CSE 167:
Introduction to Computer Graphics
Lecture #7: Shaders

Jürgen P. Schulze, Ph.D.
University of California, San Diego
Fall Quarter 2010

Announcements

- Homework project #3 due this Friday, October 15
  - To be presented between 2-4pm in lab 260
  - NEW RULE: Grading ends once list on whiteboard is empty!
- Late submissions for project #2 accepted until this Friday
- Midterm exam: Thursday, Oct 21, 2-3:20pm, WLH 2005
- Midterm tutorial: Tuesday, Oct 19, noon-1:45pm, Atkinson Hall, room 4004
  - Tutors: Jurgen and Phi
  - We will have blank index cards for everybody
- Phi's office hours on Oct 19 and 21 are cancelled

Lecture Overview

- Light Sources
- Shader programming:
  - Vertex shader

Light Sources

- Light sources can have complex properties
  - Geometric area over which light is produced
  - Anisotropy (directionally dependent)
  - Variation in color
  - Reflective surfaces act as light sources (indirect light)

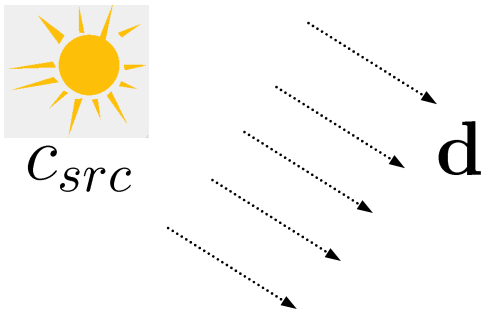
- Interactive rendering is based on simple, standard light sources

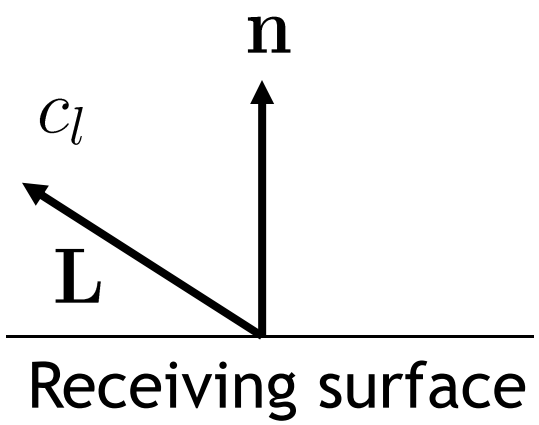
Light Sources

- At each point on surfaces we need to know
  - Direction of incoming light (the $\mathbf{L}$ vector)
  - Intensity of incoming light (the c_l values)
- Standard light sources in OpenGL
  - Directional: from a specific direction
  - Point light source: from a specific point
  - Spotlight: from a specific point with intensity that depends on the direction

Directional Light

- Light from a distant source
  - Light rays are parallel
  - Direction and intensity are the same everywhere
  - As if the source were infinitely far away
  - Good approximation of sunlight
- Specified by a unit length direction vector, and a color





$$\mathbf{L} = -\mathbf{d}$$

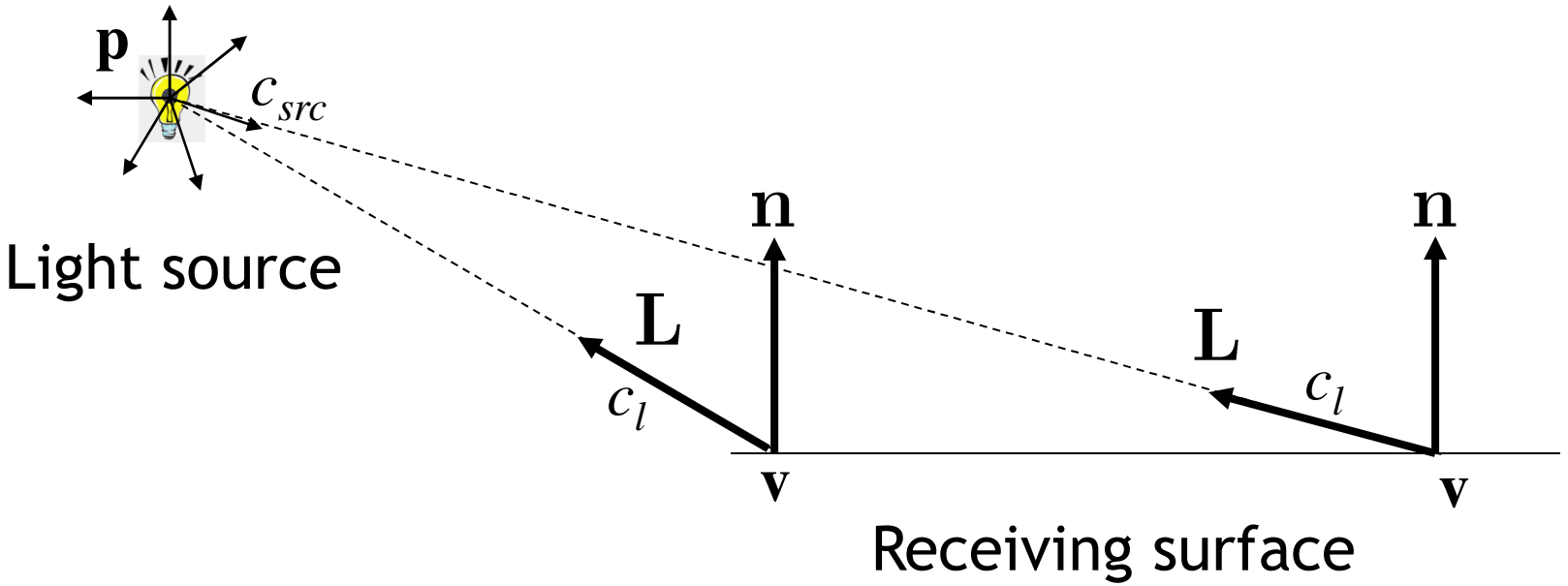
$$c_l = c_{src}$$

Point Lights

- Simple model for light bulbs
- Point that radiates light in all directions equally
  - Light vector varies across the surface
  - Intensity drops off proportionally to the inverse square of the distance from the light
  - Reason for inverse square falloff:
    - Surface area A of sphere:

      $A = 4 \pi r^2$

Point Lights



$$\mathbf{L} = \frac{\mathbf{p} - \mathbf{v}}{\|\mathbf{p} - \mathbf{v}\|}$$

$$c_l = \frac{c_{src}}{\|\mathbf{p} - \mathbf{v}\|^2}$$

Attenuation

- Sometimes, it is desirable to modify the inverse square falloff behavior of point lights
  - Common (OpenGL) model for distance attenuation

$$c_l = \frac{c_{src}}{k_c + k_l \left|\mathbf{p} - \mathbf{v}\right| + k_q \left|\mathbf{p} - \mathbf{v}\right|^2}$$

  - Not physically accurate

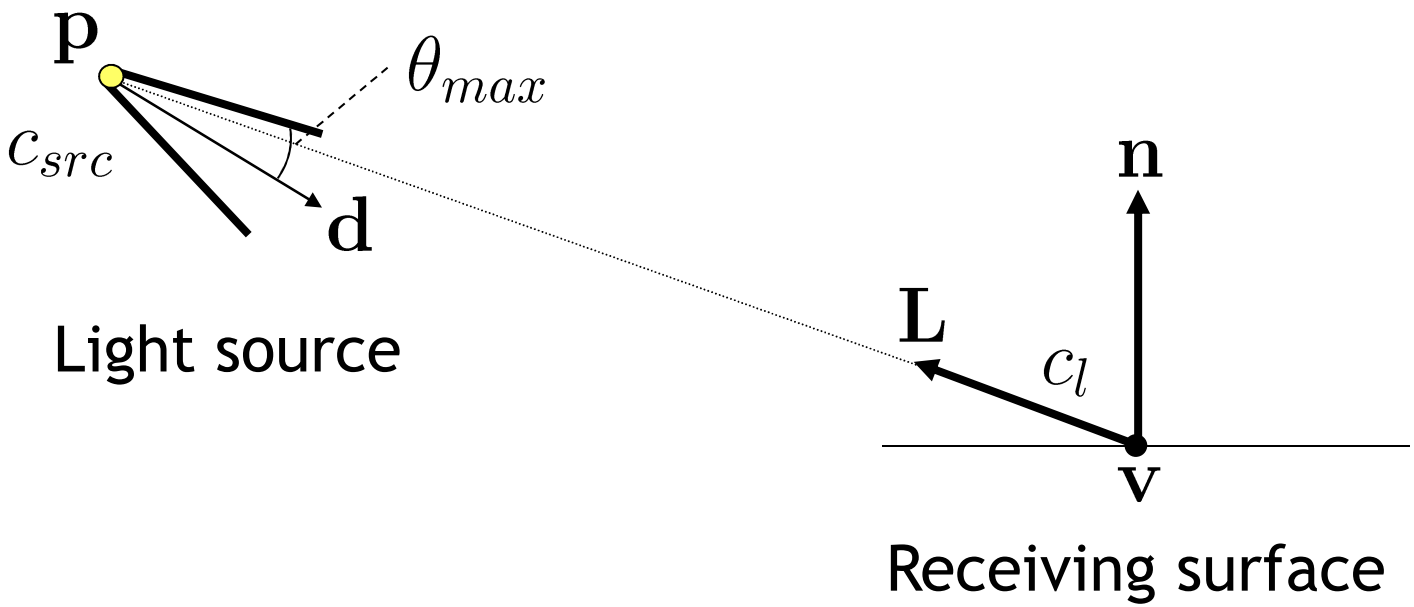
Spotlights

- Like point source, but intensity depends on direction

Parameters

- Position, the location of the source
- Spot direction, the center axis of the light
- Falloff parameters
  - Beam width (cone angle)
  - The way the light tapers off at edges of the beam (cosine exponent)

Spotlights

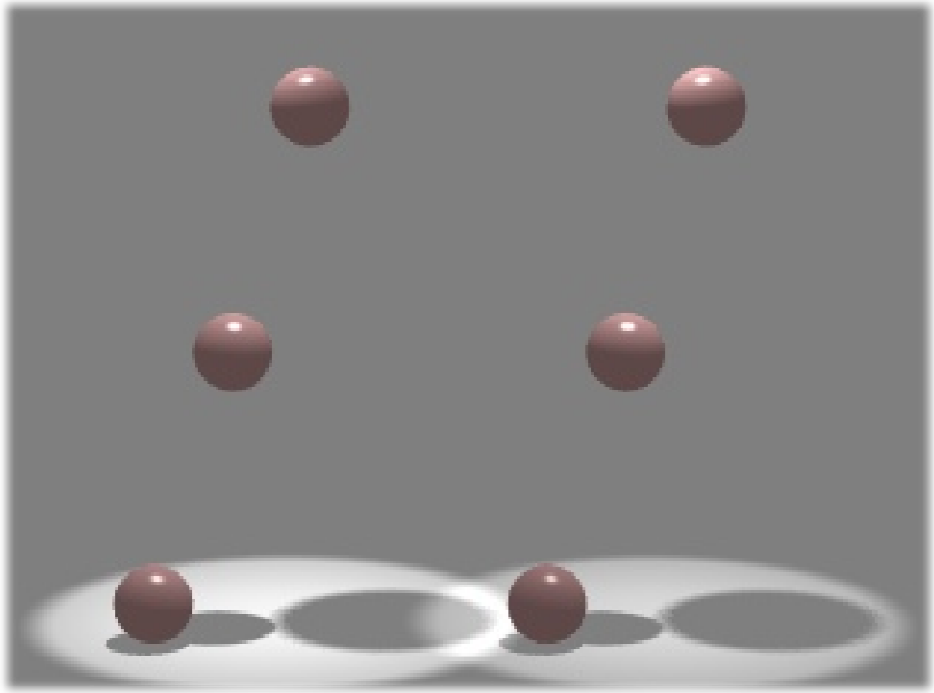


$$\mathbf{L} = \frac{\mathbf{p} - \mathbf{v}}{\|\mathbf{p} - \mathbf{v}\|}$$

$$c_l = \begin{cases} 0 & \text{if} \quad -\mathbf{L} \cdot \mathbf{d} \leq \cos(\theta_{max}) \\ c_{src} \left(-\mathbf{L} \cdot \mathbf{d}\right)^f & \text{otherwise} \end{cases}$$

Spotlights

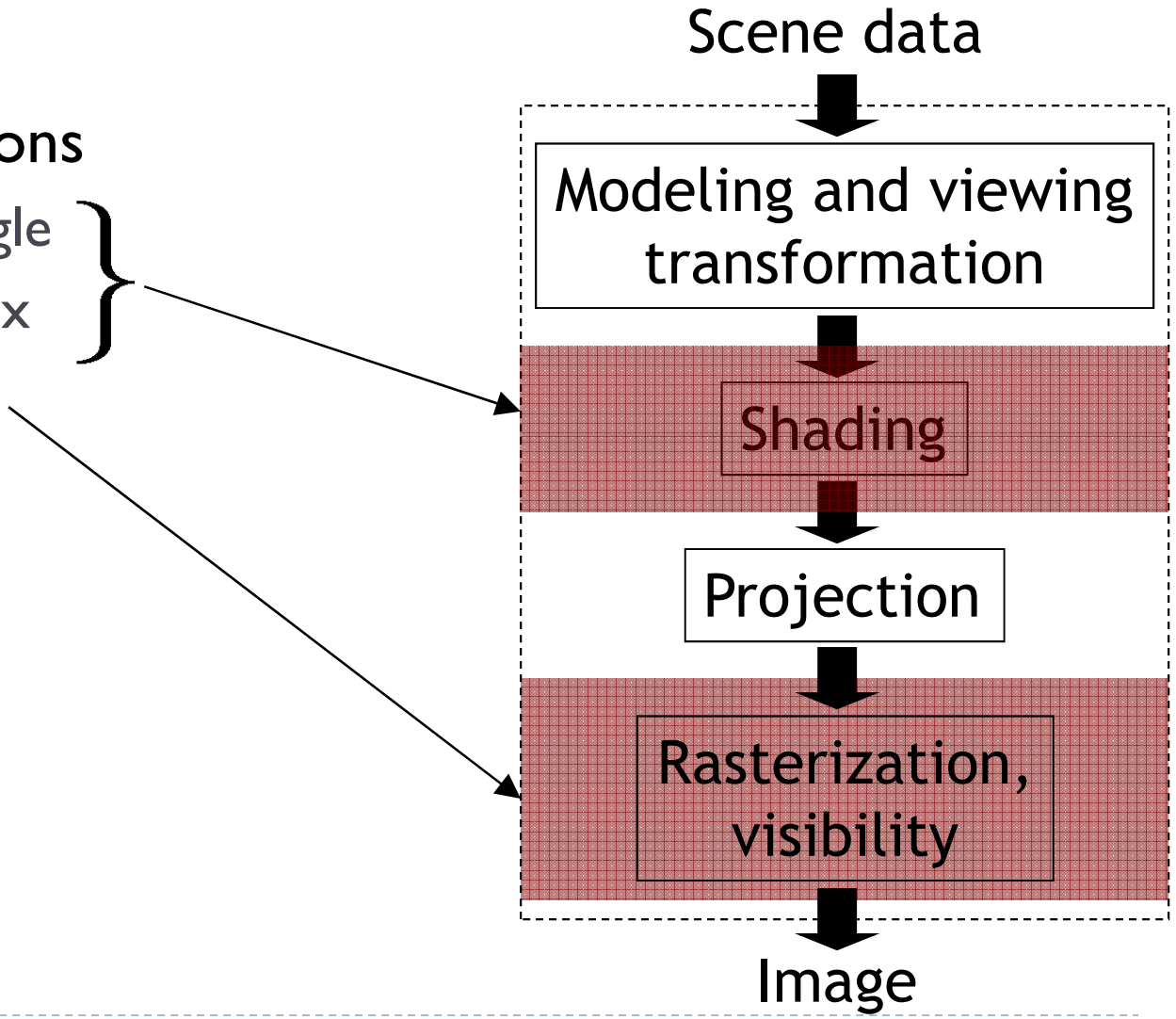
Photograph of spotlight



Spotlights in OpenGL

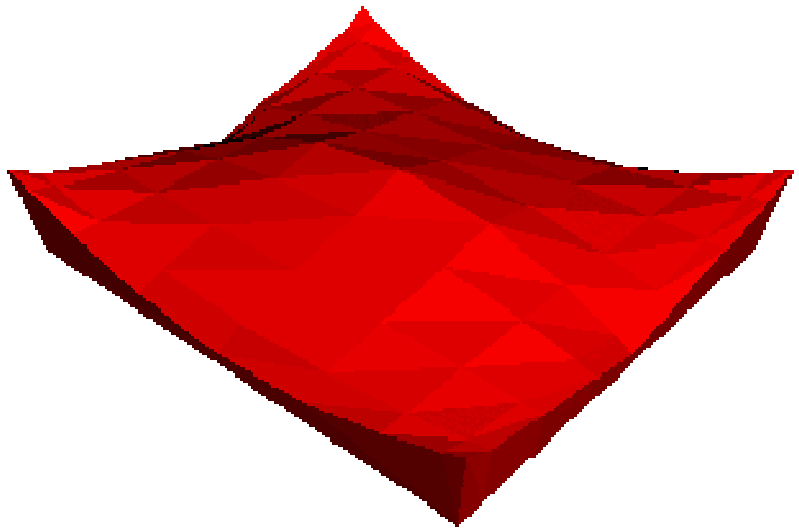
Per-Triangle, -Vertex, -Pixel Shading

- Shading operations
  - Once per triangle
  - Once per vertex
  - Once per pixel



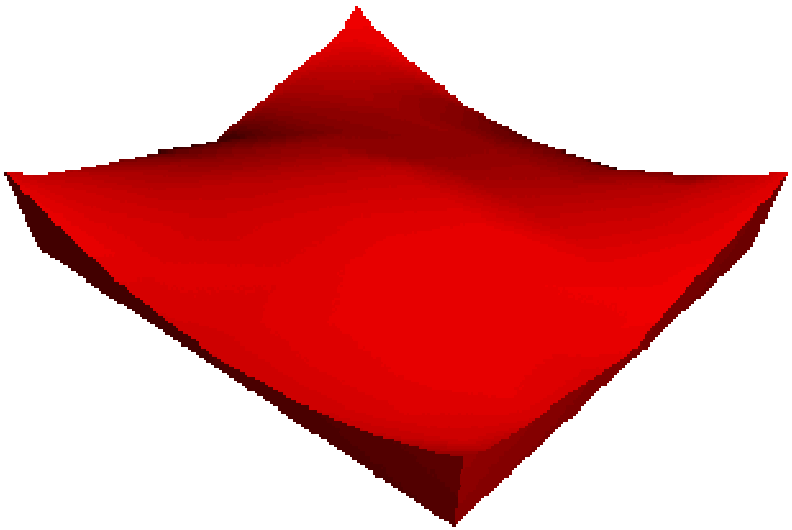
Per-Triangle Shading

- Known as *flat shading*
- Evaluate shading once per triangle
- Advantages
  - Fast
- Disadvantages
  - Faceted appearance



Per-Vertex Shading

- Known as *Gouraud shading* (Henri Gouraud 1971)
- Interpolate vertex colors across triangles
- OpenGL default
- Advantages
  - Fast
  - Smoother than flat shading
- Disadvantages
  - Problems with small highlights

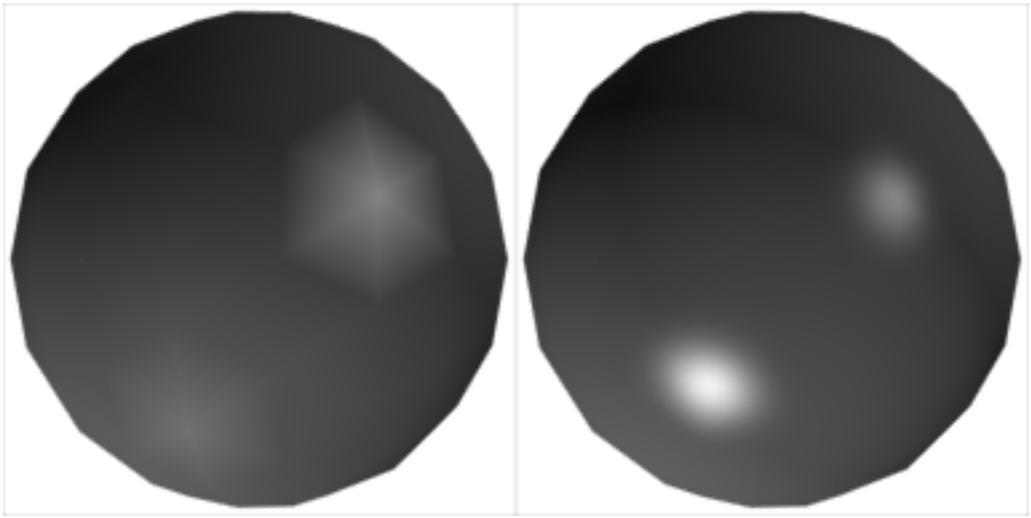


Per-Pixel Shading

- Also known as *Phong interpolation* (not to be confused with *Phong illumination* model)
  - Rasterizer interpolates normals across triangles
  - Illumination model evaluated at each pixel
  - Implemented using *fragment shaders* (later today)
- Advantages
  - Higher quality than Gouraud shading
- Disadvantages
  - Much slower

Gouraud vs. Per-Pixel Shading

- Gouraud has problems with highlights
- More triangles would improve result, but impact frame rate



Gouraud Per-pixel

Shading in OpenGL

```
// Somewhere in the initialization part of your
// program…
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);

// Make sure vertex colors are used as material properties
glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT, GL_DIFFUSE);
glColorMaterial(GL_FRONT, GL_SPECULAR);

// Create light components
Glfloat ambientLight[] = { 0.2f, 0.2f, 0.2f, 1.0f };
Glgloat diffuseLight[] = { 0.8f, 0.8f, 0.8, 1.0f };
Glfloat specularLight[] = { 0.5f, 0.5f, 0.5f, 1.0f };
Glfloat position[] = { -1.5f, 1.0f, -4.0f, 1.0f };

// Assign created components to GL_LIGHT0
glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight);
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight);
glLightfv(GL_LIGHT0, GL_SPECULAR, specularLight);
glLightfv(GL_LIGHT0, GL_POSITION, position);
```

Shading in OpenGL

- Shading computations (diffuse, specular, ambient) are performed automatically (unless you use shader programs)

Shading in OpenGL

- Need to provide per vertex normals
- Shading is performed in camera space
  - Position, direction of light sources is transformed by GL_MODELVIEW matrix
- If light sources should be fixed relative to objects
  - Set GL_MODELVIEW to desired object-to-camera transform
  - Choose object space coordinates for light position
  - Will be transformed using current GL_MODELVIEW
- Lots of details, highly recommend OpenGL programming guide
  - http://glprogramming.com/red/chapter05.html
  - http://www.falloutsoftware.com/tutorials/gl/gl8.htm

Transforming Normals

- If the object-to-camera transformation **M** includes shearing or scaling, transforming normals using **M** does not work:
  - Transformed normals are not perpendicular to surfaces any more
- To avoid the problem, we need to transform the normals differently:
  - by transforming the end points of the normal vectors separately
  - or using
- Find deriv$\mathbf{M}^{-1^T}$n-line at:
  - http://www.oocities.com/vmelkon/transformingnormals.html
- OpenGL does this automatically for us on the GPU

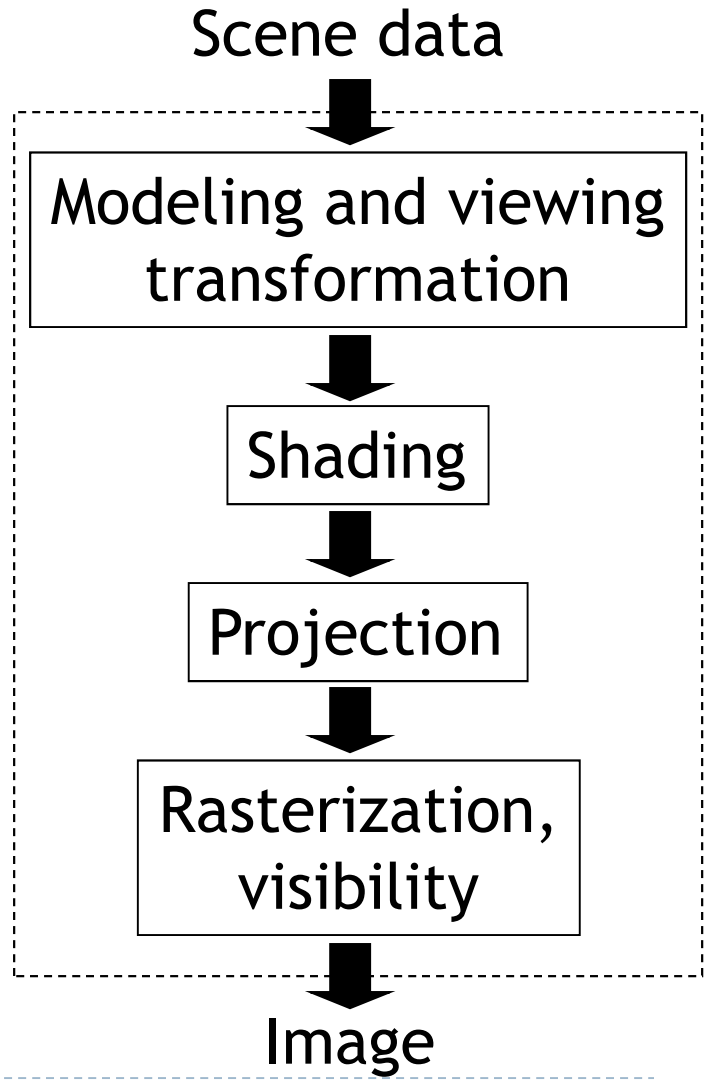
Lecture Overview

- Light Sources
- Shader programming:
  - Vertex shader

Configurable Pipeline

Before 2002:

- APIs (OpenGL, Direct3D) to configure the rendering pipeline
- Enable/disable functionality
  - E.g., lighting, texturing
- Set parameters for given functionality
  - E.g., light direction, texture blending mode



Configurable Pipeline

Disadvantages

- Restricted to preset functionality
  - Limited types of light sources (directional, point, spot)
  - Limited set of reflection models (ambient, diffuse, Phong)
  - Limited use of texture maps
- More flexibility desired for more photorealistic effects

Demo

- NVIDIA Time Machine
- http://www.nzone.com/object/nzone_timemachinedemo_home.html

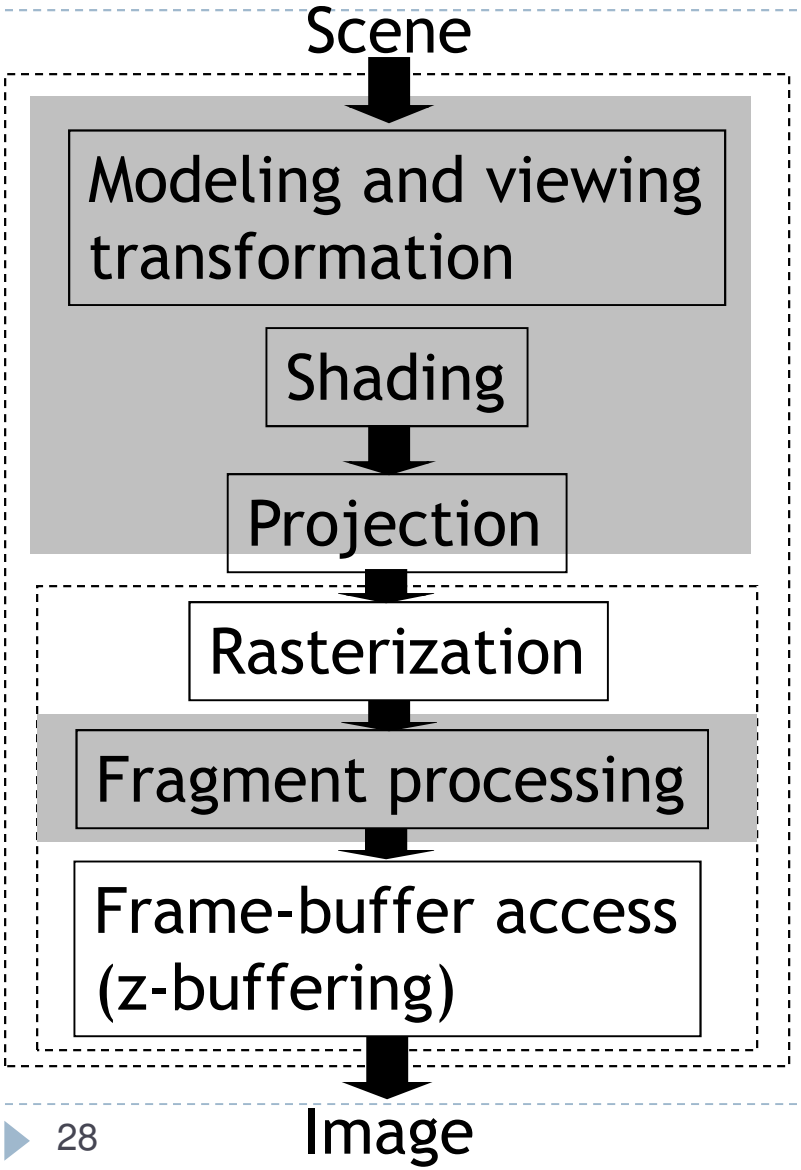
Programmable Pipeline

- Replace functionality in parts of the pipeline by user specified programs
- Called shaders, or shader programs
- Not all functionality in the pipeline is programmable

Shader Programs

- Written in a shading language
  - Cg: early shading language by NVidia
  - Shading languages today:
    - GLSL for OpenGL (GL shading language)
    - HLSL for DirectX (high level shading language)
  - Syntax similar to C
- Novel, quickly changing technology
- Driven by more and more flexible GPUs

Programmable Pipeline



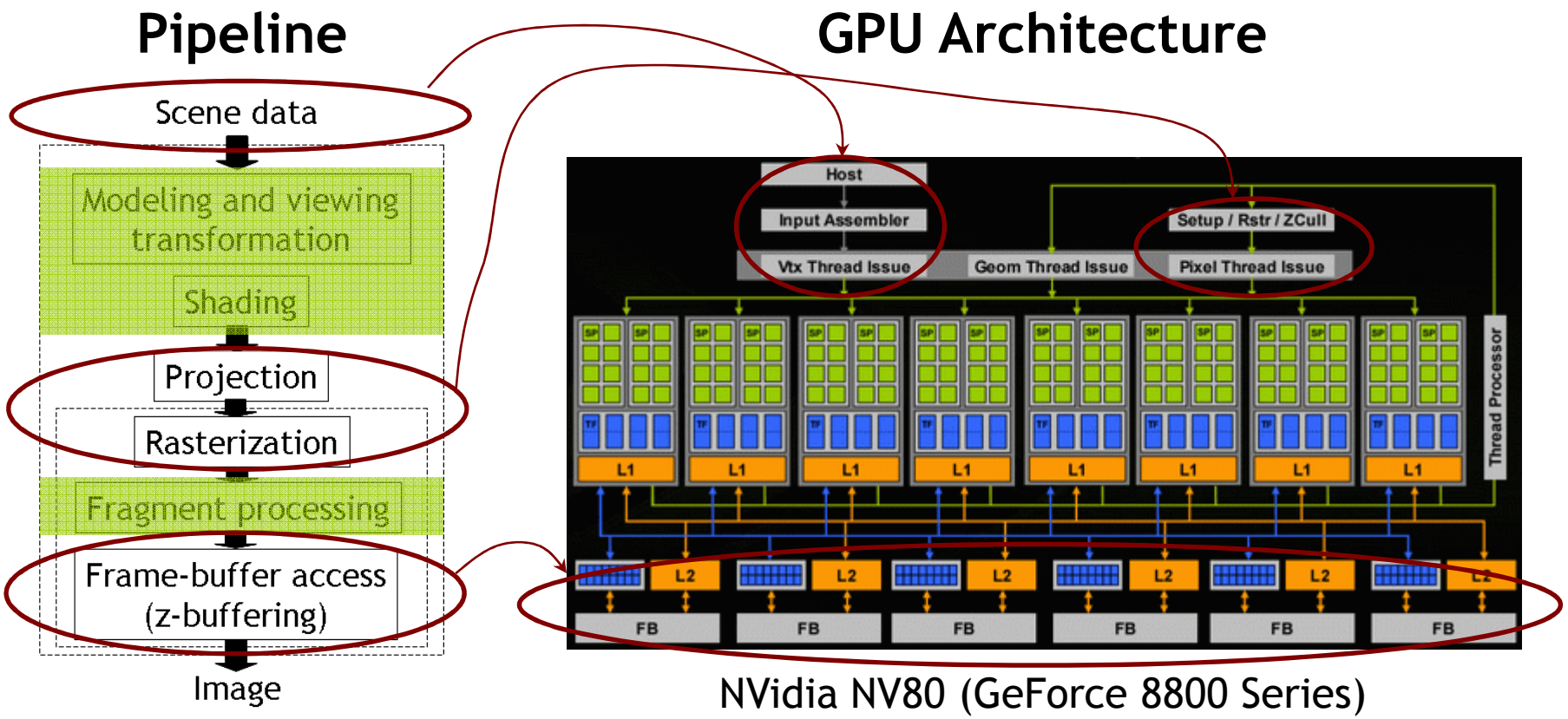
Vertex program

Executed once for each vertex

Fragment program

Executed once for each fragment (= pixel location in a triangle)

GPU Architecture



NVidia NV80 (GeForce 8800 Series)

128 stream processors

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

Programmable Pipeline

Not programmable:

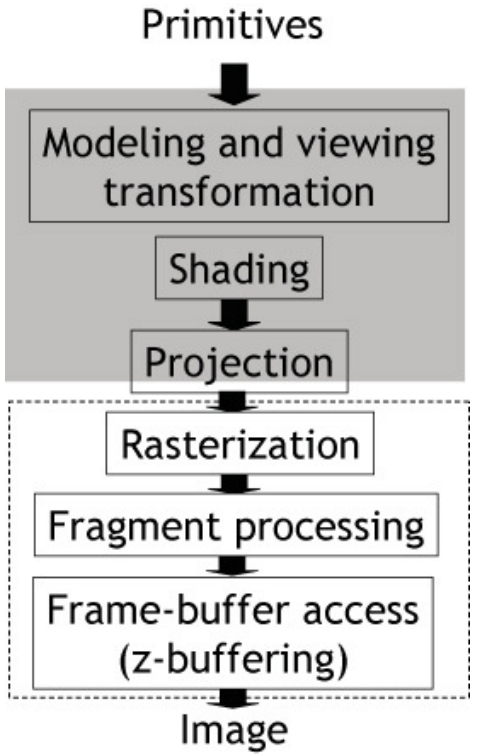
- Projective division
- Rasterization
  - Determination of which pixels lie inside a triangle
  - Vertex attribute interpolation (color, texture coordinates)
- Access to frame buffer
  - Texture filtering
  - Z-buffering
  - Frame buffer blending

Shader Programming

- Application programmer can provide:
  - No shaders, standard OpenGL functions are executed
  - Vertex shader only
  - Fragment shader only
  - Vertex and fragment shaders
- Each shader is a separate piece of code in a separate text file
- Output of vertex shader is interpolated at each fragment and accessible as input to fragment shader

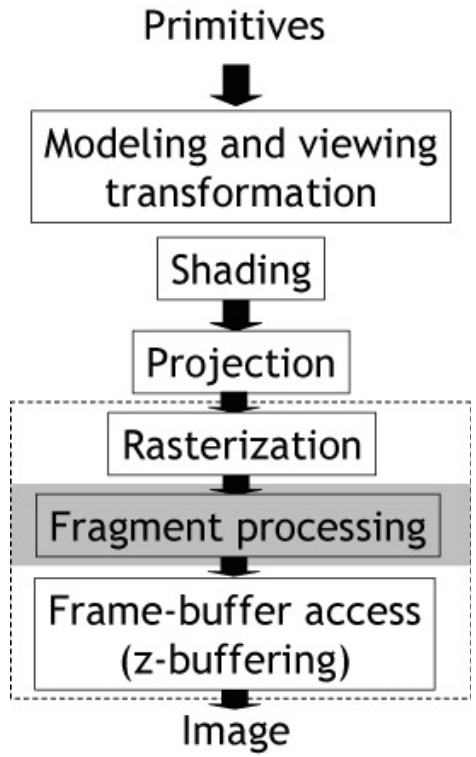
Vertex Programs

- Executed once for every vertex
- Replaces functionality for
  - Model-view, projection transformation
  - Per-vertex shading
- If you use a vertex program, you need to implement this functionality in the program
- Vertex shader often used for animation
  - Characters
  - Particle systems

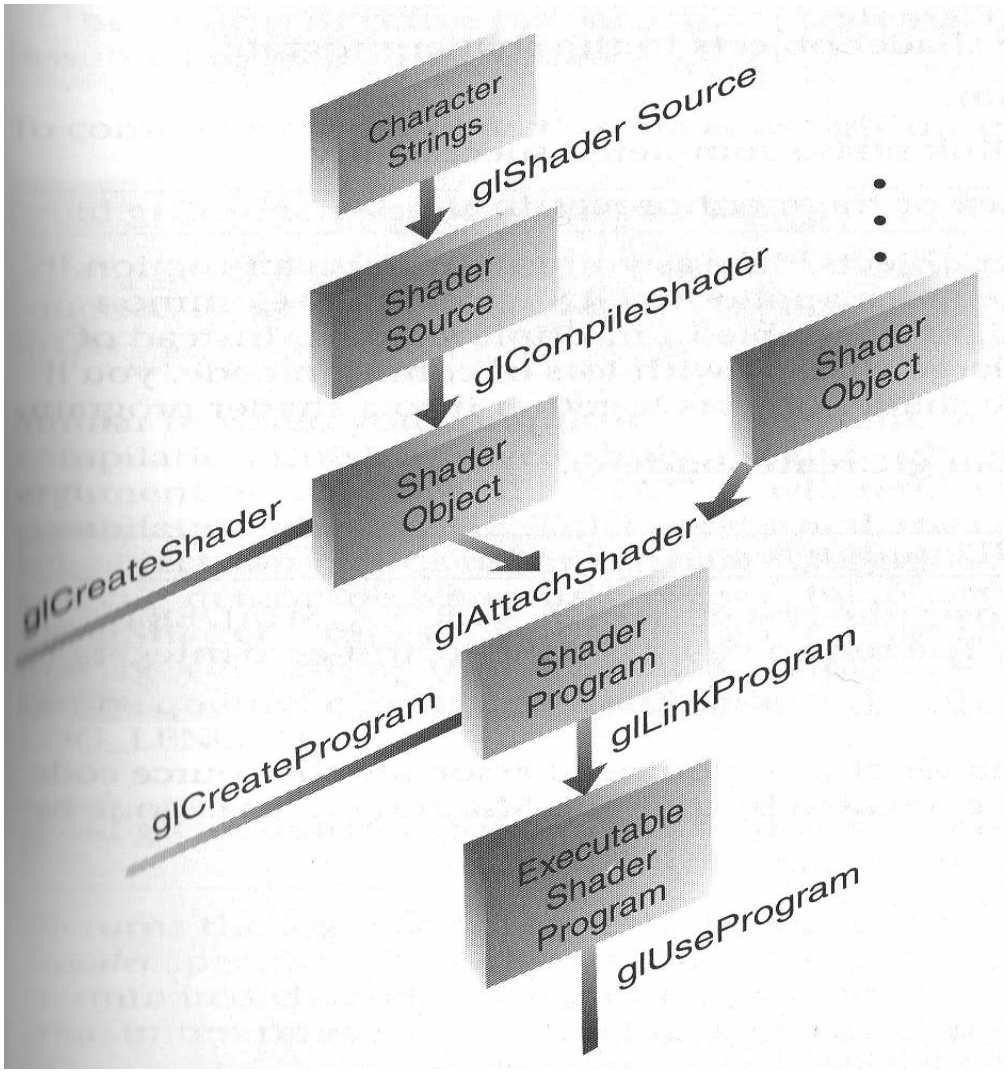


Fragment Programs

- Executed once for every fragment
- Implements functionality for
  - Texturing
  - Per pixel effects
  - Per pixel shading
  - Bump mapping
  - Shadows
  - Blending
  - Look-up tables
  - Etc.



Creating Shaders in OpenGL

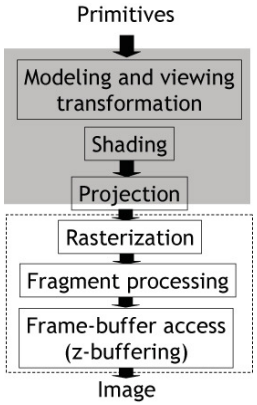


Source: OpenGL Programming Guide

Lecture Overview

- Light Sources
- Shader programming:
  - Vertex shader

Vertex Programs



Vertex attributes
Coordinates in object space, additional vertex attributes

From application

Uniform parameters
OpenGL state, application specified parameters

Vertex program

To rasterizer

Transformed vertices, processed vertex attributes

Types of Input Data

- Vertex attributes
  - Change for each execution of the vertex program
  - Predefined OpenGL attributes (color, position, etc.)
  - User defined attributes
- Uniform parameters
  - Normally the same for all vertices
  - OpenGL state variables
  - Application defined parameters

Vertex Attributes

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

Vertex Attributes

- OpenGL vertex attributes accessible through predefined variables

```
attribute vec4 gl_Vertex;
attribute vec3 gl_Normal;
attribute vec4 gl_Color;
```

  etc.

- Optional user defined attributes

OpenGL State Variables

- Provide access to state of rendering pipeline, which you set through OpenGL calls in application
- Predefined variables

```
uniform mat4 gl_ModelViewMatrix;
uniform mat4 gl_ModelViewProjectionMatrix;
uniform mat4 gl_ProjectionMatrix;
uniform gl_LightSourceParameters
        gl_LightSource[gl_MaxLights];
```

  etc.
- Declared using `uniform` storage classifier
  - Read-only

Uniform Parameters

- Parameters that are set by the application
- Should not change frequently
  - Not on a per-vertex basis!
- Will be the same for each vertex until application changes it again
- Declared using `uniform` storage classifier
  - Read-only

Uniform Parameters

- To access, use `glGetUniformLocation, glUniform*` in application
- 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 output: homogeneous vertex coordinates
  `vec4 gl_Position`
- **`varying`** outputs
  - Mechanism to send data to the fragment shader
  - Will be interpolated during rasterization
  - Interpolated values accessible in fragment shader (using same variable name)
- Predefined `varying` outputs
  ```
  varying vec4 gl_FrontColor;
  varying vec4 gl_TexCoord[ ];
  ```
  etc.
- User defined `varying` outputs

Output Variables

Note

- Any predefined output variable that you do not write will assume the value of the current OpenGL state
- E.g., your vertex shader does not write
  `varying vec4 gl_TexCoord[ ]`
  - Your fragment shader may still read it
  - The value will be the current OpenGL state

“Hello world” Vertex Program

- `main()` function is executed for every vertex
- Use predefined variables

```
void main()
{
  gl_Position =           // required output
  gl_ProjectionMatrix *   // predefined uniform
  gl_ModelViewMatrix *    // predefined uniform
  gl_Vertex;              // predefined attribute
}
```

- Alternatively, use `gl_ModelViewProjectionMatrix` or `ftransform()`

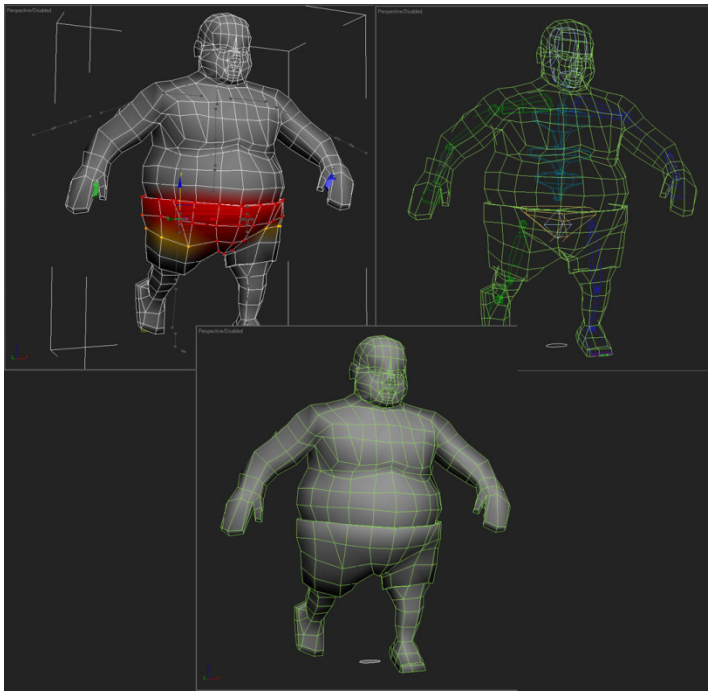
Vertex Programs

Limitations

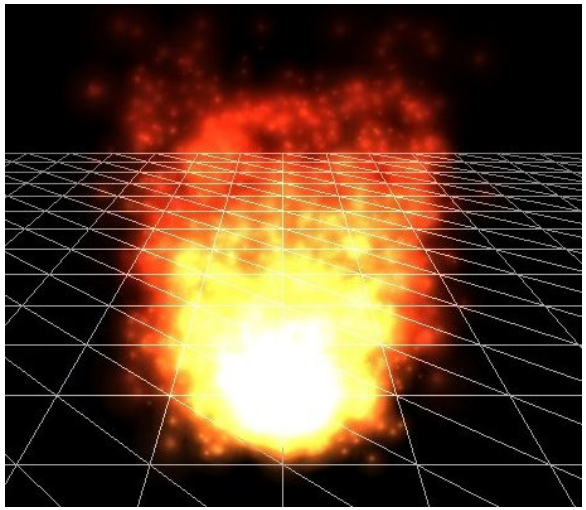
- Cannot write data to memory accessible by application
  - Workaround: CUDA
- Cannot pass data between vertices
  - Each vertex is independent
- Except for latest graphics cards:
  For each incoming vertex there is one outgoing vertex
  - Cannot generate new geometry
  - Newest cards have Geometry Shader

Examples

- Character skinning
- Particle systems
- Water



Character skinning



Particle system

Next Lecture

- Fragment Shaders
- Texturing