

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

Jürgen P. Schulze, Ph.D.  
University of California, San Diego  
Fall Quarter 2013

# Announcements

---

- ▶ Limited office hours this Thursday: only 3:30-5:30pm

## **Midterm Exam #1**

- ▶ This Thursday, Oct 24<sup>th</sup>
- ▶ Location: Center Hall 119
- ▶ Time: 2:00pm – 3:20pm
- ▶ To bring:
  - ▶ Pen/pencil, eraser
  - ▶ Ruler
  - ▶ Scratch paper
  - ▶ Photo ID: put on your table until you got checked off
- ▶ Not allowed:
  - ▶ Books, written or printed notes, cell phones, other electronic devices

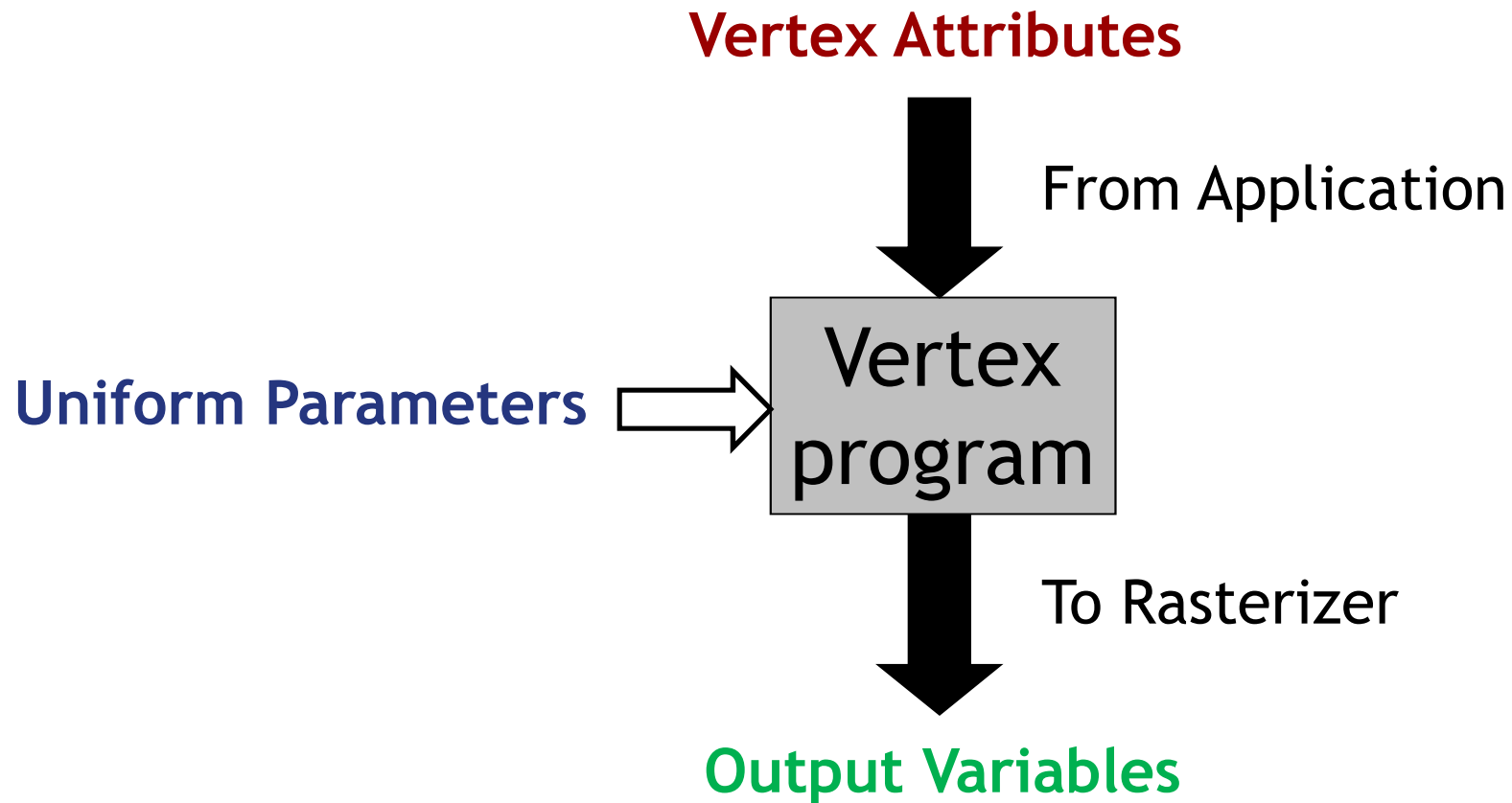
# Lecture Overview

---

- ▶ Programmable Shaders
  - ▶ Vertex Programs
  - ▶ Fragment Programs
  - ▶ GLSL

# Vertex Programs

---



# Vertex Attributes

---

- ▶ Declared using the `attribute` storage classifier
- ▶ Different for each execution of the vertex program
- ▶ Can be modified by the vertex program
- ▶ Two types:
  - ▶ Pre-defined OpenGL attributes. Examples:  

```
attribute vec4 gl_Vertex;  
attribute vec3 gl_Normal;  
attribute vec4 gl_Color;
```
  - ▶ User-defined attributes. Example:  

```
attribute float myAttrib;
```

# Uniform Parameters

---

- ▶ Declared by `uniform` storage classifier
- ▶ Normally the same for all vertices
- ▶ Read-only
- ▶ Two types:
  - ▶ Pre-defined OpenGL state variables
  - ▶ User-defined parameters

# Uniform Parameters: Pre-Defined

---

- ▶ Provide access to the OpenGL state

- ▶ Examples for pre-defined variables:

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

# Uniform Parameters: User-Defined

---

- ▶ Parameters that are set by the application
- ▶ Should not be changed frequently
  - ▶ Especially not on a per-vertex basis!
- ▶ **To access, use** `glGetUniformLocation`, `glUniform*` in application
- ▶ **Example:**
  - ▶ In shader declare  
`uniform float a;`
  - ▶ **Set value of a in application:**  
`GLuint p;`  
`int I = glGetUniformLocation(p, "a");`  
`glUniform1f(i, 1.0f);`



# Vertex Programs: Output Variables

---

- ▶ Required output: homogeneous vertex coordinates

```
vec4 gl_Position
```

- ▶ **varying** output variables

- ▶ Mechanism to send data to the fragment shader

- ▶ Will be interpolated during rasterization

- ▶ Fragment shader gets interpolated data

- ▶ Pre-defined `varying` output variables, for example:

```
varying vec4 gl_FrontColor;
```

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

Any pre-defined output variable that you do not overwrite will have the value of the OpenGL state.

- ▶ User-defined `varying` output variables, e.g.:

```
varying vec4 vertex_color;
```

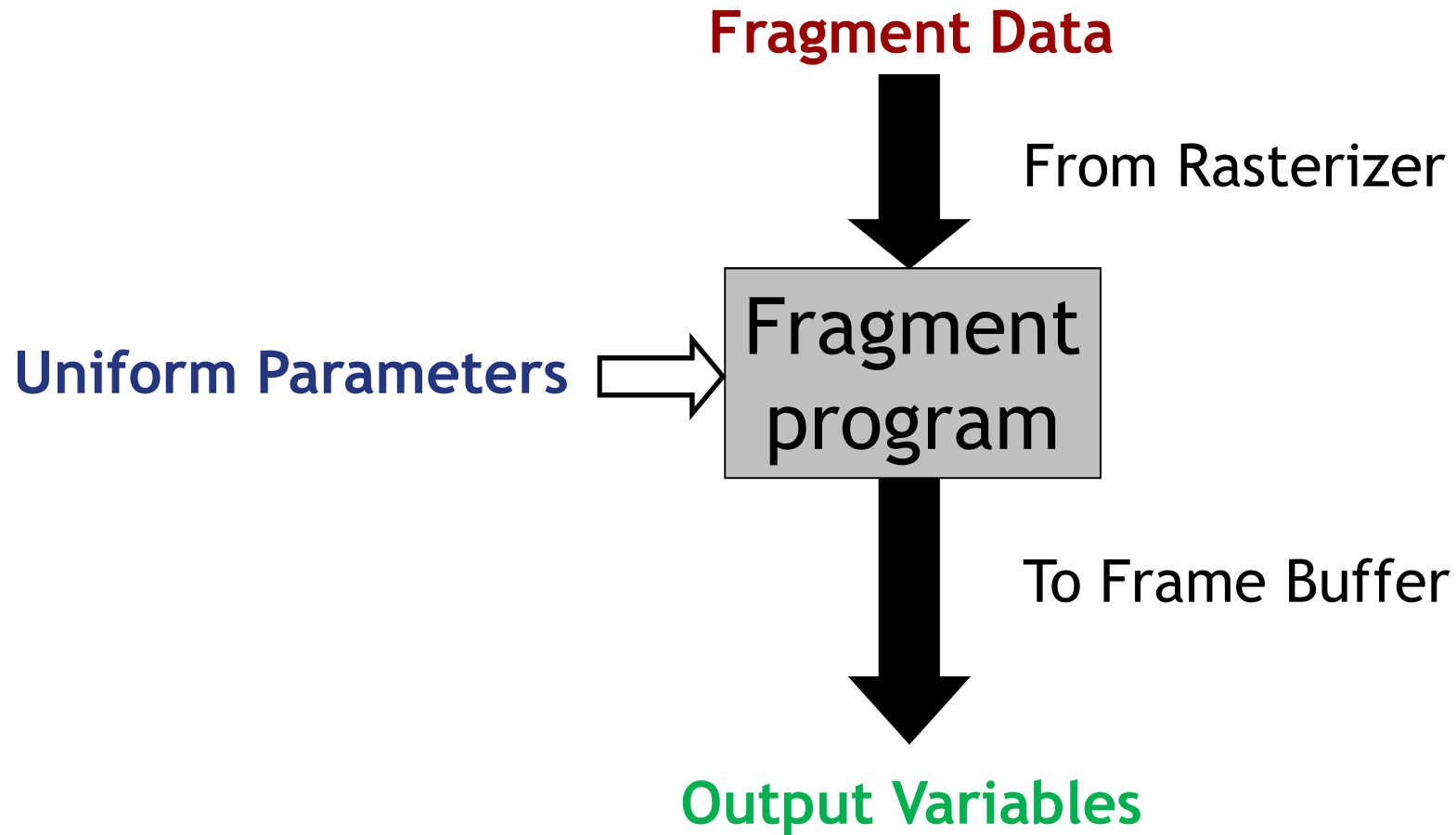
# Lecture Overview

---

- ▶ Programmable Shaders
  - ▶ Vertex Programs
  - ▶ **Fragment Programs**
  - ▶ GLSL

# Fragment Programs

---



# Fragment Data

---

- ▶ Changes for each execution of the fragment program
- ▶ Fragment data includes:
  - ▶ Interpolated standard OpenGL variables for fragment shader, as generated by vertex shader, for example:

```
varying vec4 gl_Color;  
varying vec4 gl_TexCoord[];
```
  - ▶ **Interpolated varying variables from vertex shader**
    - ▶ Allows data to be passed from vertex to fragment shader

# Uniform Parameters

---

- ▶ Same as in vertex programs

# Output Variables

---

- ▶ **Pre-defined output variables:**
  - ▶ `gl_FragColor`
  - ▶ `gl_FragDepth`
- ▶ **OpenGL writes these to the frame buffer**
- ▶ **Result is undefined if you do not set these variables!**

# Lecture Overview

---

- ▶ **Programmable Shaders**
  - ▶ Vertex Programs
  - ▶ Fragment Programs
  - ▶ **GLSL**

# GLSL Main Features

---

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



# Example: Treat normals as colors

---

```
// Vertex Shader
varying vec4 color;

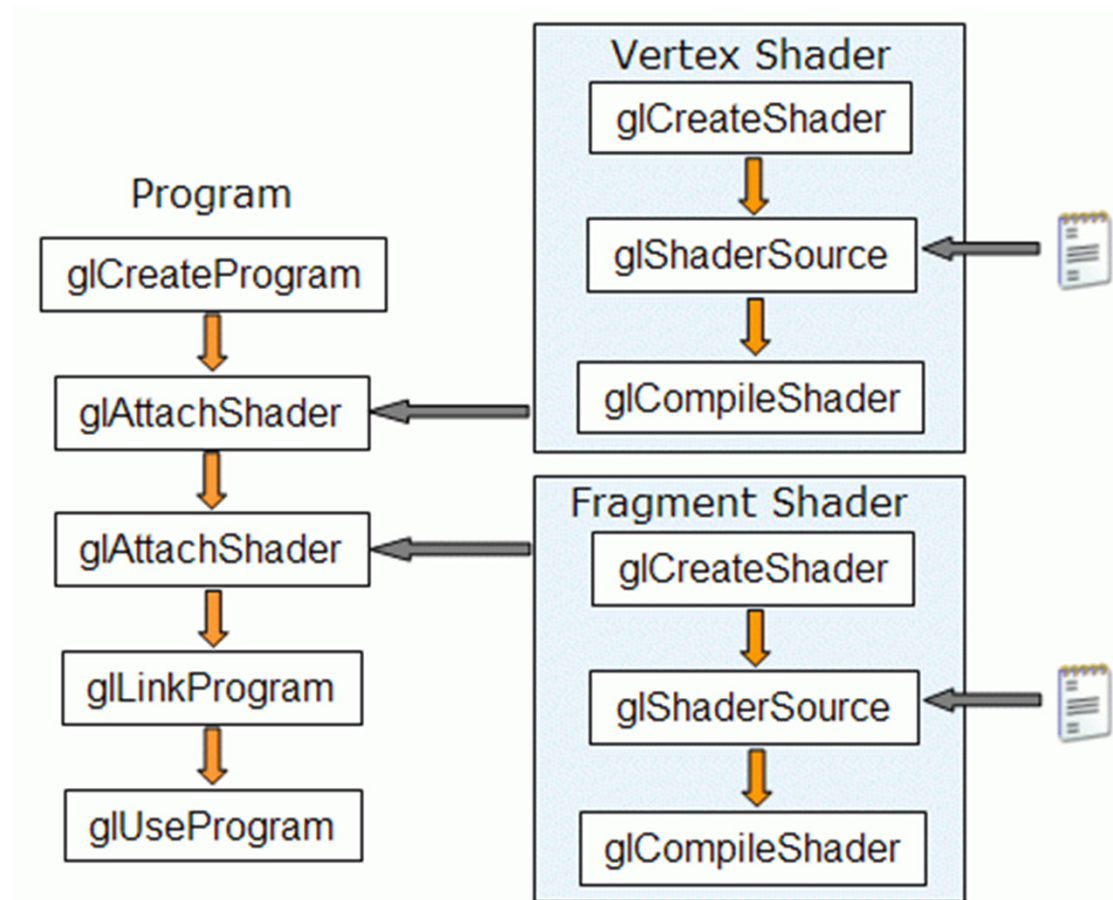
void main()
{
    // Treat the normal (x, y, z) values as (r, g, b) color
    components.
    color = vec4(clamp(abs((gl_Normal + 1.0) * 0.5), 0.0, 1.0),
1.0);

    gl_Position = ftransform();
}

// Fragment Shader
varying vec4 color;

void main()
{
    gl_FragColor = color;
}
```

# Creating Shaders in OpenGL



Source: Gabriel Zachmann, Clausthal University

# Video

---

- ▶ OpenGL and GLSL Demo 2

- ▶ <http://www.youtube.com/watch?v=cQ8PI6X0Op8>



# 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/opengl/tutorials.html>
- ▶ OpenGL Programming Guide (Red Book)
- ▶ OpenGL Shading Language (Orange Book)
- ▶ OpenGL 4.4 API Reference Card
  - ▶ <http://www.khronos.org/files/opengl44-quick-reference-card.pdf>

# Lecture Overview

---

- ▶ Texture Mapping
  - ▶ Overview
  - ▶ Wrapping
  - ▶ Texture coordinates
  - ▶ Anti-aliasing

# Large Triangles

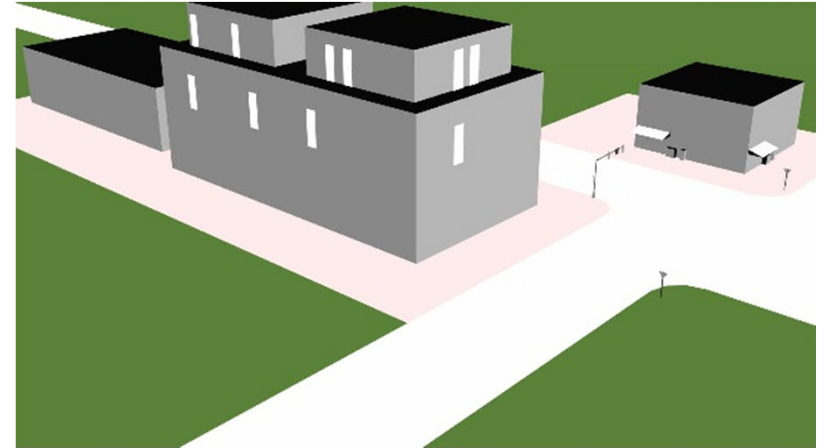
---

## Pros:

- ▶ Often sufficient for simple geometry
- ▶ Fast to render

## Cons:

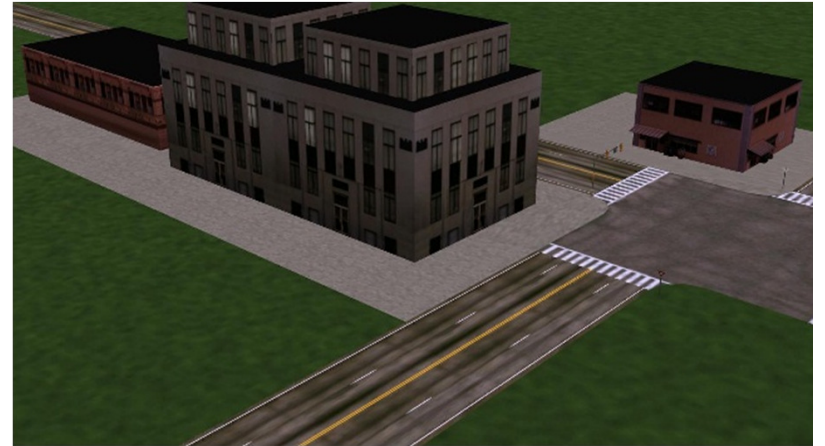
- ▶ Per vertex colors look boring and computer-generated



# Texture Mapping

---

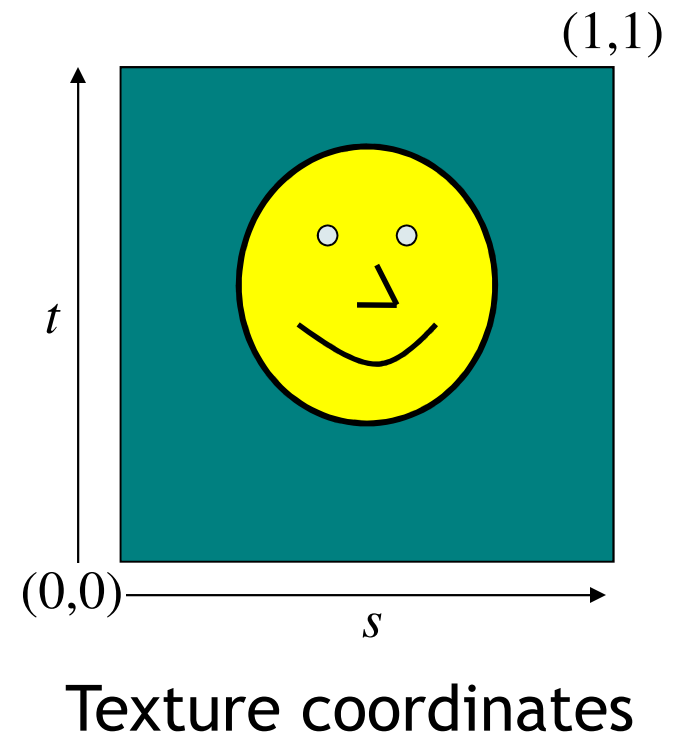
- ▶ Map textures (images) onto surface polygons
- ▶ Same triangle count, much more realistic appearance



# Texture Mapping

---

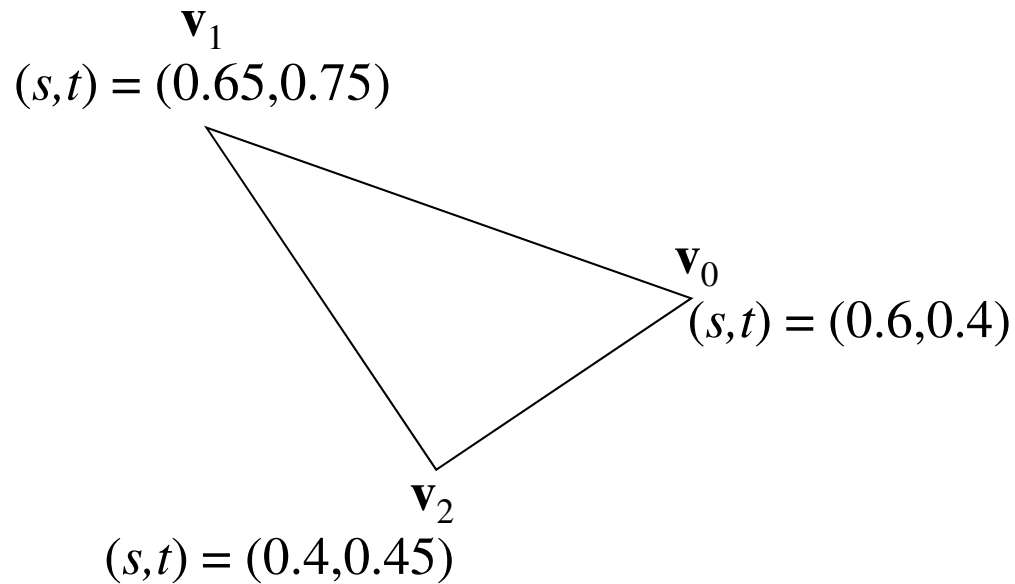
- ▶ Goal: map locations in texture to locations on 3D geometry
- ▶ Texture coordinate space
  - ▶ Texture pixels (texels) have texture coordinates  $(s, t)$
- ▶ Convention
  - ▶ Bottom left corner of texture is at  $(s, t) = (0, 0)$
  - ▶ Top right corner is at  $(s, t) = (1, 1)$



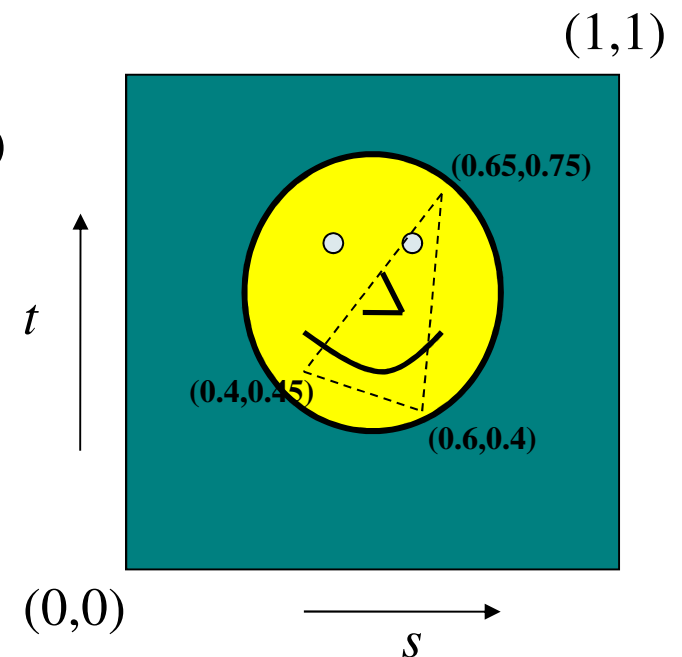


# Texture Mapping

- Store 2D texture coordinates  $s, t$  with each triangle vertex



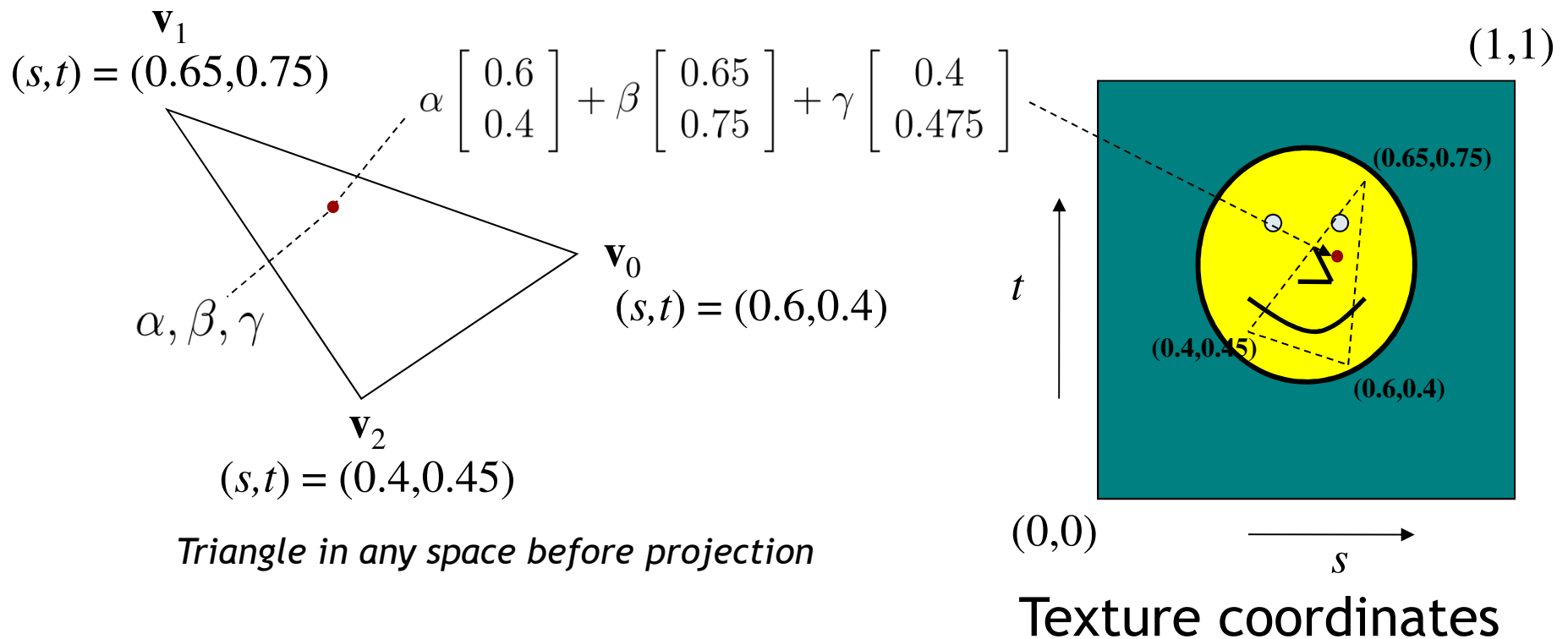
*Triangle in any space before projection*



Texture coordinates

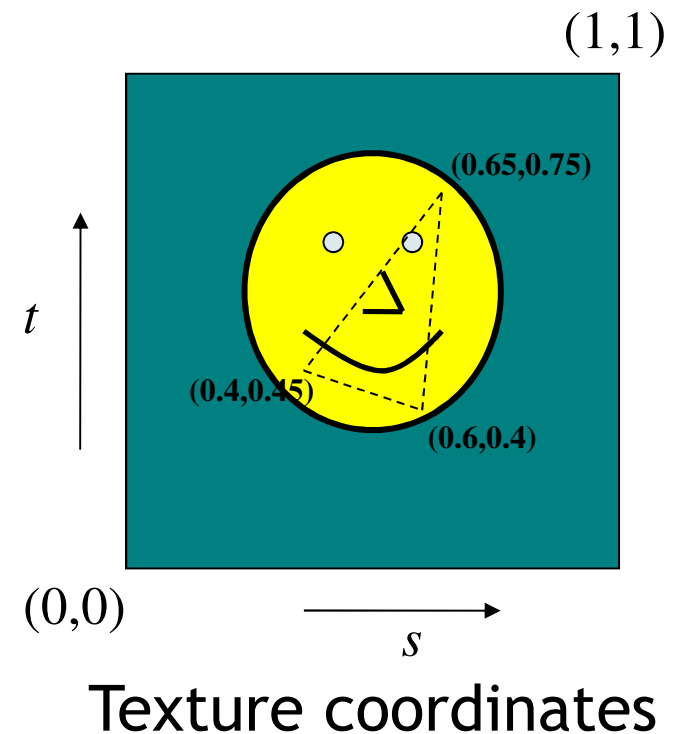
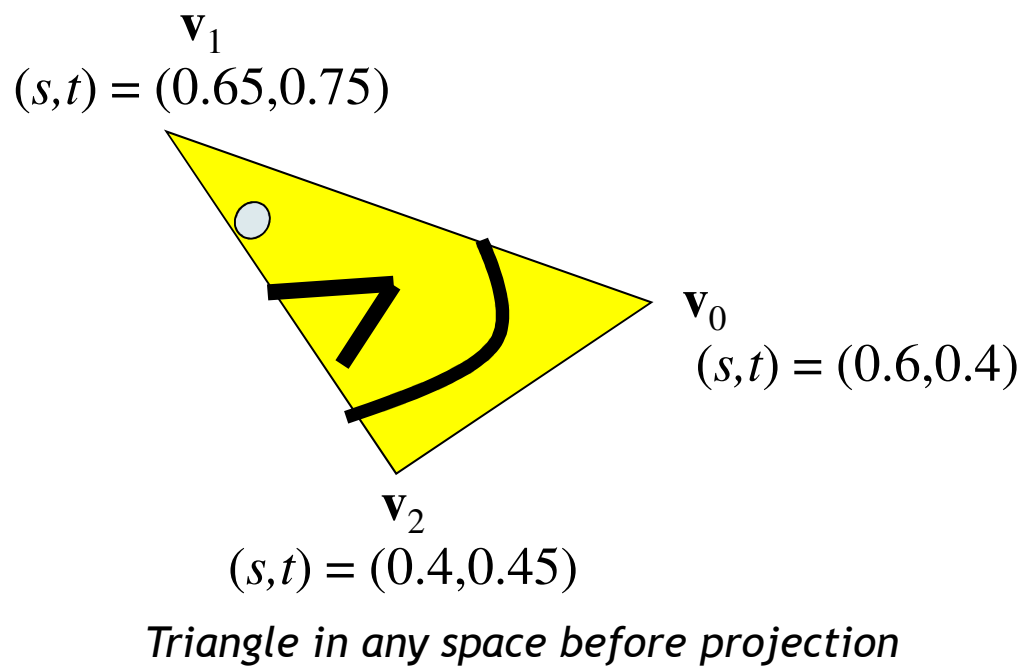
# Texture Mapping

- ▶ Each point on triangle has barycentric coordinates  $\alpha, \beta, \gamma$
- ▶ Barycentric coordinates interpolate texture coordinates
- ▶ Done automatically on GPU



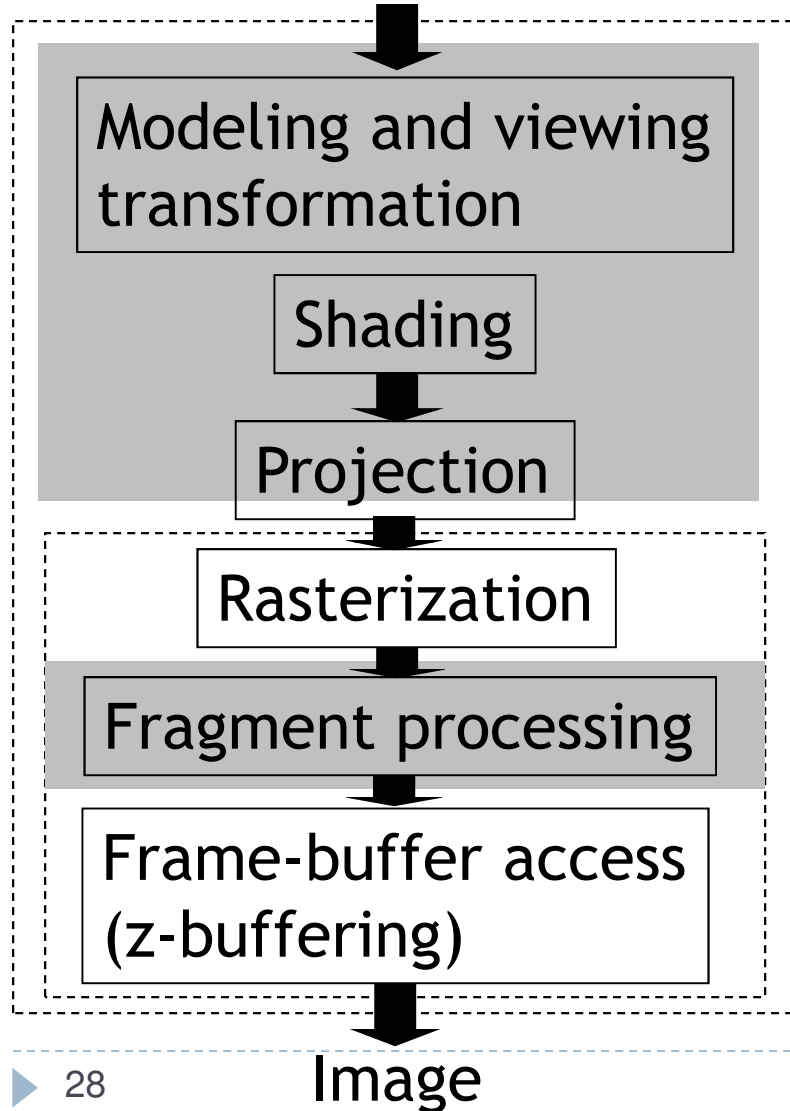
# Texture Mapping

- ▶ Each point on triangle gets color from its corresponding point in texture



# Texture Mapping

Primitives

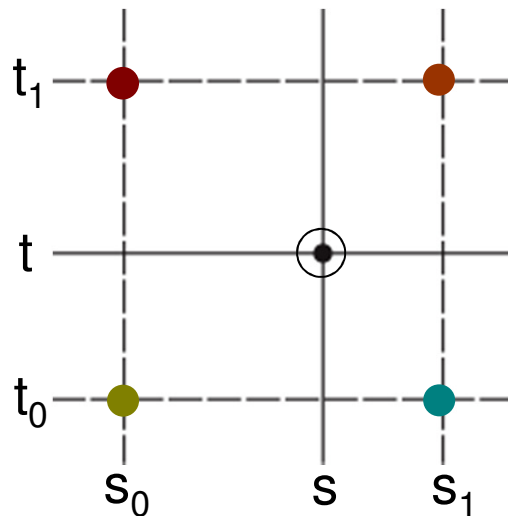


 Includes texture mapping

# Texture Look-Up

---

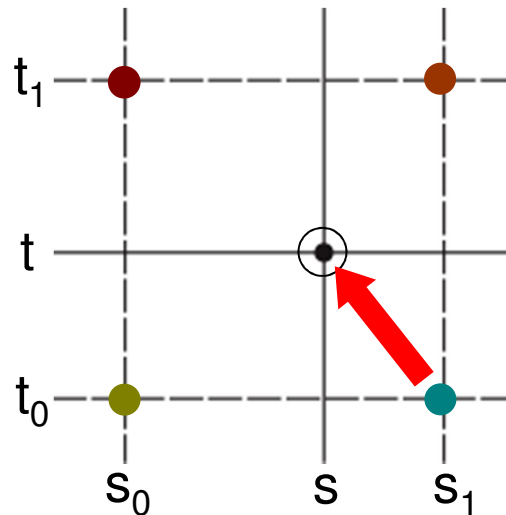
- ▶ Given interpolated texture coordinates  $(s, t)$  at current pixel
- ▶ Closest four texels in texture space are at  $(s_0, t_0)$ ,  $(s_1, t_0)$ ,  $(s_0, t_1)$ ,  $(s_1, t_1)$
- ▶ How to compute pixel color?



# Nearest-Neighbor Interpolation

---

- ▶ Use color of closest texel



- ▶ Simple, but low quality and aliasing

# Bilinear Interpolation

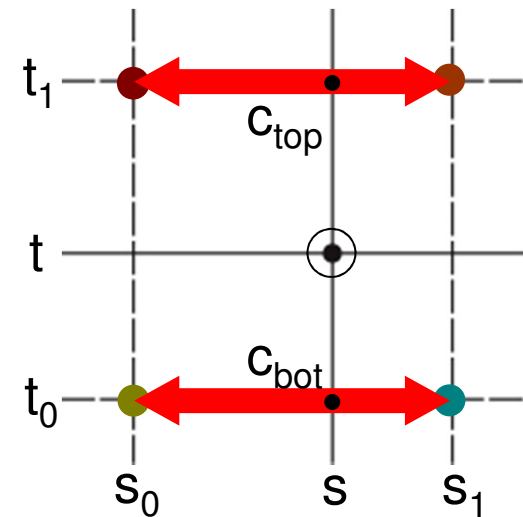
## I. Linear interpolation horizontally:

Ratio in s direction  $r_s$ :

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

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

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



# Bilinear Interpolation

## 2. Linear interpolation vertically

Ratio in  $t$  direction  $r_t$ :

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

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

