CSE 167: Introduction to Computer Graphics Lecture #12: GLSL

Jürgen P. Schulze, Ph.D.
University of California, San Diego
Fall Quarter 2014

Announcements

- Project 5 due Friday at 3:30pm
- 2 REU positions for high speed networking
  - Under Dr. Thomas DeFanti

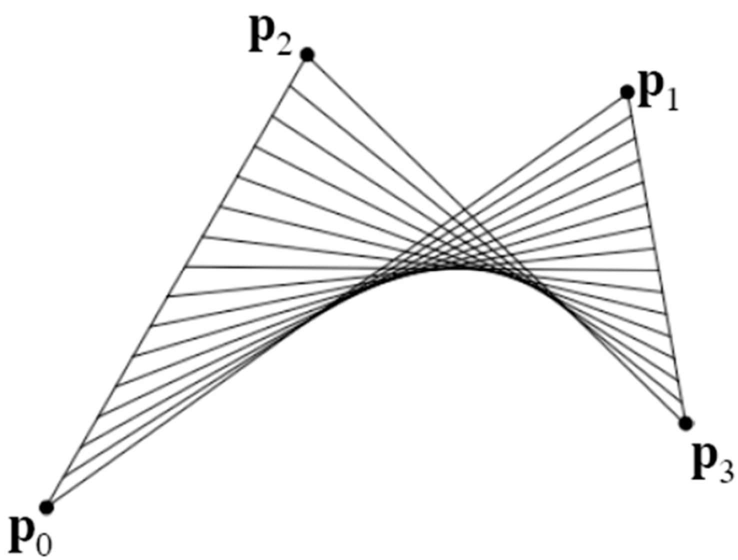
UCSD

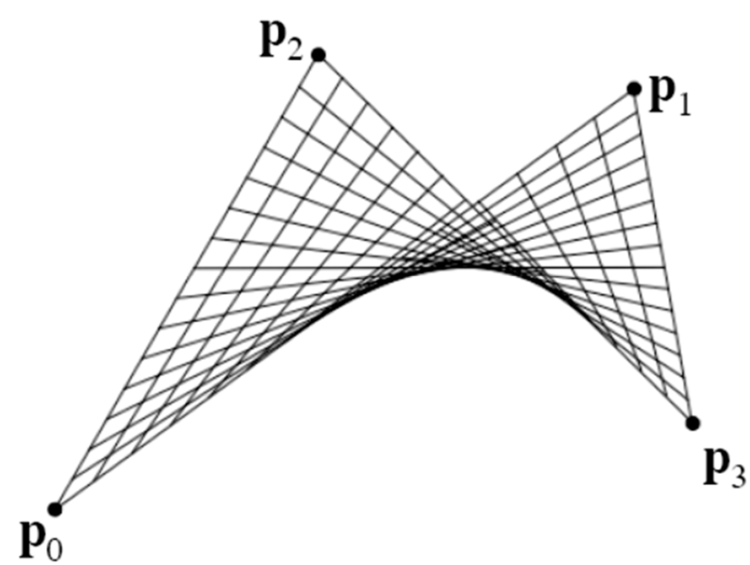
Overview

- Bi-linear patch
- Bi-cubic Bézier patch
- Advanced parametric surfaces

Bilinear Patch

- Visualization





Bilinear Patches

- Weighted sum of control points

$$\mathbf{x}(u,v) = (1-u)(1-v)\mathbf{p}_0 + u(1-v)\mathbf{p}_1 + (1-u)v\mathbf{p}_2 + uv\mathbf{p}_3$$

- Bilinear polynomial

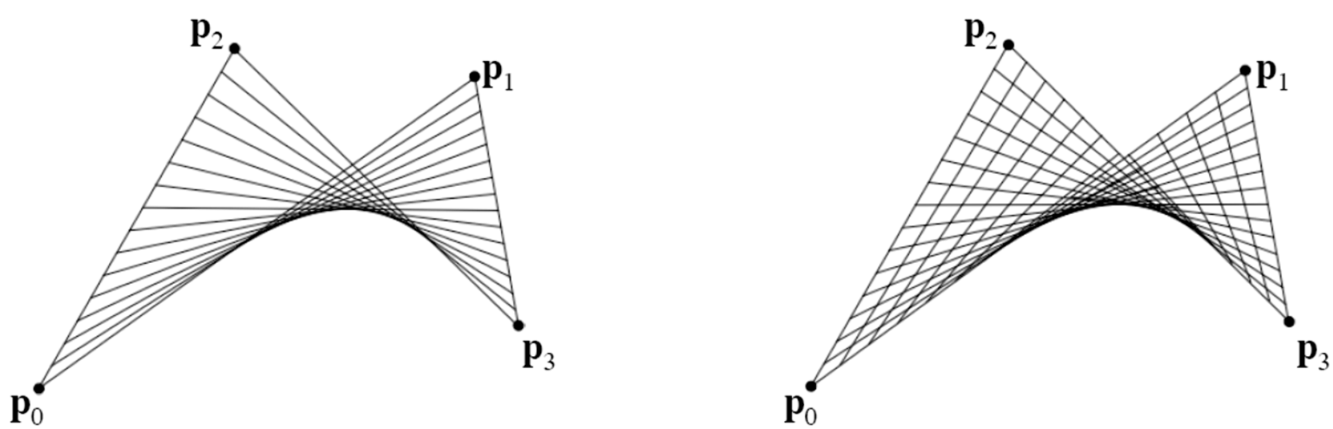
$$\mathbf{x}(u,v) = (\mathbf{p}_0 - \mathbf{p}_1 - \mathbf{p}_2 + \mathbf{p}_3)uv + (\mathbf{p}_1 - \mathbf{p}_0)u + (\mathbf{p}_2 - \mathbf{p}_0)v + \mathbf{p}_0$$

- Matrix form

$$x(u,v) = \begin{bmatrix} 1-u & u \end{bmatrix} \begin{bmatrix} p_0 & p_2 \\ p_1 & p_3 \end{bmatrix} \begin{bmatrix} 1-v \\ v \end{bmatrix}$$

Properties

- Interpolates the control points
- The boundaries are straight line segments
- If all 4 points of the control mesh are co-planar, the patch is flat
- If the points are not co-planar, we get a curved surface
  - saddle shape (hyperbolic paraboloid)
- *The parametric curves are all straight line segments!*
  - a (doubly) *ruled surface:* has (two) straight lines through every point



- Not terribly useful as a modeling primitive

Overview

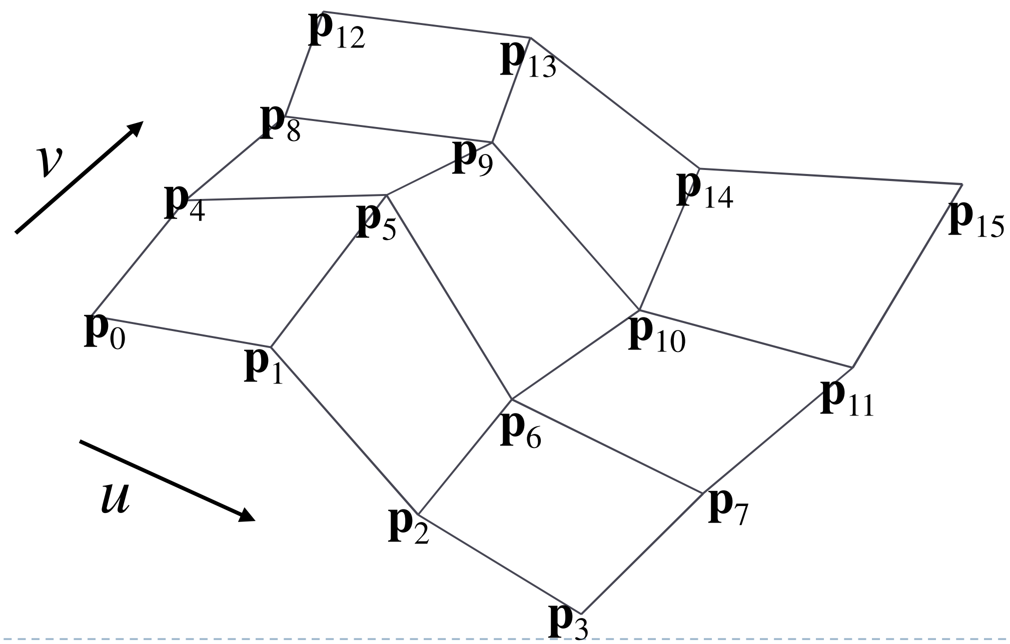
- Bi-linear patch
- Bi-cubic Bézier patch
- Advanced parametric surfaces

Bicubic Bézier patch

- Grid of 4x4 control points, $\mathbf{p}_0$ through $\mathbf{p}_{15}$
- Four rows of control points define Bézier curves along u

  $\mathbf{p}_0,\mathbf{p}_1,\mathbf{p}_2,\mathbf{p}_3$; $\mathbf{p}_4,\mathbf{p}_5,\mathbf{p}_6,\mathbf{p}_7$; $\mathbf{p}_8,\mathbf{p}_9,\mathbf{p}_{10},\mathbf{p}_{11}$; $\mathbf{p}_{12},\mathbf{p}_{13},\mathbf{p}_{14},\mathbf{p}_{15}$
- Four columns define Bézier curves along v

  $\mathbf{p}_0,\mathbf{p}_4,\mathbf{p}_8,\mathbf{p}_{12}$; $\mathbf{p}_1,\mathbf{p}_6,\mathbf{p}_9,\mathbf{p}_{13}$; $\mathbf{p}_2,\mathbf{p}_6,\mathbf{p}_{10},\mathbf{p}_{14}$; $\mathbf{p}_3,\mathbf{p}_7,\mathbf{p}_{11},\mathbf{p}_{15}$

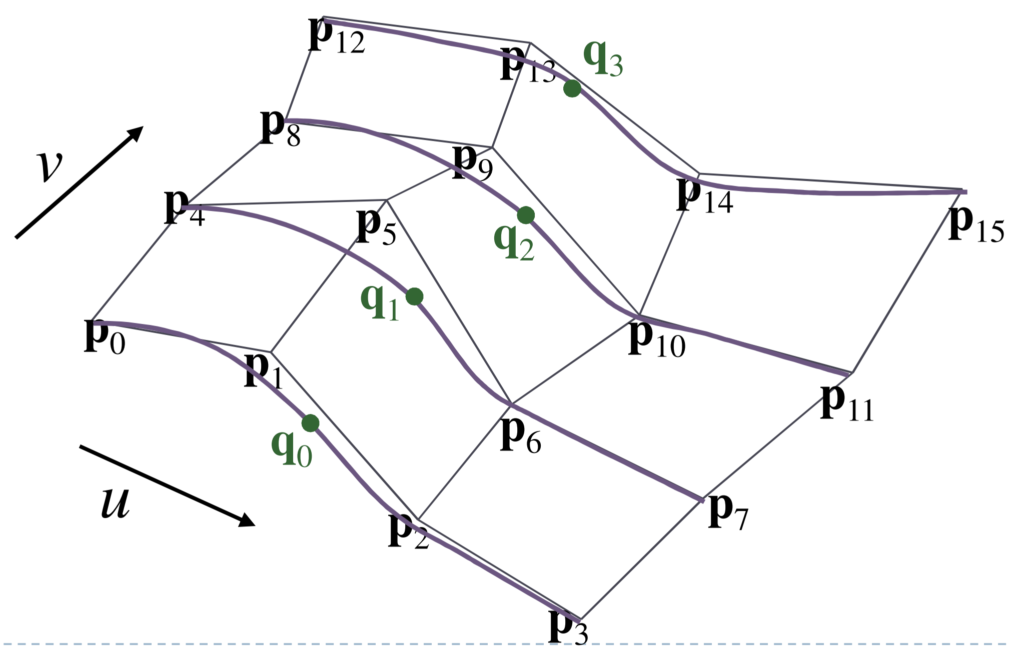


UCSD

Bézier Patch (Step 1)

- Evaluate four *u*-direction Bézier curves at scalar value *u [0..1]*
- Get points $\mathbf{q}_0 \ldots \mathbf{q}_3$

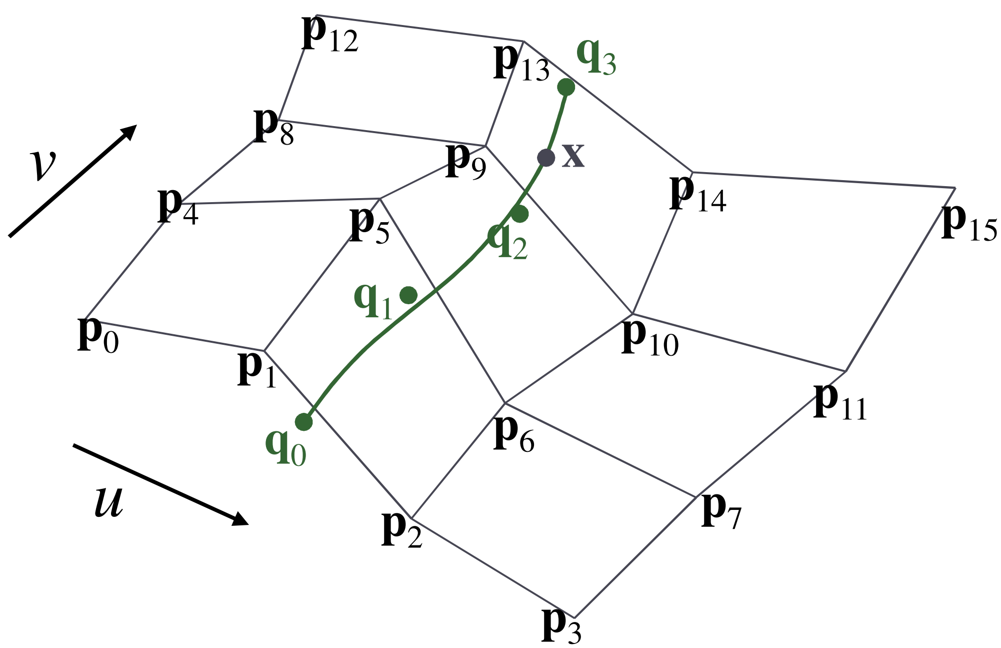
$$\mathbf{q}_0 = Bez(u, \mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3)$$
$$\mathbf{q}_1 = Bez(u, \mathbf{p}_4, \mathbf{p}_5, \mathbf{p}_6, \mathbf{p}_7)$$
$$\mathbf{q}_2 = Bez(u, \mathbf{p}_8, \mathbf{p}_9, \mathbf{p}_{10}, \mathbf{p}_{11})$$
$$\mathbf{q}_3 = Bez(u, \mathbf{p}_{12}, \mathbf{p}_{13}, \mathbf{p}_{14}, \mathbf{p}_{15})$$



Bézier Patch (Step 2)

- Points $\mathbf{q}_{0} \ldots \mathbf{q}_{3}$ define a Bézier curve
- Evaluate it at v *[0..1]*

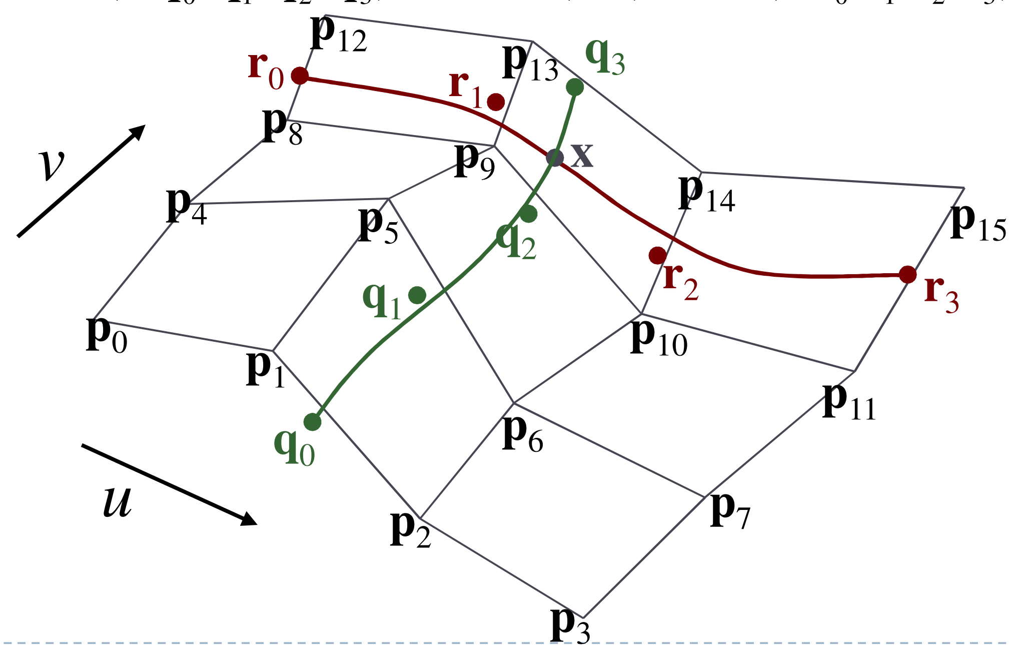
$$\mathbf{x}(u,v) = Bez(v, \mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2, \mathbf{q}_3)$$



Bézier Patch

- Same result in either order (evaluate u before v or vice versa)

$$
\begin{aligned}
\mathbf{q_0} &= Bez(u,\mathbf{p}_0,\mathbf{p}_1,\mathbf{p}_2,\mathbf{p}_3) & \mathbf{r_0} &= Bez(v,\mathbf{p}_0,\mathbf{p}_4,\mathbf{p}_8,\mathbf{p}_{12})\\
\mathbf{q}_1 &= Bez(u,\mathbf{p}_4,\mathbf{p}_5,\mathbf{p}_6,\mathbf{p}_7) & \mathbf{r}_1 &= Bez(v,\mathbf{p}_1,\mathbf{p}_5,\mathbf{p}_9,\mathbf{p}_{13})\\
\mathbf{q}_2 &= Bez(u,\mathbf{p}_8,\mathbf{p}_9,\mathbf{p}_{10},\mathbf{p}_{11}) \quad \Leftrightarrow & \mathbf{r}_2 &= Bez(v,\mathbf{p}_2,\mathbf{p}_6,\mathbf{p}_{10},\mathbf{p}_{14})\\
\mathbf{q}_3 &= Bez(u,\mathbf{p}_{12},\mathbf{p}_{13},\mathbf{p}_{14},\mathbf{p}_{15}) & \mathbf{r}_3 &= Bez(v,\mathbf{p}_3,\mathbf{p}_7,\mathbf{p}_{11},\mathbf{p}_{15})\\
\mathbf{x}(u,v) &= Bez(v,\mathbf{q}_0,\mathbf{q}_1,\mathbf{q}_2,\mathbf{q}_3) & \mathbf{x}(u,v) &= Bez(u,\mathbf{r}_0,\mathbf{r}_1,\mathbf{r}_2,\mathbf{r}_3)
\end{aligned}
$$



UCSD

Bézier Patch: Matrix Form

$$\mathbf{U} = \begin{bmatrix} u^3 \\ u^2 \\ u \\ 1 \end{bmatrix} \quad \mathbf{V} = \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix} \quad \mathbf{B}_{Bez} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} = \mathbf{B}_{Bez}^T$$

$$\mathbf{C}_x = \mathbf{B}_{Bez}^T \mathbf{G}_x \mathbf{B}_{Bez}$$
$$\mathbf{C}_y = \mathbf{B}_{Bez}^T \mathbf{G}_y \mathbf{B}_{Bez}$$
$$\mathbf{C}_z = \mathbf{B}_{Bez}^T \mathbf{G}_z \mathbf{B}_{Bez}$$

$$\mathbf{G}_x = \begin{bmatrix} p_{0x} & p_{1x} & p_{2x} & p_{3x} \\ p_{4x} & p_{5x} & p_{6x} & p_{7x} \\ p_{8x} & p_{9x} & p_{10x} & p_{11x} \\ p_{12x} & p_{13x} & p_{14x} & p_{15x} \end{bmatrix}, \ \mathbf{G}_y = \cdots, \ \mathbf{G}_z = \cdots$$

$$\boxed{\mathbf{x}(u,v) = \begin{bmatrix} \mathbf{V}^T \mathbf{C}_x \mathbf{U} \\ \mathbf{V}^T \mathbf{C}_y \mathbf{U} \\ \mathbf{V}^T \mathbf{C}_z \mathbf{U} \end{bmatrix}}$$

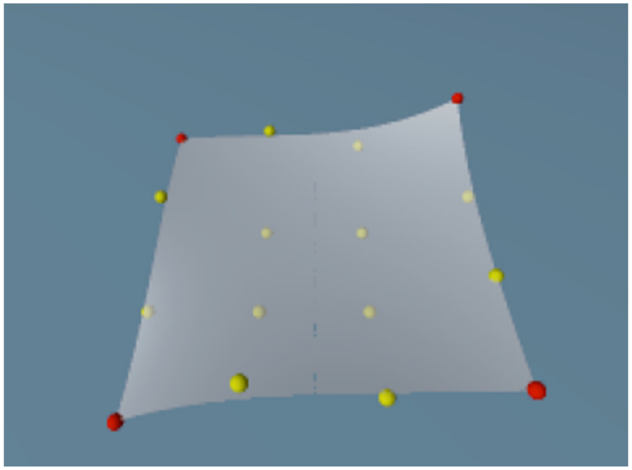
Bézier Patch: Matrix Form

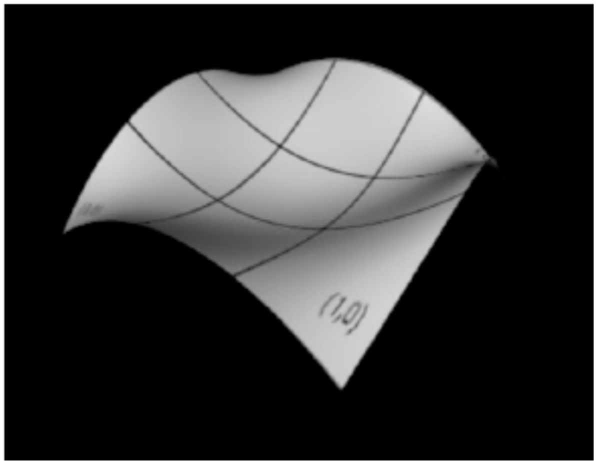
- $\mathbf{C}_x$ stores the coefficients of the bicubic equation for x
- $\mathbf{C}_y$ stores the coefficients of the bicubic equation for y
- $\mathbf{C}_z$ stores the coefficients of the bicubic equation for z
- $\mathbf{G}_x$ stores the geometry (x components of the control points)
- $\mathbf{G}_y$ stores the geometry (y components of the control points)
- $\mathbf{G}_z$ stores the geometry (z components of the control points)
- $\mathbf{B}_{Bez}$ is the basis matrix (Bézier basis)
- **U** and **V** are the vectors formed from the powers of u and v

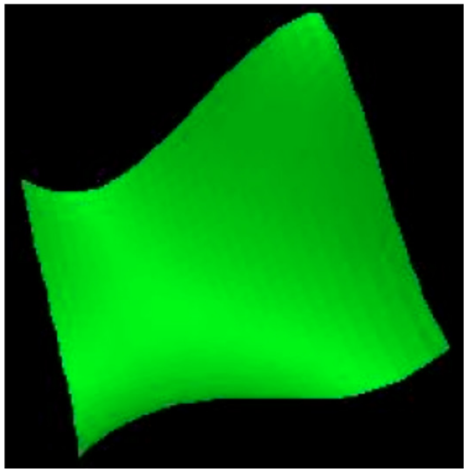
- Compact notation
- Leads to efficient method of computation
- Can take advantage of hardware support for 4x4 matrix arithmetic

Properties

- Convex hull: any point on the surface will fall within the convex hull of the control points
- Interpolates 4 corner points
- Approximates other 12 points, which act as "handles"
- The boundaries of the patch are the Bézier curves defined by the points on the mesh edges
- The parametric curves are all Bézier curves

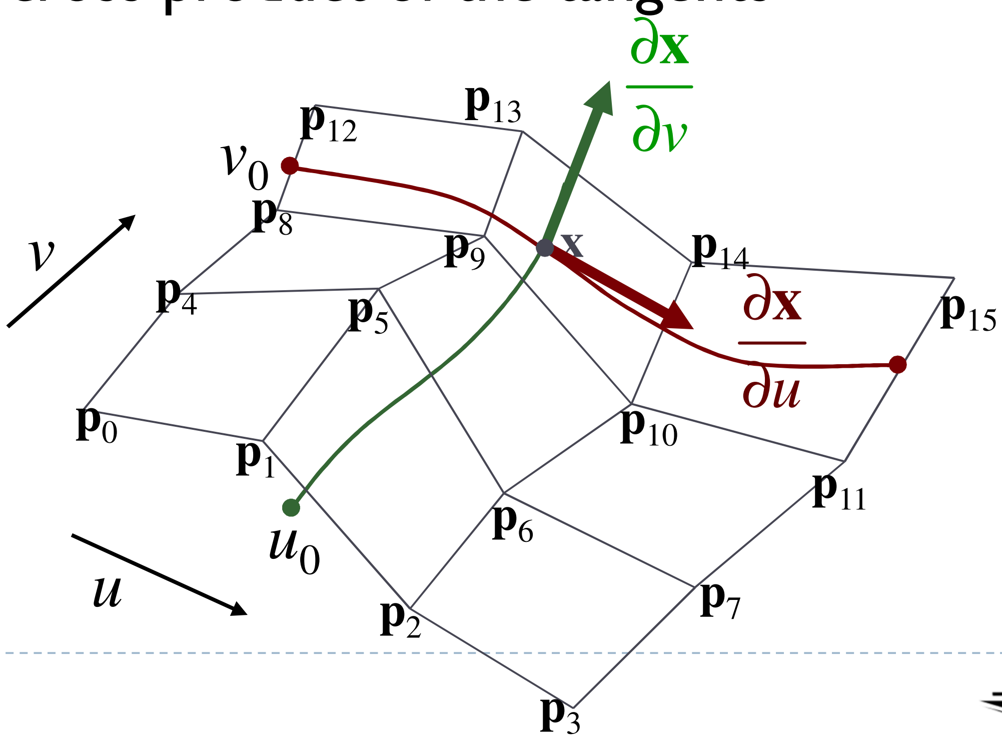






Tangents of a Bézier patch

- Remember parametric curves $\mathbf{x}(u,v_0)$, $\mathbf{x}(u_0,v)$ where v_0, u_0 is fixed
- Tangents to surface = tangents to parametric curves
- Tangents are partial derivatives of $\mathbf{x}(u,v)$
- Normal is cross product of the tangents



Tangents of a Bézier patch

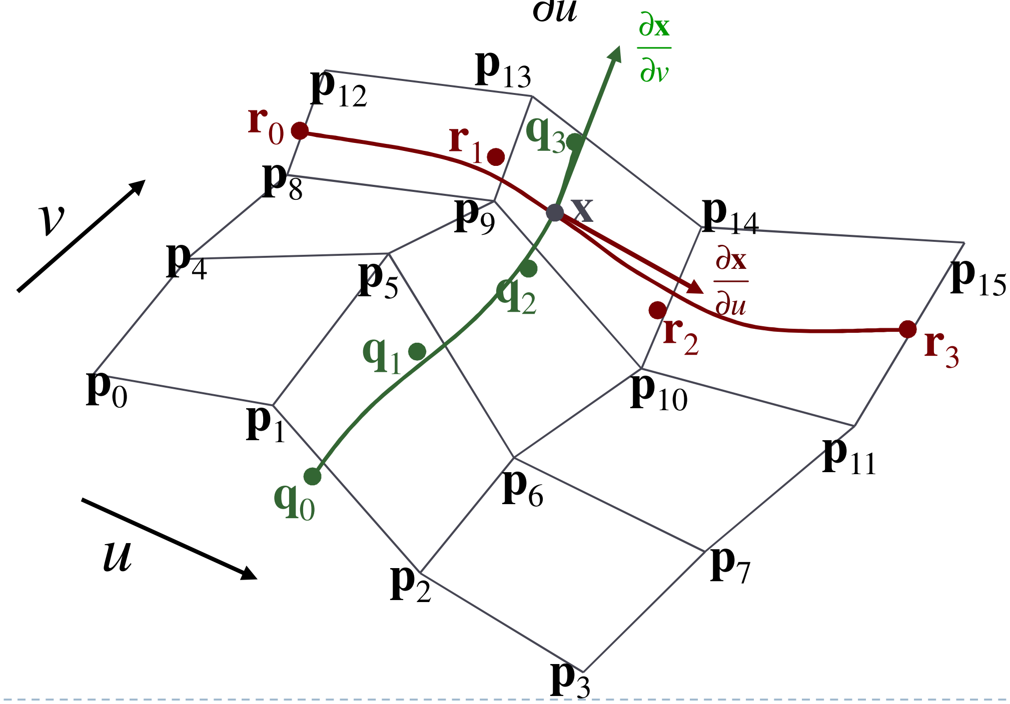
$$\mathbf{q_0} = Bez(u, \mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3) \qquad \mathbf{r_0} = Bez(v, \mathbf{p}_0, \mathbf{p}_4, \mathbf{p}_8, \mathbf{p}_{12})$$

$$\mathbf{q}_1 = Bez(u, \mathbf{p}_4, \mathbf{p}_5, \mathbf{p}_6, \mathbf{p}_7) \qquad \mathbf{r}_1 = Bez(v, \mathbf{p}_1, \mathbf{p}_5, \mathbf{p}_9, \mathbf{p}_{13})$$

$$\mathbf{q}_2 = Bez(u, \mathbf{p}_8, \mathbf{p}_9, \mathbf{p}_{10}, \mathbf{p}_{11}) \qquad \mathbf{r}_2 = Bez(v, \mathbf{p}_2, \mathbf{p}_6, \mathbf{p}_{10}, \mathbf{p}_{14})$$

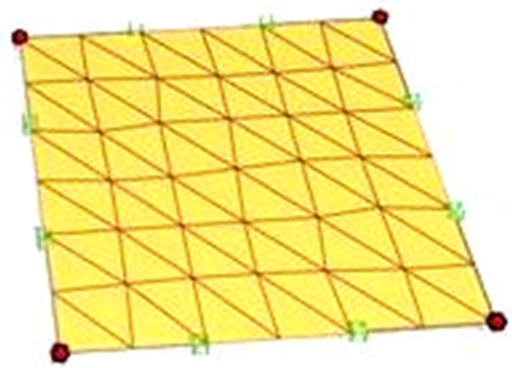
$$\mathbf{q}_3 = Bez(u, \mathbf{p}_{12}, \mathbf{p}_{13}, \mathbf{p}_{14}, \mathbf{p}_{15}) \qquad \mathbf{r}_3 = Bez(v, \mathbf{p}_3, \mathbf{p}_7, \mathbf{p}_{11}, \mathbf{p}_{15})$$

$$\frac{\partial \mathbf{x}}{\partial v}(u,v) = Bez'(v, \mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2, \mathbf{q}_3) \qquad \frac{\partial \mathbf{x}}{\partial u}(u,v) = Bez'(u, \mathbf{r}_0, \mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3)$$



Tessellating a Bézier patch

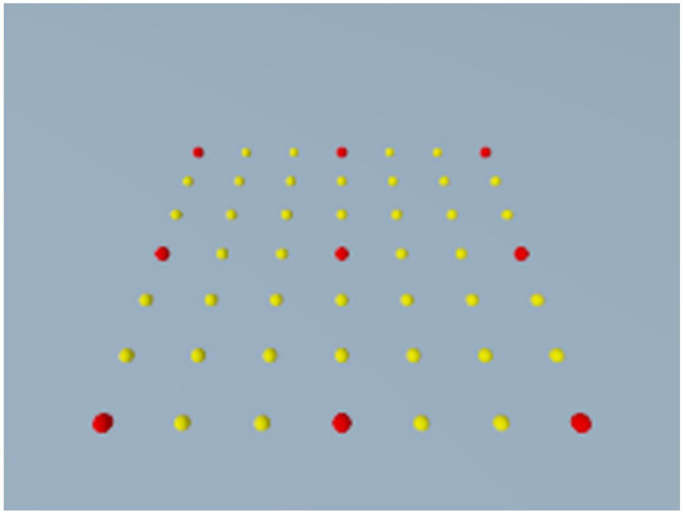
- Uniform tessellation is most straightforward
  - Evaluate points on a grid of u, v coordinates
  - Compute tangents at each point, take cross product to get per-vertex normal
  - Draw triangle strips with glBegin(GL_TRIANGLE_STRIP)



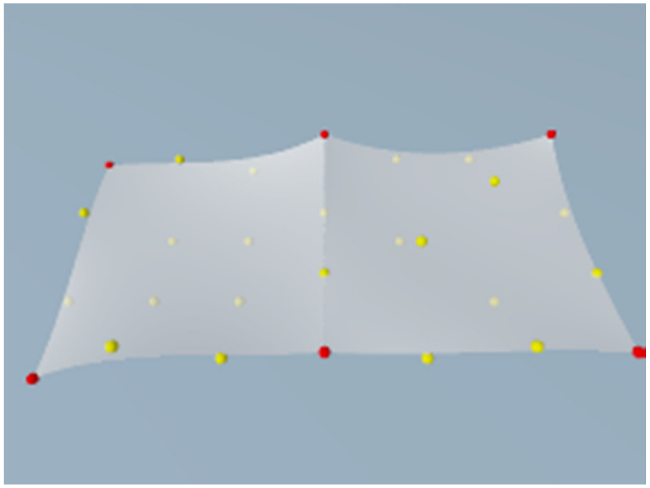
- Adaptive tessellation/recursive subdivision
  - Potential for "cracks" if patches on opposite sides of an edge divide differently
  - Tricky to get right, but can be done

Piecewise Bézier Surface

- Lay out grid of adjacent meshes of control points
- For C^0 continuity, must share points on the edge
  - Each edge of a Bézier patch is a Bézier curve based only on the edge mesh points
  - So if adjacent meshes share edge points, the patches will line up exactly
- But we have a crease…



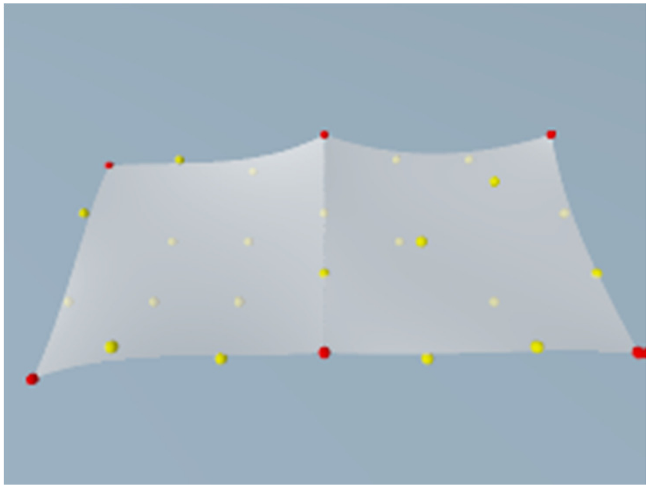
Grid of control points

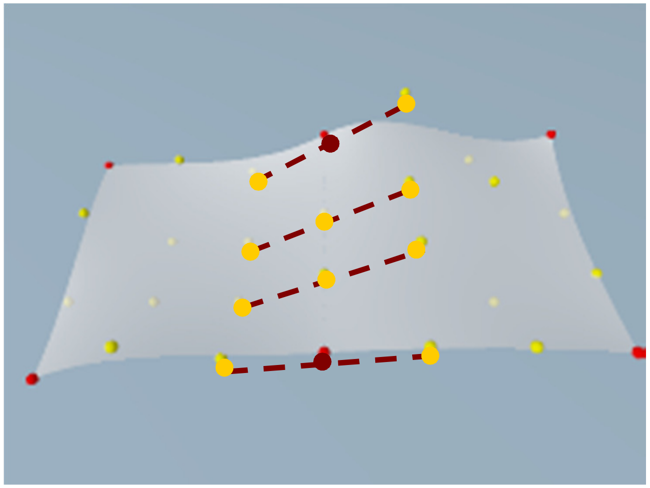


Piecewise Bézier surface

C^1 Continuity

- We want the parametric curves that cross each edge to have C^1 continuity
  - So the handles must be equal-and-opposite across the edge:



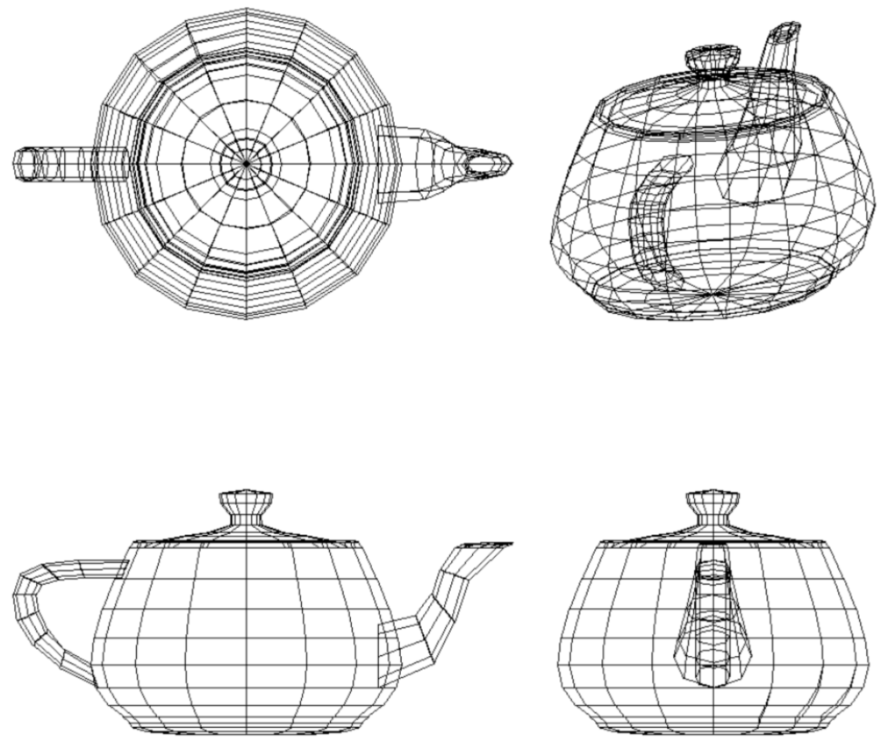


http://www.spiritone.com/~english/cyclopedia/patches.html

UCSD

Modeling With Bézier Patches

- Original Utah teapot, from Martin Newell's PhD thesis, consisted of 28 Bézier patches.
- The original had no rim for the lid and no bottom
- Later, four more patches were added to create a bottom, bringing the total to 32
- The data set was used by a number of people, including graphics guru Jim Blinn. In a demonstration of a system of his he scaled the teapot by .75, creating a stubbier teapot. He found it more pleasing to the eye, and it was this scaled version that became the highly popular dataset used today.



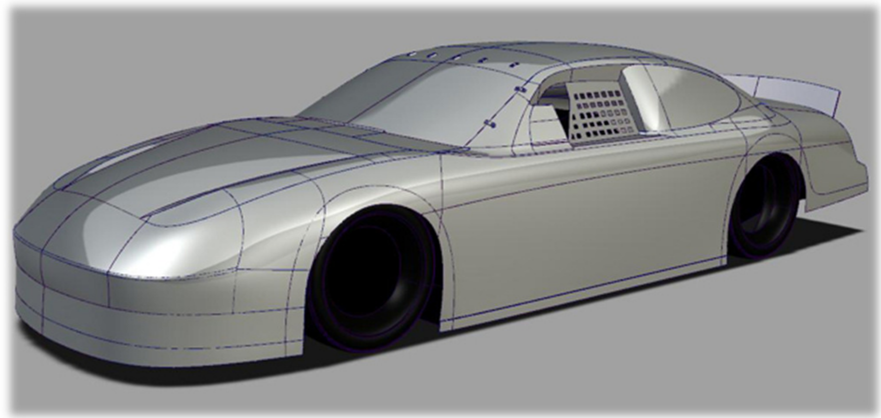
Source: http://www.holmes3d.net/graphics/teapot/

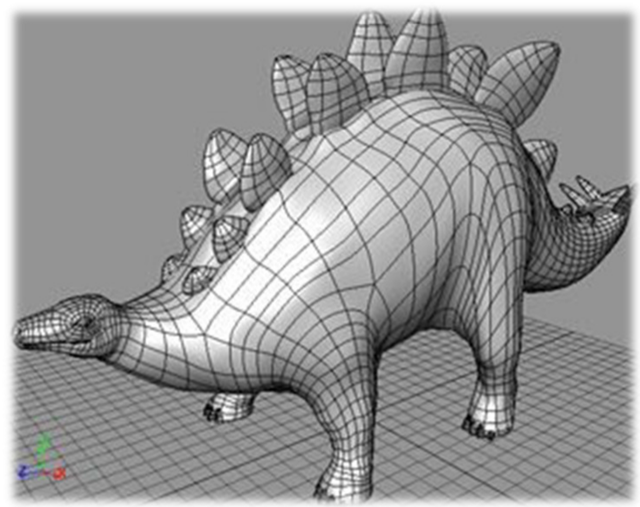
Overview

- Bi-linear patch
- Bi-cubic Bézier patch
- Advanced parametric surfaces

Problems with Bezier and NURBS Patches

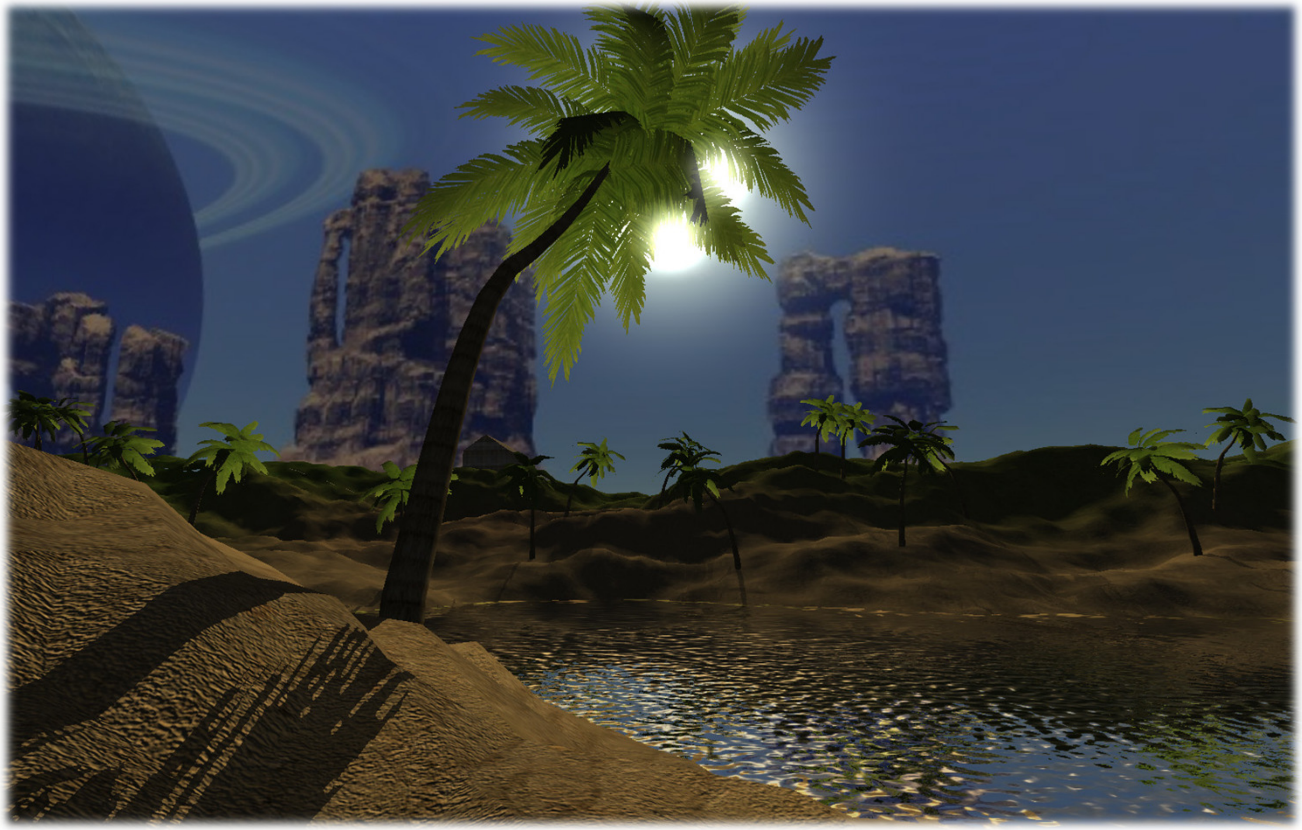
- **NURBS surfaces are versatile**
  - Conic sections
  - Can blend, merge, trim…
- **But:**
  - Any surface will be made of quadrilateral patches (quadrilateral topology)
- **This makes it hard to**
  - Join or abut curved pieces
  - Build surfaces with complex topology or structure





GLSL

- Real Time 3D Demo C++/OpenGL/GLSL Engine
  http://www.youtube.com/watch?v=9N-kgCqy2xs



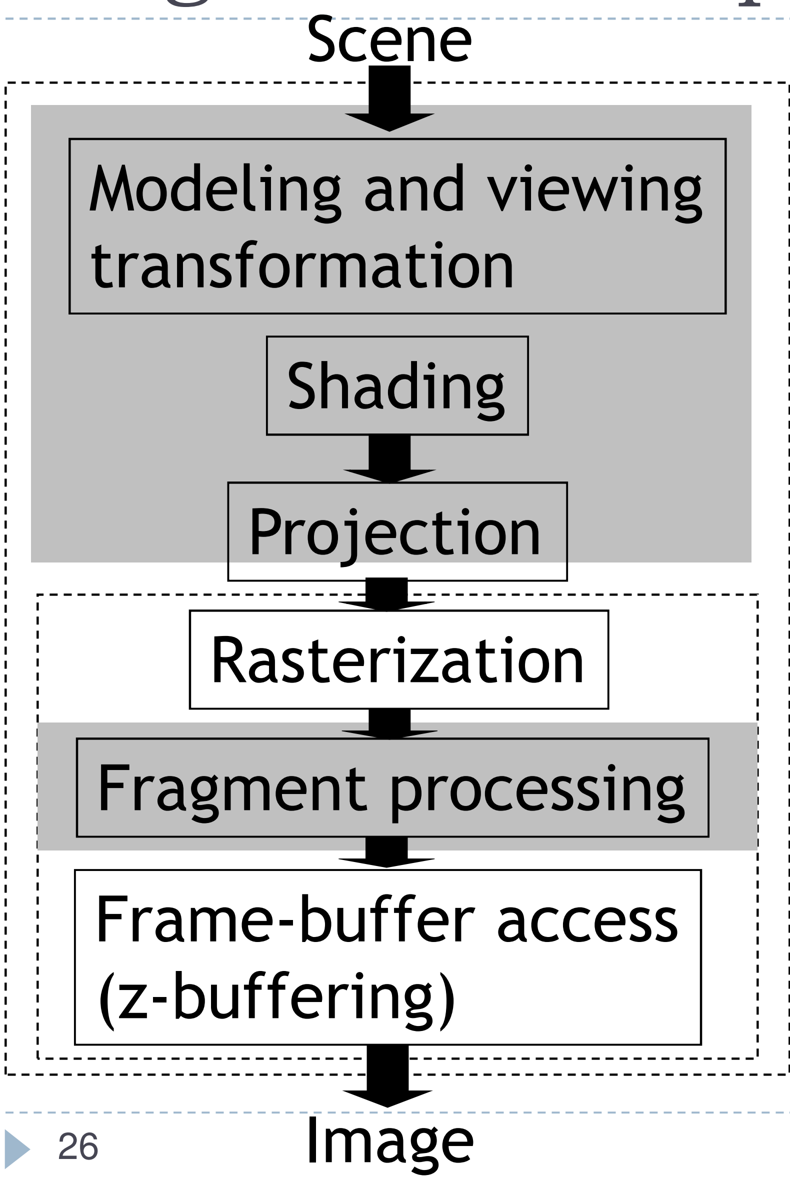
Lecture Overview

- Programmable Shaders
  - Vertex Programs
  - Fragment Programs
  - GLSL

Shader Programs

- Programmable shaders consist of shader programs
- Written in a shading language
  - Syntax similar to C language
- Each shader is a separate piece of code in a separate ASCII text file
- Shader types:
  - Vertex shader
  - Tessellation shader
  - Geometry shader
  - Fragment shader (a.k.a. pixel shader)
- The programmer can provide any number of shader types to work together to achieve a certain effect
- If a shader type is not provided, OpenGL's fixed-function pipeline is used

Programmable Pipeline



- Executed once per vertex:
  - Vertex Shader
  - Tessellation Shader
  - Geometry Shader

- Executed once per fragment:
  - Fragment Shader

Vertex Shader

- Executed once per vertex
- Cannot create or remove vertices
- Does not know the primitive it belongs to
- Replaces functionality for
  - Model-view, projection transformation
  - Per-vertex shading
- If you use a vertex program, you need to implement behavior for the above functionality in the program!
- Typically used for:
  - Character animation
  - Particle systems

Tessellation Shader

- Executed once per primitive
- Generates new primitives by subdividing each line, triangle or quad primitive
- Typically used for:
  - Adapting visual quality to the required level of detail
    - For instance, for automatic tessellation of Bezier curves and surfaces
  - Geometry compression: 3D models stored at coarser level of resolution, expanded at runtime
  - Allows detailed displacement maps for less detailed geometry

Geometry Shader

- Executed once per primitive (triangle, quad, etc.)
- Can create new graphics primitives from output of tessellation shader (e.g., points, lines, triangles)
  - Or can remove the primitive
- Typically used for:
  - Per-face normal computation
  - Easy wireframe rendering
  - Point sprite generation
  - Shadow volume extrusion
  - Single pass rendering to a cube map
  - Automatic mesh complexity modification (depending on resolution requirements)

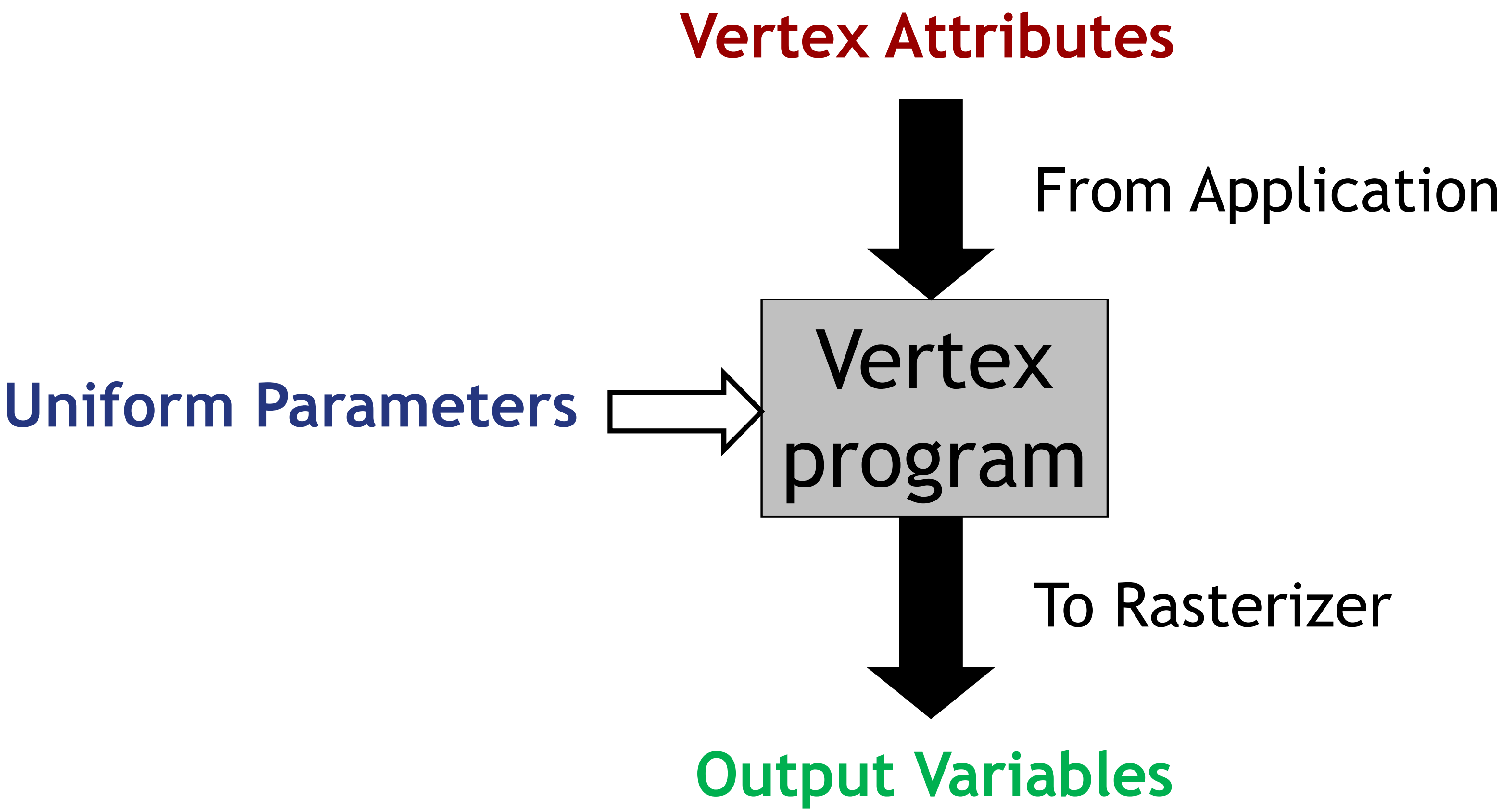
Fragment Shader

- A.k.a. Pixel Shader
- Executed once per fragment
- Cannot access other pixels or vertices
  - Makes execution highly parallelizable
- Computes color, opacity, z-value, texture coordinates
- Typically used for:
  - Per-pixel shading (e.g., Phong shading)
  - Advanced texturing
  - Bump mapping
  - Shadows

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[];
```

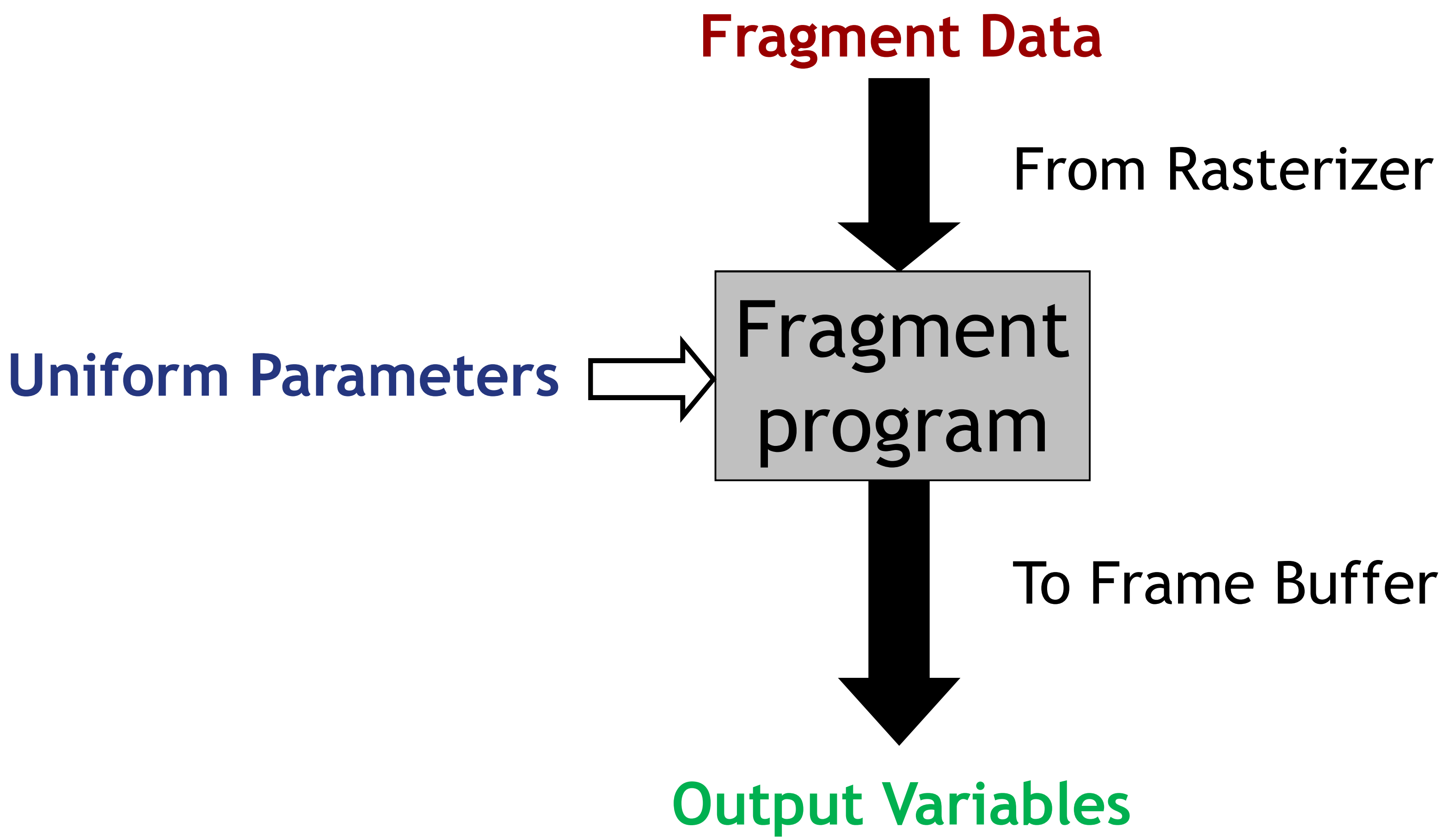
  Any pre-defined output variable that you do not overwrite will have the value of the OpenGL state.
- User-defined `varying` output variables, e.g.:

```
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

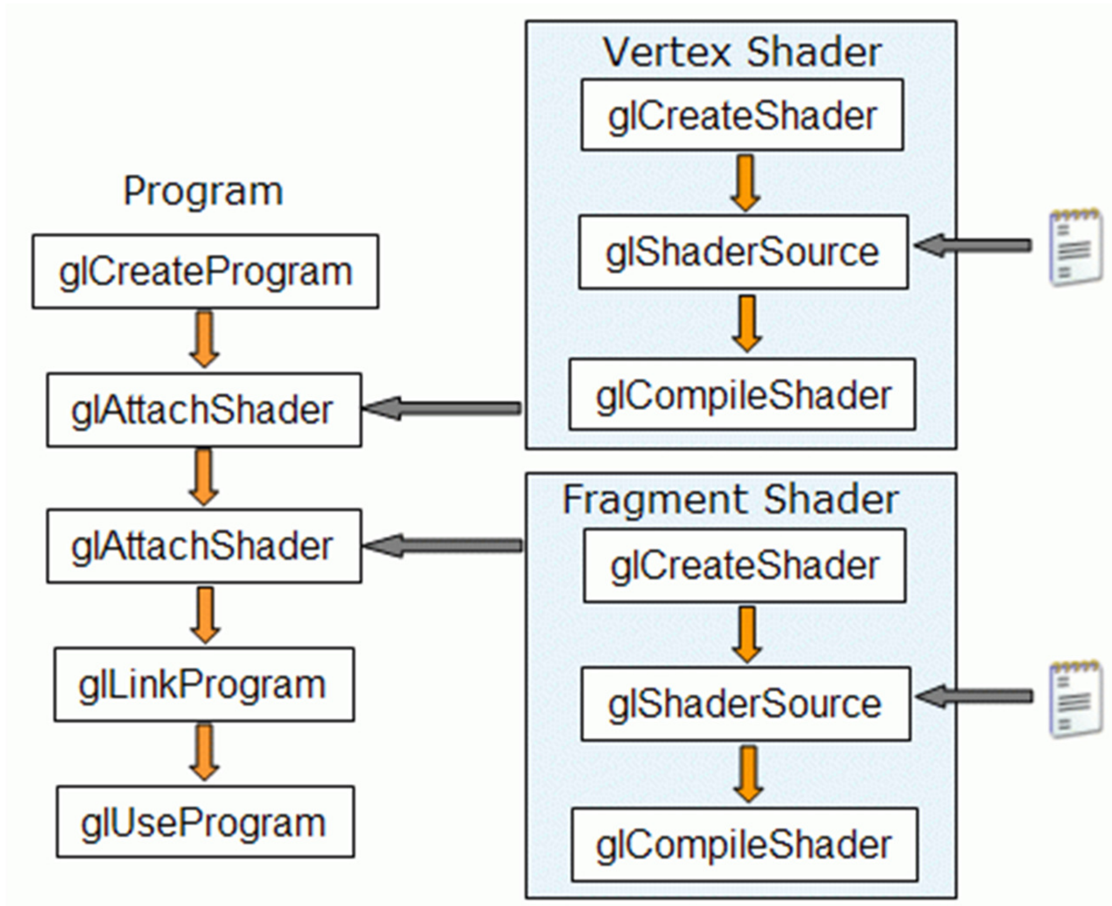
Uniform Parameters

- 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!

Creating Shaders in OpenGL



Source: Gabriel Zachmann, Clausthal University

Tutorials and Documentation

- OpenGL and GLSL Specifications
  - https://www.opengl.org/registry/
- GLSL Tutorials
  - http://www.lighthouse3d.com/opengl/glsl/
  - http://www.clockworkcoders.com/oglsl/tutorials.html
- OpenGL Programming Guide (Red Book)
  - http://www.glprogramming.com/red/
- OpenGL Shading Language (Orange Book)
  - http://wiki.labomedia.org/images/1/10/Orange_Book_-_OpenGL_Shading_Language_2nd_Edition.pdf
- OpenGL 4.5 API Reference Card
  - https://www.opengl.org/sdk/docs/reference_card/opengl45-reference-card.pdf