup

Summary

- Shader programs specify functionality of parts of the rendering pipeline
- Written in special shading language (GLSL in OpenGL)
- Sequence of OpenGL calls to compile/activate shaders
- Several types of shaders, discussed here:
  - Vertex shaders
  - Fragment shaders

GLSL main features

- Similar to C, with specialties
- Most important: **`in`**, **`out`**, **`uniform`** storage classifiers
- Parameters of shader (**`uniform`** variables) passed from host application via specific API calls
- Built in vector data types, vector operations
- No pointers, classes, inheritance, etc.

Debugging shaders

- No direct way to debug (setting breakpoints, inspecting values)
- Practical technique
  - Render intermediate steps of your shader
  - Color code information that you want to see (e.g, paint pixel a specific color if you reach certain part of shader code)
- Forum discussions

  http://stackoverflow.com/questions/2508818/how-to-debug-a-glsl-shader

Tutorials and documentation

- OpenGL and GLSL specifications
  http://www.opengl.org/documentation/specs/
- OpenGL/GLSL quick reference card
  http://www.khronos.org/files/opengl-quick-reference-card.pdf
- Learn from example code and use the Ilias forum!

GPGPU programming

- “General purpose” GPU programming
- Special GPU programming languages
  - CUDA
    http://en.wikipedia.org/wiki/CUDA
  - OpenCL
    http://en.wikipedia.org/wiki/OpenCL
- Exploit data parallelism
- SIMT (single instruction multiple threads) programming model
  - Each thread has unique ID
  - Each thread operates on single data item (as opposed to vector of data items in SIMD)
  - Data items accessed via thread ID

A note on transforming normals

- If object-to-camera transformation $\mathbf{M}$ includes shearing, transforming normals using $\mathbf{M}$ does not work
  - Transformed normals are not perpendicular to surface any more
- To avoid problem, need to transform normals by ${\mathbf{M}^{-1}}^T$
- No derivation here, but remember for rotations ${\mathbf{R}^{-1}}^T = \mathbf{R}$