CSE 167:
Introduction to Computer Graphics
Lecture #9: GLSL

Jürgen P. Schulze, Ph.D.
University of California, San Diego
Fall Quarter 2011

Announcements

- Homework assignment #4 due Friday, Oct 28
  - To be presented after 1:30pm in lab 260
- Late submissions for project #3 accepted until tomorrow (grading starts at 2pm)
- Midterm exam: Thursday, Oct 27, 2-3:20pm
- Midterm tutorial: Tuesday, Oct 25, 3:45-5pm, Atkinson Hall, room 4004

Lecture Overview

- Shader programming
  - Vertex shaders
  - Fragment shaders
  - GLSL
- Texturing
  - Overview

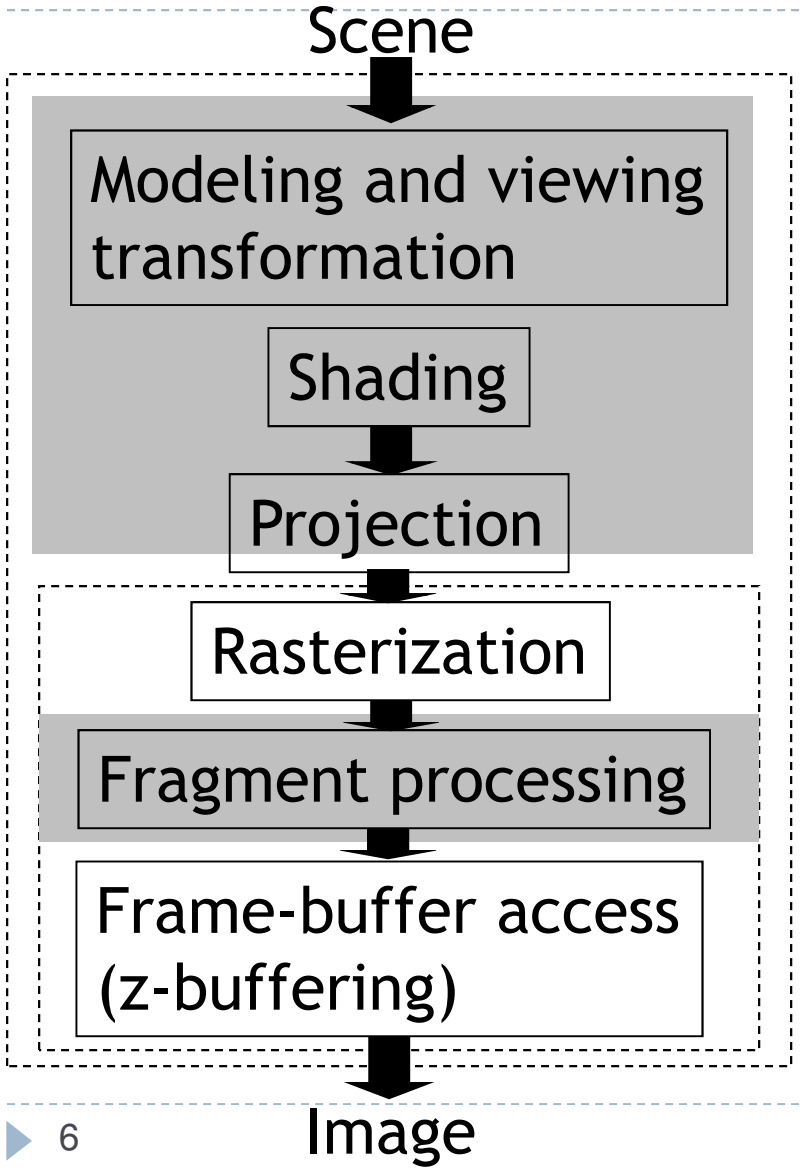
Programmable Pipeline

- 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
- Development driven by more and more flexible GPUs

Programmable Pipeline



Vertex program

Executed once for each vertex

Fragment program

Executed once for each fragment (rasterized pixel)

Programmable Pipeline

Not programmable:

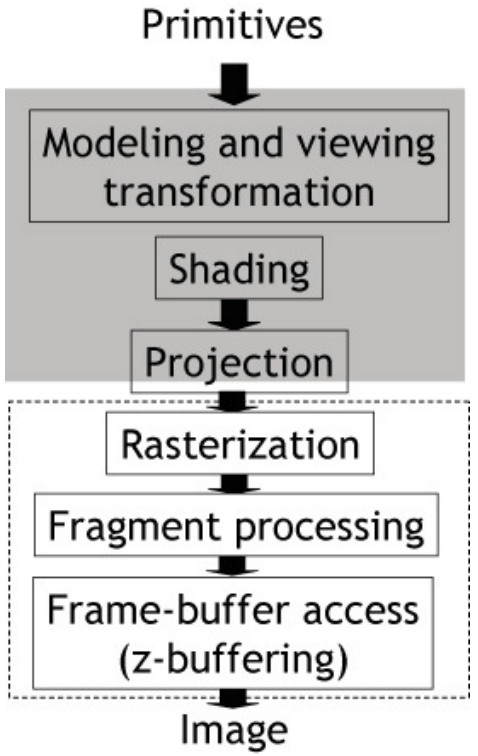
- Projective division
- Rasterization
  - Determination of which pixels lie inside a triangle
  - Vertex attribute interpolation (color, texture coordinates)
- Access to frame buffer
  - Texture filtering
  - Z-buffering
  - Frame buffer blending

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

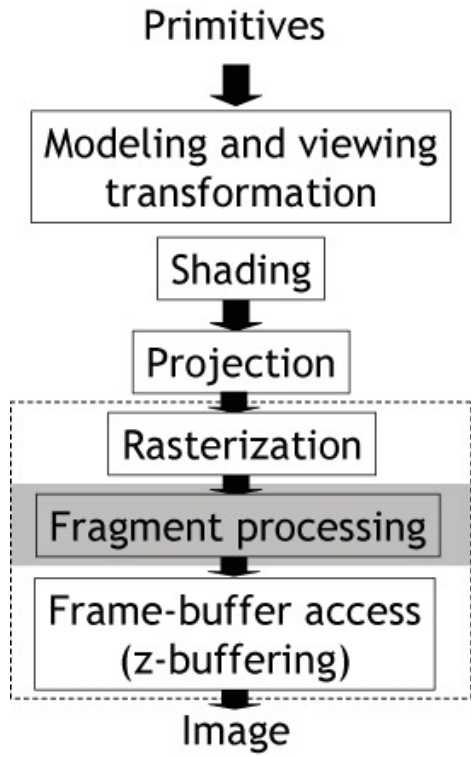
Vertex Programs

- 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
  - Advanced Texturing
  - Per pixel shading
  - Bump mapping



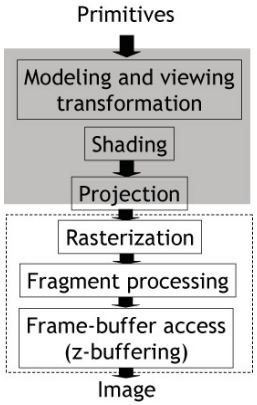
Summary

- Most often, vertex and fragment shaders work together to achieve a desired effect

Lecture Overview

- Shader programming
  - Vertex shaders
  - Fragment shaders
  - GLSL
- Texturing
  - Overview

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 overwrite will assume the value of the current OpenGL state
- E.g., your vertex shader does not use
  `varying vec4 gl_TexCoord[ ]`
  - Your fragment shader may still read it, but the value will be defiled by the OpenGL state

Vertex Programs

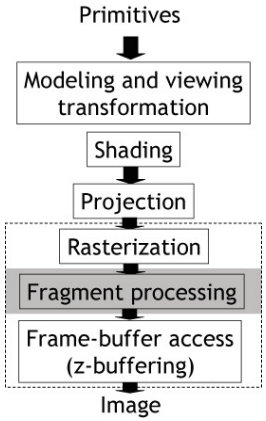
Limitations

- Cannot write data to memory accessible by application
  - Workaround: CUDA Direct Memory Access
- Cannot pass data between vertices
  - Each vertex is independent
- Except for latest graphics cards:
  For each incoming vertex there is one outgoing vertex
  - Generally cannot generate new geometry
  - Exception: Geometry Shader (since OpenGL 3.2)

Lecture Overview

- Shader programming
  - Vertex shaders
  - Fragment shaders
  - GLSL
- Texturing
  - Overview

Fragment Programs



Fragment data

Interpolated vertex attributes, additional fragment attributes

From rasterizer

Uniform parameters

OpenGL state, application specified parameters

Fragment program

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

Fragment color, depth

Types of Input Data

Fragment data

- Change for each execution of the fragment program
- Interpolated from vertex data during rasterization, `varying` variables

- Interpolated fragment color, texture coordinates
- Standard OpenGL fragment data accessible through predefined variables

```
varying vec4 gl_Color;
varying vec4 gl_TexCoord[ ];
```

etc.

- Note `varying` storage classifier, read-only
- User defined data possible, too

Types of Input Data

Uniform parameters

- Same as in vertex shader
- OpenGL state
- Application defined parameters
  - Use `glGetUniformLocation, glUniform*` in application

Output Variables

- Predefined outputs
  - `gl_FragColor`
  - `gl_FragDepth`
- OpenGL writes these to the frame buffer
- Result is undefined if you do not write these variables

Fragment Programs

Limitations

- Cannot read frame buffer (color, depth)
- Can only write to frame buffer pixel that corresponds to fragment being processed
  - No random write access to frame buffer
- Limited number of `varying` variables passed from vertex to fragment shader
- Limited number of application-defined uniform parameters

“Hello World” Fragment Program

- `main()` function is executed for every fragment
- Use predefined variables
- Draws every pixel in green color

```
void main()
{
  gl_FragColor = vec4(0.0,1.0,0.0,1.0);
}
```

Lecture Overview

- Shader programming
  - Vertex shaders
  - Fragment shaders
  - GLSL
- Texturing
  - Overview

GLSL Main Features

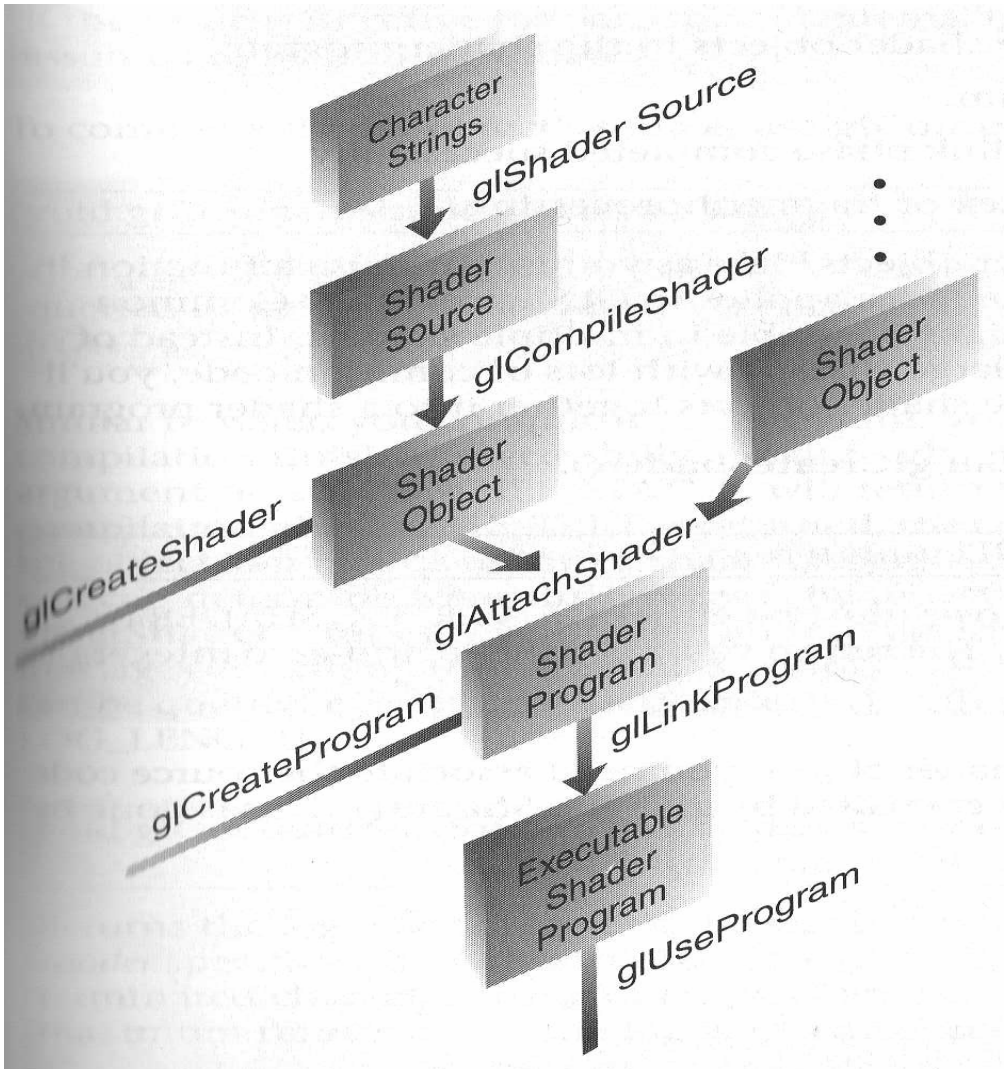
- Similar to C language
- `attribute, uniform, varying` storage classifiers
- Set of predefined variables
  - Access per vertex, per fragment data
  - Access OpenGL state
- Built-in vector data types, vector operations
- No pointers
- No direct access to data, variables in your C++ code

Example: Per-pixel Diffuse Lighting

```
// Vertex shader, stored in file diffuse.vert
varying vec3 normal, lightDir;
void main()
{
  lightDir = normalize(vec3(gl_LightSource[0].position));
  normal = normalize(gl_NormalMatrix * gl_Normal);
  gl_Position = ftransform();
}
```

```
// Pixel shader, stored in file diffuse.frag
varying vec3 normal, lightDir;
void main()
{
        gl_FragColor =
       gl_LightSource[0].diffuse *
       max(dot(normalize(normal), normalize(lightDir)),0.0) *
       gl_FrontMaterial.diffuse;
}
```

Creating Shaders in OpenGL



Source: OpenGL Programming Guide

GLSL Quick Reference Guide

OpenGL® Shading Language (GLSL) Quick Reference Guide

Describes GLSL version 1.10, as included in OpenGL v2.0, and specified by "The OpenGL® Shading Language", version 1.10.59. Section and page numbers refer to that version of the spec.

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

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

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

PREPROCESSOR (3.3 p9)

#
#define
#undef
#if
#ifdef
#ifndef
#else
#elif
#endif
#error
#pragma
#line

__LINE__
__FILE__
__VERSION__

GLSL version declaration and extensions protocol:

#version
default is "#version 110" (3.3 p11)
#extension {*name* | all} : {require | enable | warn | disable}
default is "#extension all : disable" (3.3 p11)

BUILT-IN FUNCTIONS

Key:
vec = vec2 | vec3 | vec4
mat = mat2 | mat3 | mat4
ivec = ivec2 | ivec3 | ivec4
bvec = bvec2 | bvec3 | bvec4
genType = float | vec2 | vec3 | vec4

Angle and Trigonometry Functions (8.1 p51)
genType sin(genType)
genType cos(genType)
genType tan(genType)

genType asin(genType)
genType acos(genType)
genType atan(genType, genType)
genType atan(genType)

genType radians(genType)
genType degrees(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)

Common Functions (8.3 p52)
genType abs(genType)
genType ceil(genType)
genType clamp(genType, genType, genType)
genType clamp(genType, float, float)
genType floor(genType)
genType fract(genType)
genType max(genType, genType)
genType max(genType, float)
genType min(genType, genType)
genType min(genType, float)
genType mix(genType, genType, genType)
genType mix(genType, genType, float)
genType mod(genType, genType)
genType mod(genType, float)
genType sign(genType)
genType smoothstep(genType, genType, genType)
genType smoothstep(float, float, genType)
genType step(genType, genType)
genType step(float, genType)

Geometric Functions (8.4 p54)
vec4 ftransform() *Vertex ONLY*
vec3 cross(vec3, vec3)
float distance(genType, genType)
float dot(genType, genType)
genType faceforward(genType *V*, genType *I*, genType *N*)
float length(genType)
genType normalize(genType)
genType reflect(genType *I*, genType *N*)
genType refract(genType *I*, genType *N*, float *eta*)

Fragment Processing Functions (8.8 p58) *Fragment ONLY*
genType dFdx(genType)
genType dFdy(genType)
genType fwidth(genType)

Matrix Functions (8.5 p55)
mat matrixCompMult(mat, mat)

Vector Relational Functions (8.6 p55)
bool all(bvec)
bool any(bvec)
bvec equal(vec, vec)
bvec equal(ivec, ivec)
bvec equal(bvec, bvec)
bvec greaterThan(vec, vec)
bvec greaterThan(ivec, ivec)
bvec greaterThanEqual(vec, vec)
bvec greaterThanEqual(ivec, ivec)
bvec lessThan(vec, vec)
bvec lessThan(ivec, ivec)
bvec lessThanEqual(vec, vec)
bvec lessThanEqual(ivec, ivec)
bvec not(bvec)
bvec notEqual(vec, vec)
bvec notEqual(ivec, ivec)
bvec notEqual(bvec, bvec)

Texture Lookup Functions (8.7 p56)
Optional bias term is Fragment ONLY
vec4 texture1D(sampler1D, float [,float *bias*])
vec4 texture1DProj(sampler1D, vec2 [,float *bias*])
vec4 texture1DProj(sampler1D, vec4 [,float *bias*])

vec4 texture2D(sampler2D, vec2 [,float *bias*])
vec4 texture2DProj(sampler2D, vec3 [,float *bias*])
vec4 texture2DProj(sampler2D, vec4 [,float *bias*])

vec4 texture3D(sampler3D, vec3 [,float *bias*])
vec4 texture3DProj(sampler3D, vec4 [,float *bias*])

vec4 textureCube(samplerCube, vec3 [,float *bias*])

vec4 shadow1D(sampler1DShadow, vec3 [,float *bias*])
vec4 shadow2D(sampler2DShadow, vec3 [,float *bias*])
vec4 shadow1DProj(sampler1DShadow, vec4 [,float *bias*])
vec4 shadow2DProj(sampler2DShadow, vec4 [,float *bias*])

Texture Lookup Functions with LOD (8.7 p56)
Vertex ONLY; ensure GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS > 0
vec4 texture1DLod(sampler1D, float, float *lod*)
vec4 texture1DProjLod(sampler1D, vec2, float *lod*)
vec4 texture1DProjLod(sampler1D, vec4, float *lod*)

vec4 texture2DLod(sampler2D, vec2, float *lod*)
vec4 texture2DProjLod(sampler2D, vec3, float *lod*)
vec4 texture2DProjLod(sampler2D, vec4, float *lod*)
vec4 texture3DProjLod(sampler3D, vec4, float *lod*)

vec4 textureCubeLod(samplerCube, vec3, float *lod*)

vec4 shadow1DLod(sampler1DShadow, vec3, float *lod*)
vec4 shadow2DLod(sampler2DShadow, vec3, float *lod*)
vec4 shadow1DProjLod(sampler1DShadow, vec4, float *lod*)
vec4 shadow2DProjLod(sampler2DShadow, vec4, float *lod*)

Noise Functions (8.9 p60)
float noise1(genType)
vec2 noise2(genType)
vec3 noise3(genType)
vec4 noise4(genType)

http://www.opengl.org/sdk/libs/OpenSceneGraph/glsl_quickref.pdf

GLSL Quick Reference Guide

VERTEX SHADER VARIABLES

Special Output Variables (7.1 p42) *access=RW*
```
vec4  gl_Position;      shader must write
float gl_PointSize;     enable GL_VERTEX_PROGRAM_POINT_SIZE
vec4  gl_ClipVertex;
```

Attribute Inputs (7.3 p44) *access=RO*
```
attribute vec4  gl_Vertex;
attribute vec3  gl_Normal;
attribute vec4  gl_Color;
attribute vec4  gl_SecondaryColor;
attribute vec4  gl_MultiTexCoord0;
attribute vec4  gl_MultiTexCoord1;
attribute vec4  gl_MultiTexCoord2;
attribute vec4  gl_MultiTexCoord3;
attribute vec4  gl_MultiTexCoord4;
attribute vec4  gl_MultiTexCoord5;
attribute vec4  gl_MultiTexCoord6;
attribute vec4  gl_MultiTexCoord7;
attribute float gl_FogCoord;
```

Varying Outputs (7.6 p48) *access=RW*
```
varying vec4  gl_FrontColor;
varying vec4  gl_BackColor;  enable GL_VERTEX_PROGRAM_TWO_SIDE
varying vec4  gl_FrontSecondaryColor;
varying vec4  gl_BackSecondaryColor;
varying vec4  gl_TexCoord[ ];     MAX=gl_MaxTextureCoords
varying float gl_FogFragCoord;
```

FRAGMENT SHADER VARIABLES

Special Output Variables (7.2 p43) *access=RW*
```
vec4  gl_FragColor;
vec4  gl_FragData[gl_MaxDrawBuffers];
float gl_FragDepth;     DEFAULT=glFragCoord.z
```

Varying Inputs (7.6 p48) *access=RO*
```
varying vec4  gl_Color;
varying vec4  gl_SecondaryColor;
varying vec4  gl_TexCoord[ ];     MAX=gl_MaxTextureCoords
varying float gl_FogFragCoord;
```

Special Input Variables (7.2 p43) *access=RO*
```
vec4  gl_FragCoord;          pixel coordinates
bool  gl_FrontFacing;
```

BUILT-IN CONSTANTS (7.4 p44)
```
const int gl_MaxVertexUniformComponents;
const int gl_MaxFragmentUniformComponents;
const int gl_MaxVertexAttribs;
const int gl_MaxVaryingFloats;
const int gl_MaxDrawBuffers;
const int gl_MaxTextureCoords;
const int gl_MaxTextureUnits;
const int gl_MaxTextureImageUnits;
const int gl_MaxVertexTextureImageUnits;
const int gl_MaxCombinedTextureImageUnits;
const int gl_MaxLights;
const int gl_MaxClipPlanes;
```

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];

uniform mat4 gl_ModelViewMatrixInverseTranspose;
uniform mat4 gl_ModelViewProjectionMatrixInverseTranspose;
uniform mat4 gl_ProjectionMatrixInverseTranspose;
uniform mat4 gl_TextureMatrixInverseTranspose[gl_MaxTextureCoords];

uniform mat3 gl_NormalMatrix;
uniform float gl_NormalScale;

struct gl_DepthRangeParameters {
   float near;
   float far;
   float diff;
};
uniform gl_DepthRangeParameters gl_DepthRange;

struct gl_FogParameters {
   vec4 color;
   float density;
   float start;
   float end;
   float scale;
};
uniform gl_FogParameters gl_Fog;

struct gl_LightSourceParameters {
   vec4 ambient;
   vec4 diffuse;
   vec4 specular;
   vec4 position;
   vec4 halfVector;
   vec3 spotDirection;
   float spotExponent;
   float spotCutoff;
   float spotCosCutoff;
   float constantAttenuation;
   float linearAttenuation;
   float quadraticAttenuation;
};
uniform gl_LightSourceParameters gl_LightSource[gl_MaxLights];

struct gl_LightModelParameters {
   vec4 ambient;
};
uniform gl_LightModelParameters gl_LightModel;

struct gl_LightModelProducts {
   vec4 sceneColor;
};
uniform gl_LightModelProducts gl_FrontLightModelProduct;
uniform gl_LightModelProducts gl_BackLightModelProduct;

struct gl_LightProducts {
   vec4 ambient;
   vec4 diffuse;
   vec4 specular;
};
uniform gl_LightProducts gl_FrontLightProduct[gl_MaxLights];
uniform gl_LightProducts gl_BackLightProduct[gl_MaxLights];

struct gl_MaterialParameters {
   vec4 emission;
   vec4 ambient;
   vec4 diffuse;
   vec4 specular;
   float shininess;
};
uniform gl_MaterialParameters gl_FrontMaterial;
uniform gl_MaterialParameters gl_BackMaterial;

struct gl_PointParameters {
   float size;
   float sizeMin;
   float sizeMax;
   float fadeThresholdSize;
   float distanceConstantAttenuation;
   float distanceLinearAttenuation;
   float distanceQuadraticAttenuation;
};
uniform gl_PointParameters gl_Point;

uniform vec4 gl_TextureEnvColor[gl_MaxTextureUnits];  (1)

uniform vec4 gl_ClipPlane[gl_MaxClipPlanes];

uniform vec4 gl_EyePlaneS[gl_MaxTextureCoords];
uniform vec4 gl_EyePlaneT[gl_MaxTextureCoords];
uniform vec4 gl_EyePlaneR[gl_MaxTextureCoords];
uniform vec4 gl_EyePlaneQ[gl_MaxTextureCoords];

uniform vec4 gl_ObjectPlaneS[gl_MaxTextureCoords];
uniform vec4 gl_ObjectPlaneT[gl_MaxTextureCoords];
uniform vec4 gl_ObjectPlaneR[gl_MaxTextureCoords];
uniform vec4 gl_ObjectPlaneQ[gl_MaxTextureCoords];
```

OpenSceneGraph Preset Uniforms
as of OSG 1.0
```
int    osg_FrameNumber;
float  osg_FrameTime;
float  osg_DeltaFrameTime;
mat4   osg_ViewMatrix;
mat4   osg_ViewMatrixInverse;
```


Notes
1. Corrects a typo in the OpenGL 2.0 specification.

http://www.opengl.org/sdk/libs/OpenSceneGraph/glsl_quickref.pdf

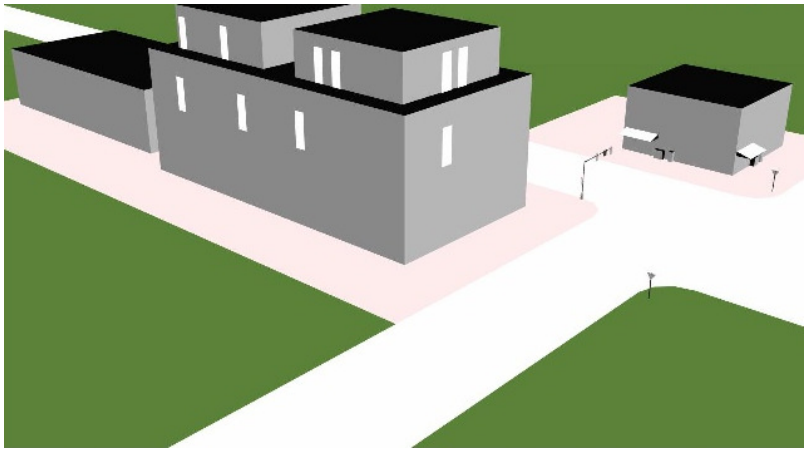
Tutorials and Documentation

- OpenGL and GLSL specifications
  http://www.opengl.org/documentation/specs/
- GLSL tutorials
  http://www.lighthouse3d.com/opengl/glsl/
  http://www.clockworkcoders.com/oglsl/tutorials.html
- OpenGL Programming Guide (Red Book)
- OpenGL Shading Language (Orange Book)

Lecture Overview

- Shader programming
  - Vertex shaders
  - Fragment shaders
  - GLSL
- Texturing
  - Overview

Large Triangles



Pros:

- Often sufficient for simple geometry
- Fast to render

Cons:

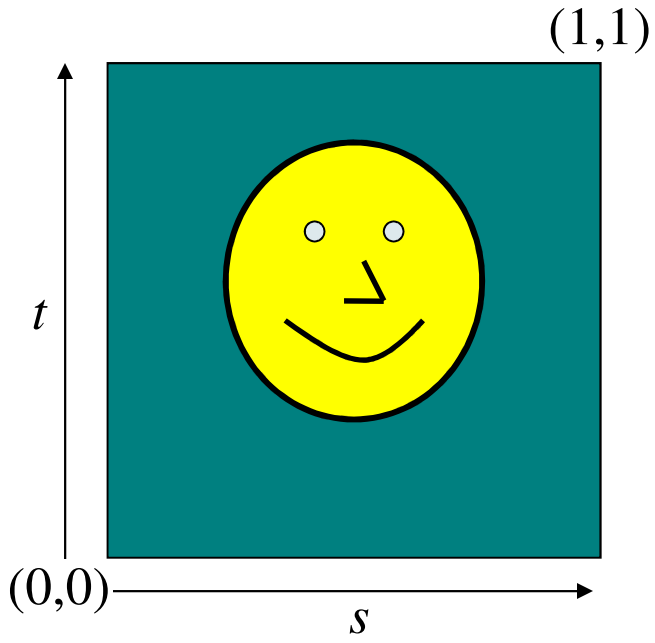
- Per vertex colors look bad
- Need more interesting surfaces
  - Detailed color variation, small scale bumps, roughness

Texture Mapping

- Attach textures (images) onto surfaces
- Same triangle count, much more realistic appearance

Texture Mapping

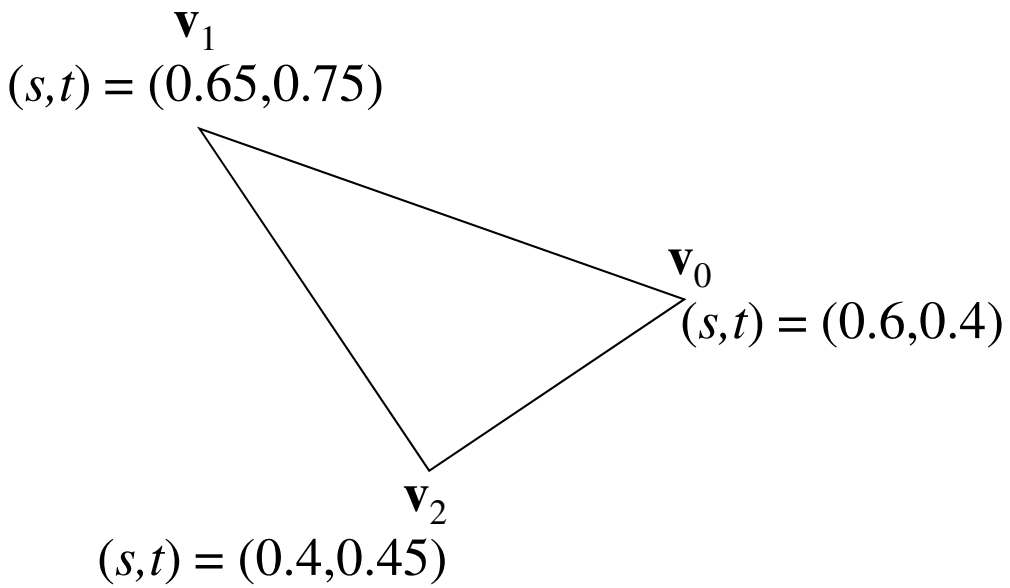
- Goal: assign locations in texture to locations on 3D geometry
- Introduce texture space
  - Texture pixels (texels) have texture coordinates (s,t)
- Convention
  - Bottom left corner of texture is at $(s,t) = (0,0)$
  - Top right corner is at $(s,t) = (1,1)$



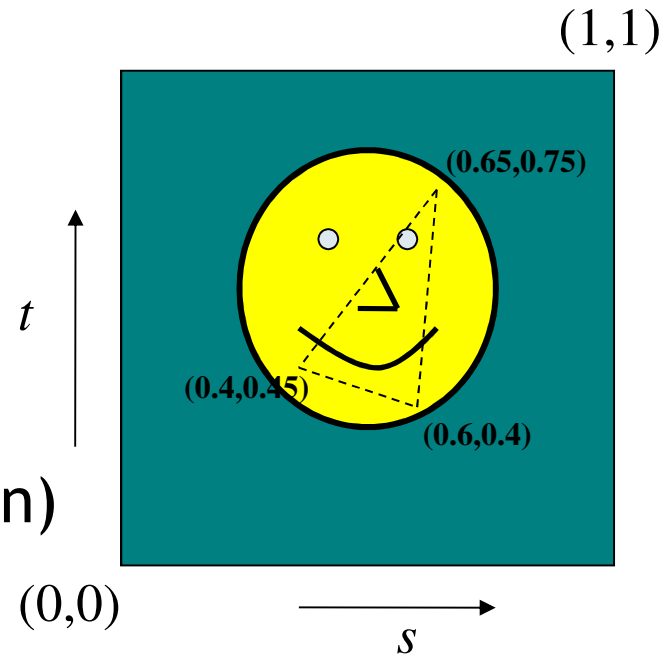
Texture space

Texture Mapping

- Store texture coordinates at each triangle vertex



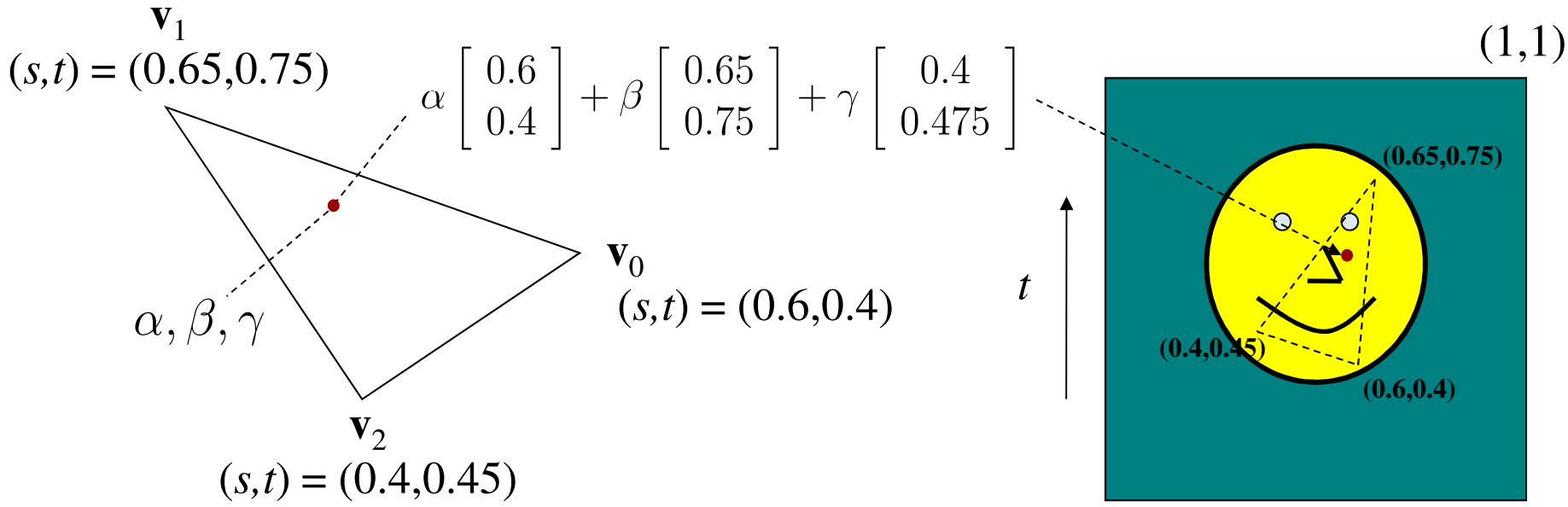
Triangle (in any space before projection)



Texture space

Texture Mapping

- Each point on triangle has barycentric coordinates α, β, γ
- Use barycentric coordinates to interpolate texture coordinates

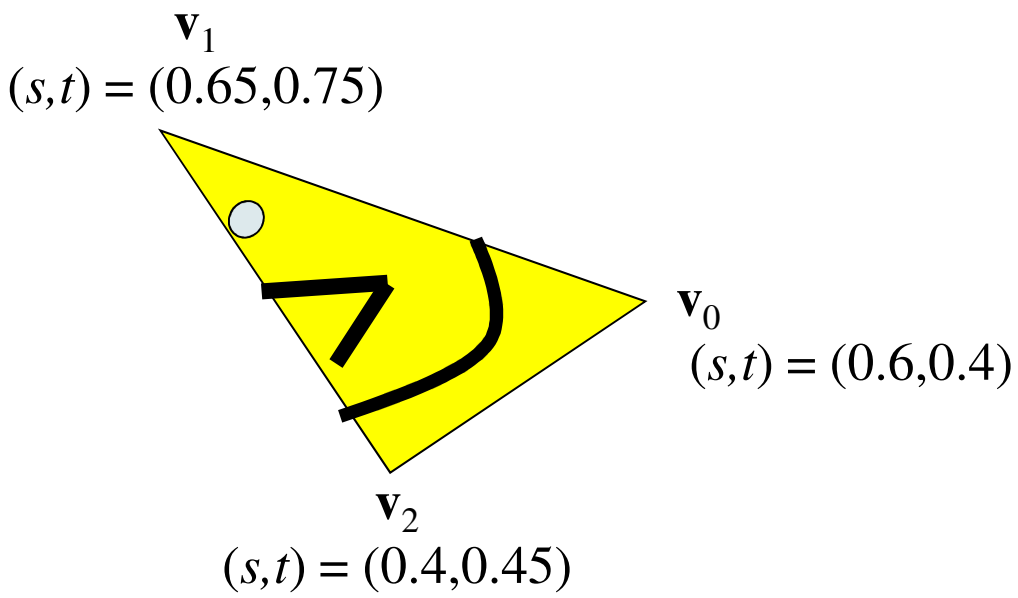


Triangle (in any space before projection)

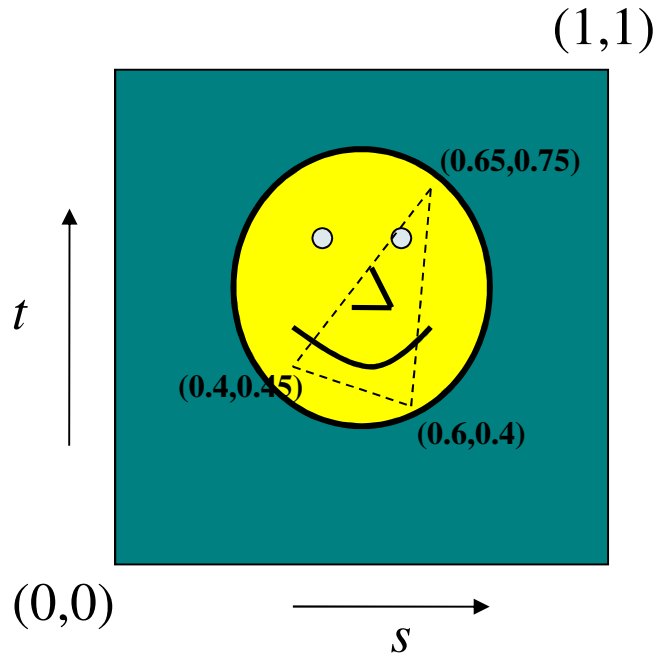
Texture space

Texture Mapping

- Each point on triangle has corresponding point in texture



Triangle (in any space before projection)

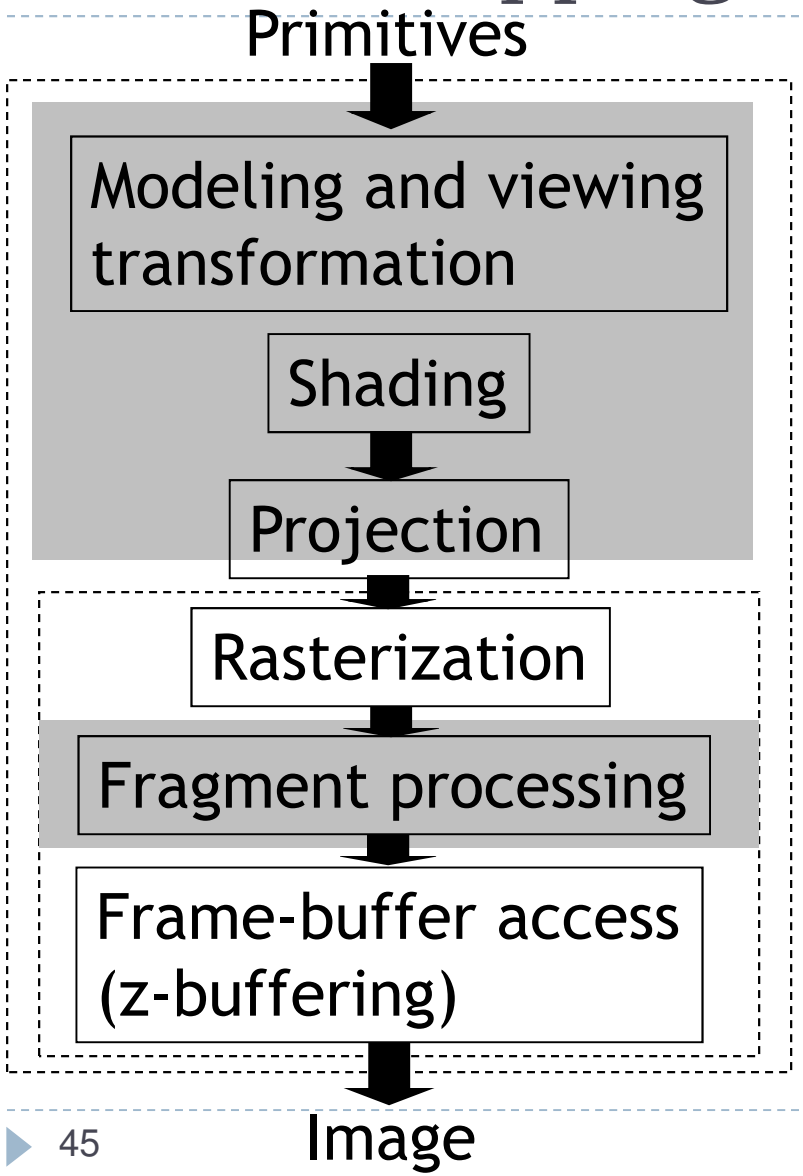


Texture space

Rendering

- Given
  - Texture coordinates at each vertex
  - Texture image
- At each pixel, use barycentric coordinates to interpolate texture coordinates
- Look up corresponding texel
- Paint current pixel with texel color
- All computations are done on the GPU

Texture Mapping

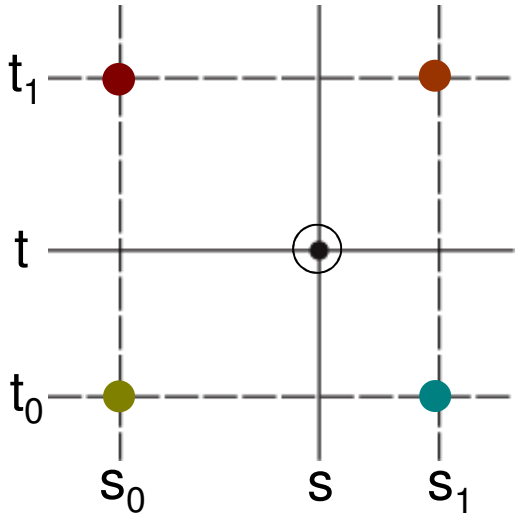


Includes texture mapping

Texture Look-Up

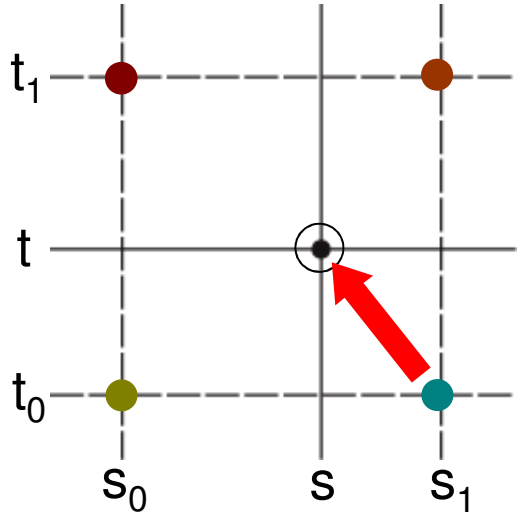
- Given interpolated texture coordinates (s, t) at current pixel
- Closest four texels in texture space are at

  (s_0,t_0), (s_1,t_0), (s_0,t_1), (s_1,t_1)
- How to compute pixel color?



Nearest-Neighbor Interpolation

- Use color of closest texel



- Simple, but low quality and aliasing

Bilinear Interpolation

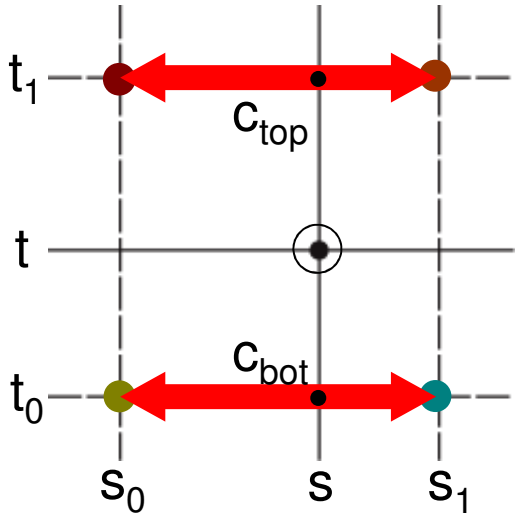
1. Linear interpolation horizontally:

   Ratio in s direction r_s:

$$r_s = \frac{s - s_0}{s_1 - s_0}$$

$c_{top} = tex(s_0,t_1)\ (1-r_s) + tex(s_1,t_1)\ r_s$

$c_{bot} = tex(s_0, t_0)\ (1-r_s) + tex(s_1,t_0)\ r_s$



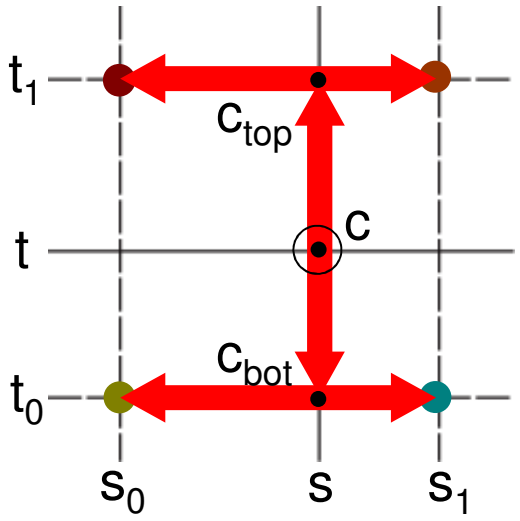
Bilinear Interpolation

2. Linear interpolation vertically

Ratio in t direction r_t:

$$r_t = \frac{t - t_0}{t_1 - t_0}$$

$c = c_{bot} (1 - r_t) + c_{top} r_t$



Next Lecture

- More on Texture Mapping