CMT107 Visual Computing

VI.1 Texture Mapping

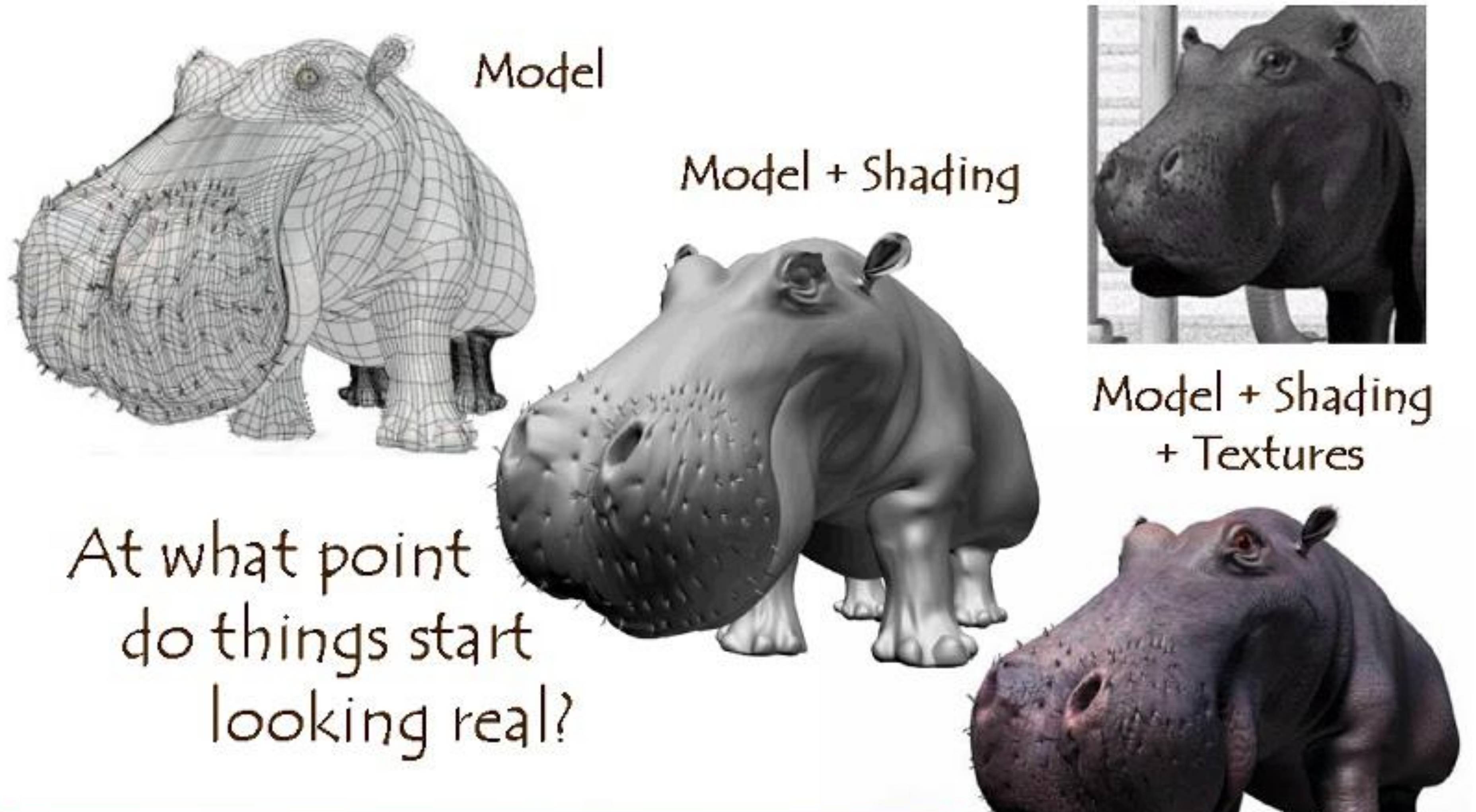
Xianfang Sun

School of Computer Science & Informatics
Cardiff University

Overview

- Texture mapping
  - Texture coordinates
  - Aliasing effects and MIP mapping
- Bump mapping
- Displacement mapping
- Light maps
- Shadow maps
- Texture Mapping in OpenGL

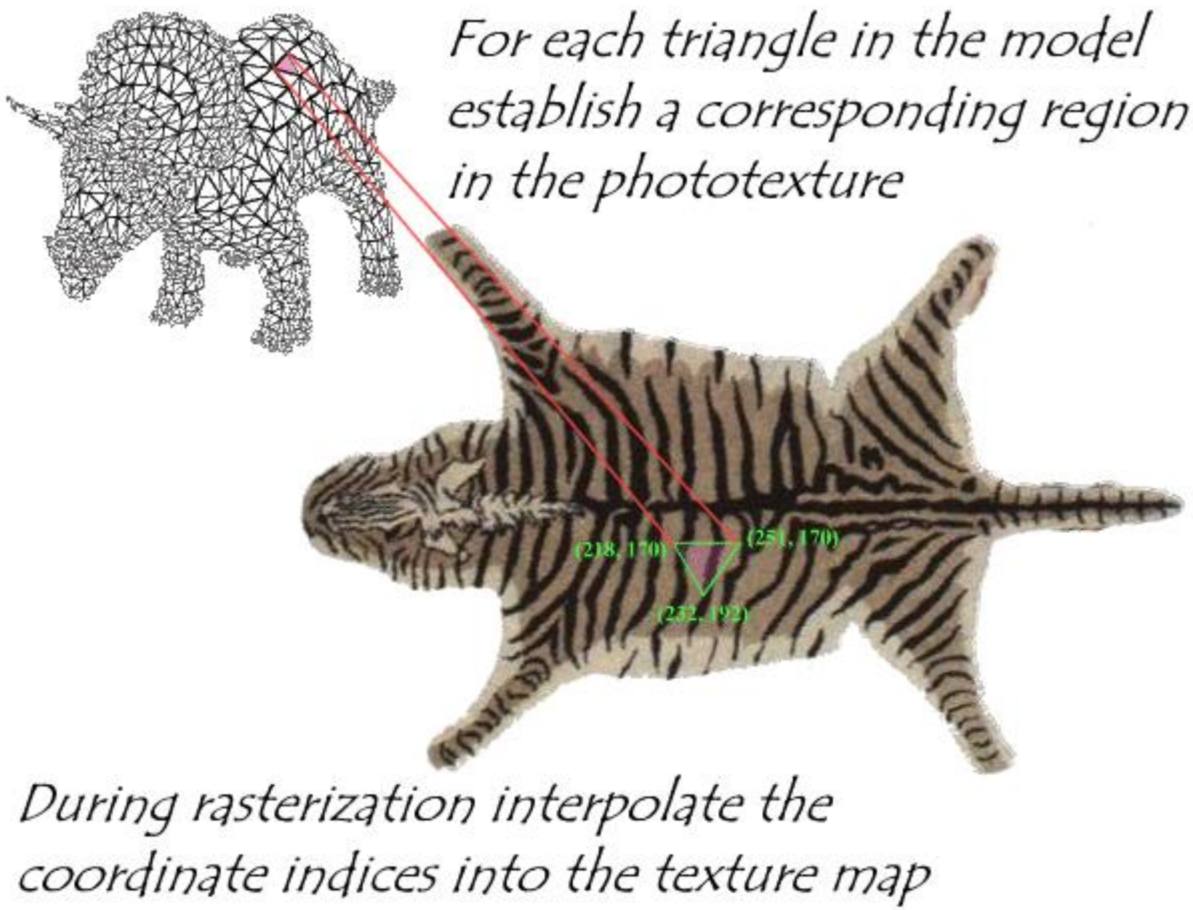
From Shading to Texture



For more info on the computer artwork of Jeremy Birn
see http://www.3drender.com/jbirn/productions.html

Texture

- Visual appearance of objects can be enhanced by textures
- The concept is simple



Texture Coordinates

- For each vertex specify *texture coordinates* $(s,t) \in [0,1]^2$
  - Canonical position of pixel in texture for vertex
  - For each point p on the 3D polygon, corresponding texture coordinates (s,t) are required

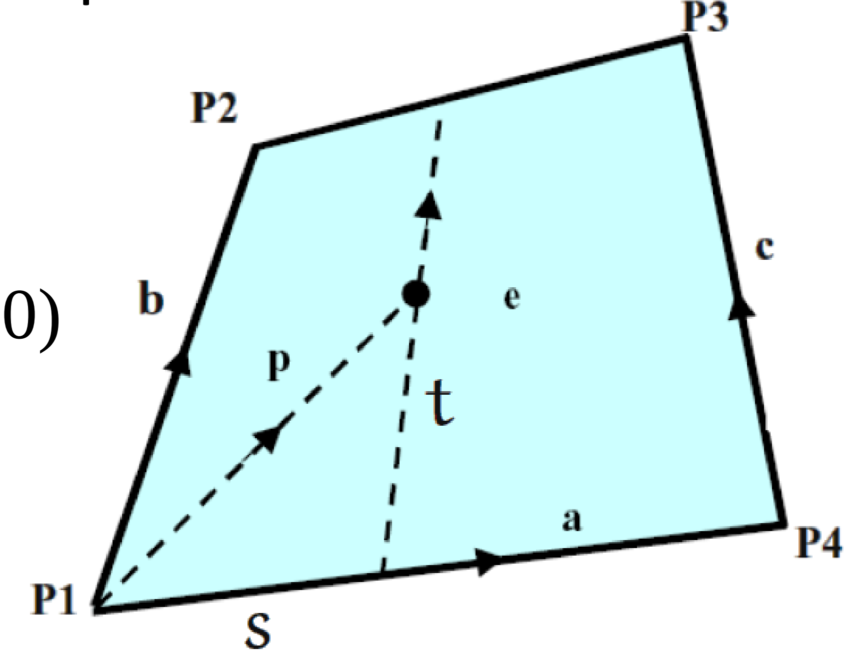
  → *Bilinearly interpolate* texture coordinates in 3D
- Texture coordinates for point on quad

$$p = sa + te$$

$$e = b + s(c - b)$$

$$p = sa + tb + st(c - b)$$

→ Solve for (s,t) (assuming $(0,0)$ is texture coordinate of P_1)



Alias Effects

- One major problem of texture: *alias effects*
  - Caused by *undersampling*; results in *unreal artefacts*

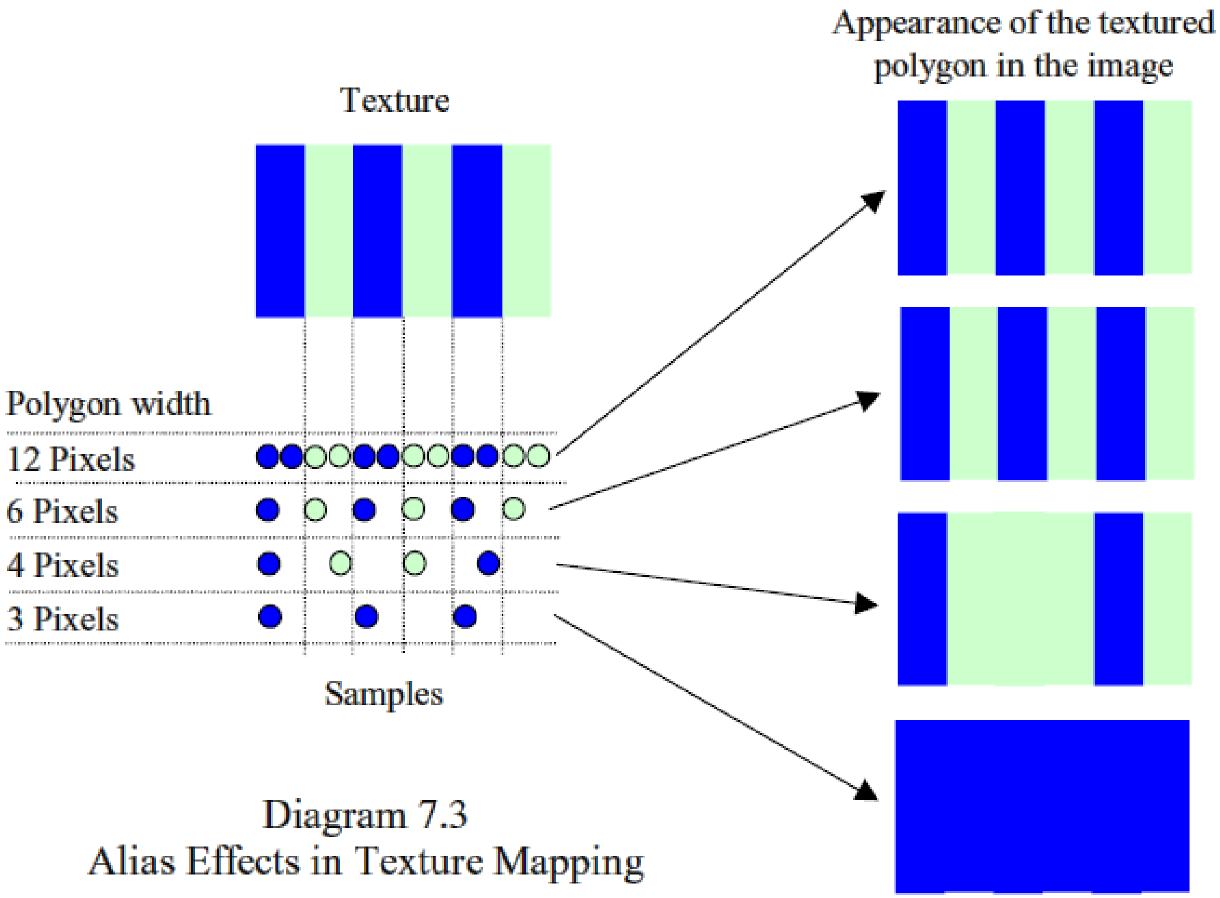


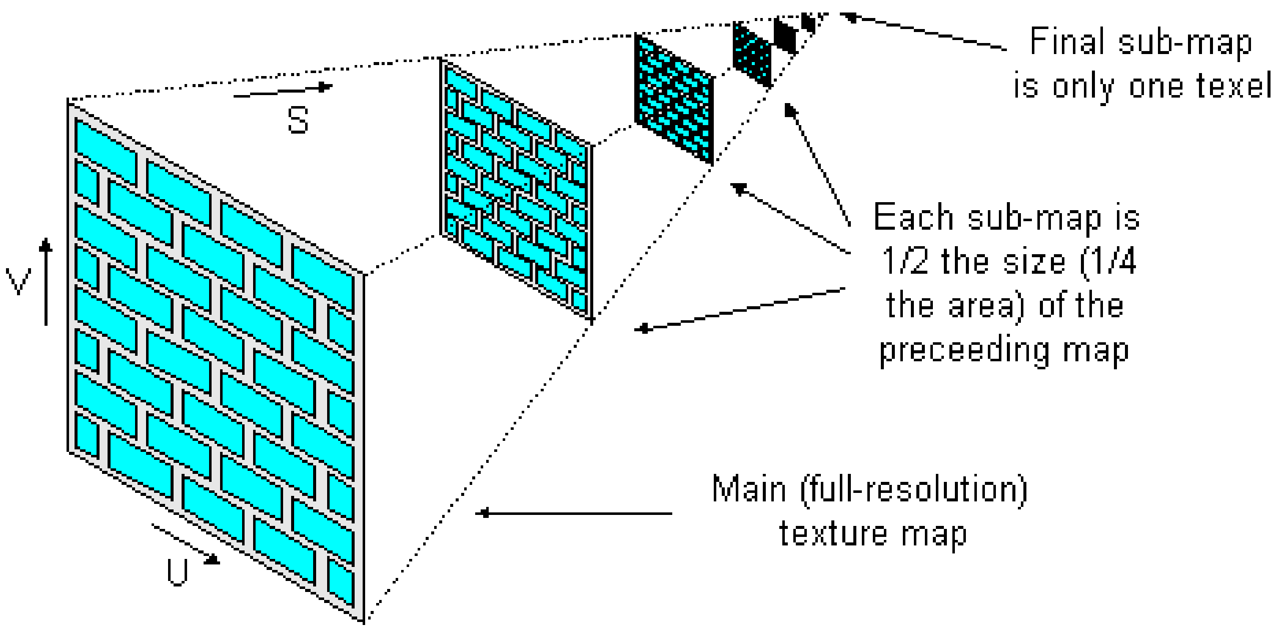
Diagram 7.3
Alias Effects in Texture Mapping

Anti-aliasing

- Similar to untextured images use *anti-aliasing* technique
- Most successful approach: *supersampling*
  - Compute picture at a *higher resolution*
  - *Average* the supersamples to find pixel colour
  - This blurs boundaries, but leaves coherent areas of colour unchanged
  - Works well for polygons, but requires a lot of computations and does not work for line drawings
- Other approaches: convolution filtering (see image processing)

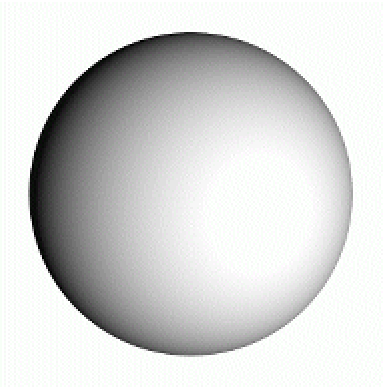
MIP Mapping

- Popular technique of precomputing / prefiltering to address alias effects (MIP = multum in parvo; much in little)
- Basic idea: construct a *pyramid of images* for *different* texture sizes (prefiltered and resampled)
  - Pick texture image suitable for size or interpolate between texture images

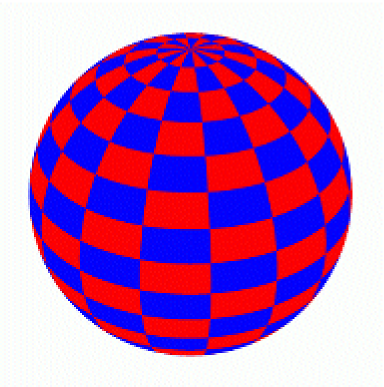


Generalising Texture Mapping

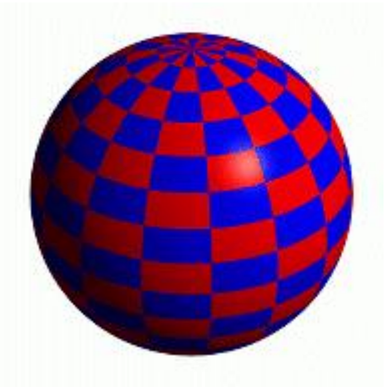
- So far: texture is a *label* (colour) for each pixel
- Can use it to modify other things
  - E.g. use it for *illumination* to adjust material properties (all light types or only some of them)



Material



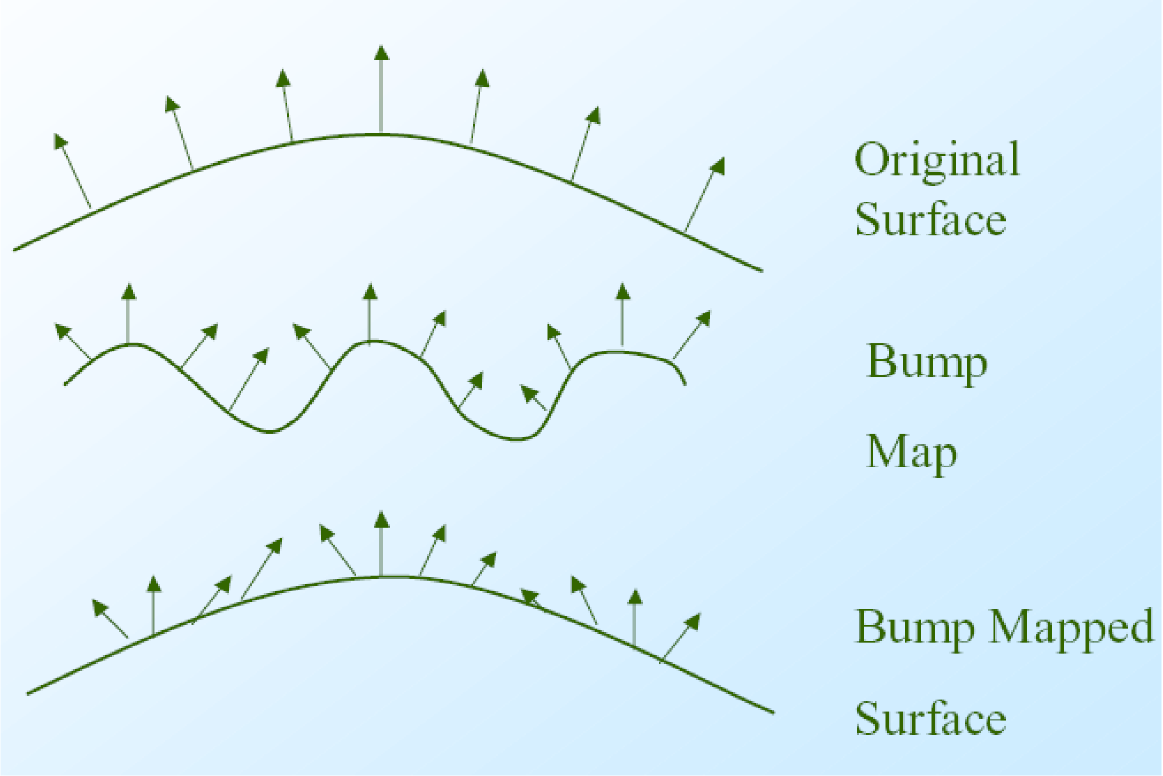
Texture as label



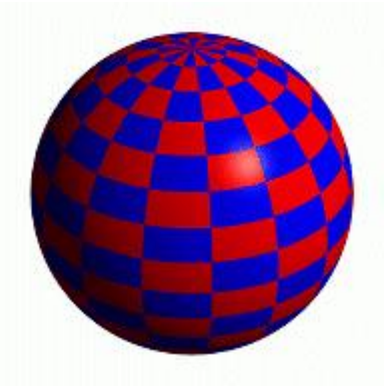
Texture as material

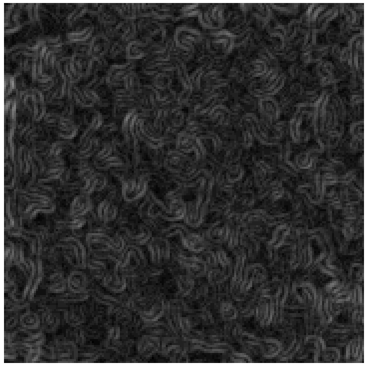
Bump Mapping

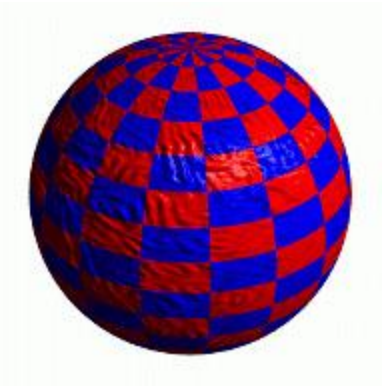
- Texture can be used to alter *surface normals* of an object
  - Does not change shape, but illumination computation
  - Changes in texture (partial derivatives) tell how to change the “height” of the normals



Bump Map Example



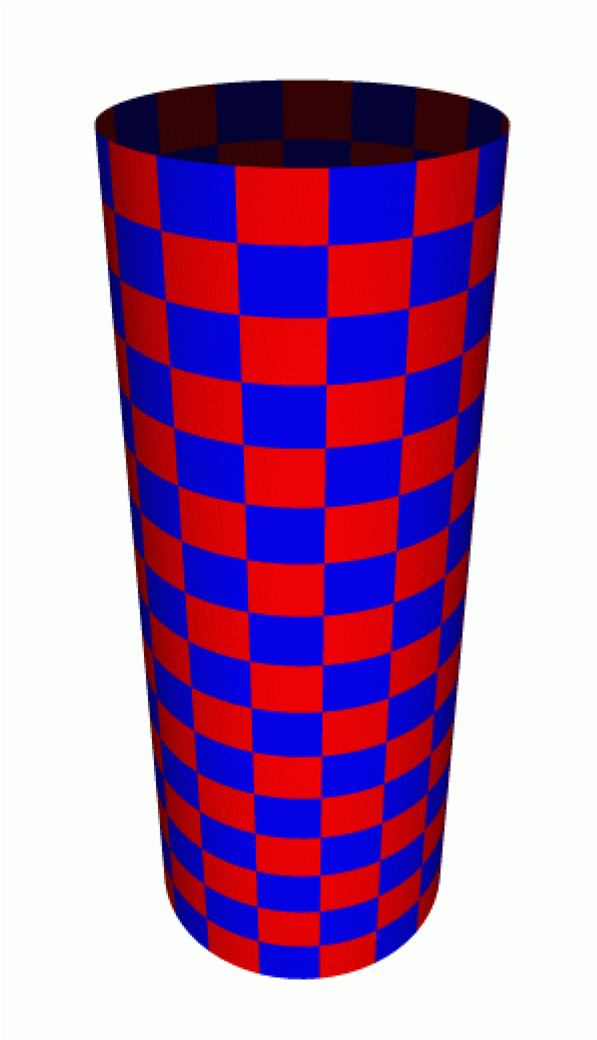


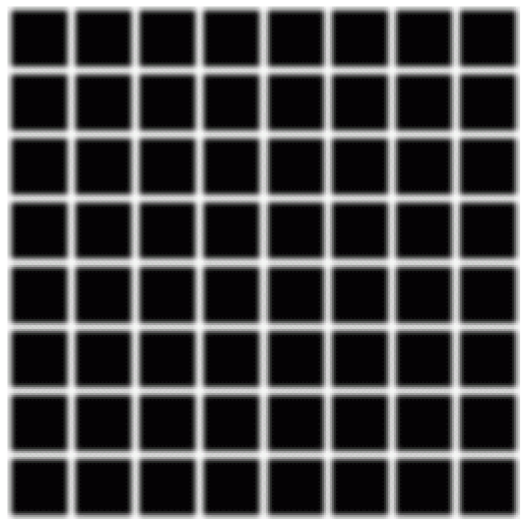


Sphere w/Texture Bump Map Bumpy Sphere

- As we do not change the shape, the silhouette does not change
  - Use only for small bumps
  - Requires illumination computation for each pixel (Phong shading, ray tracing, . . .)

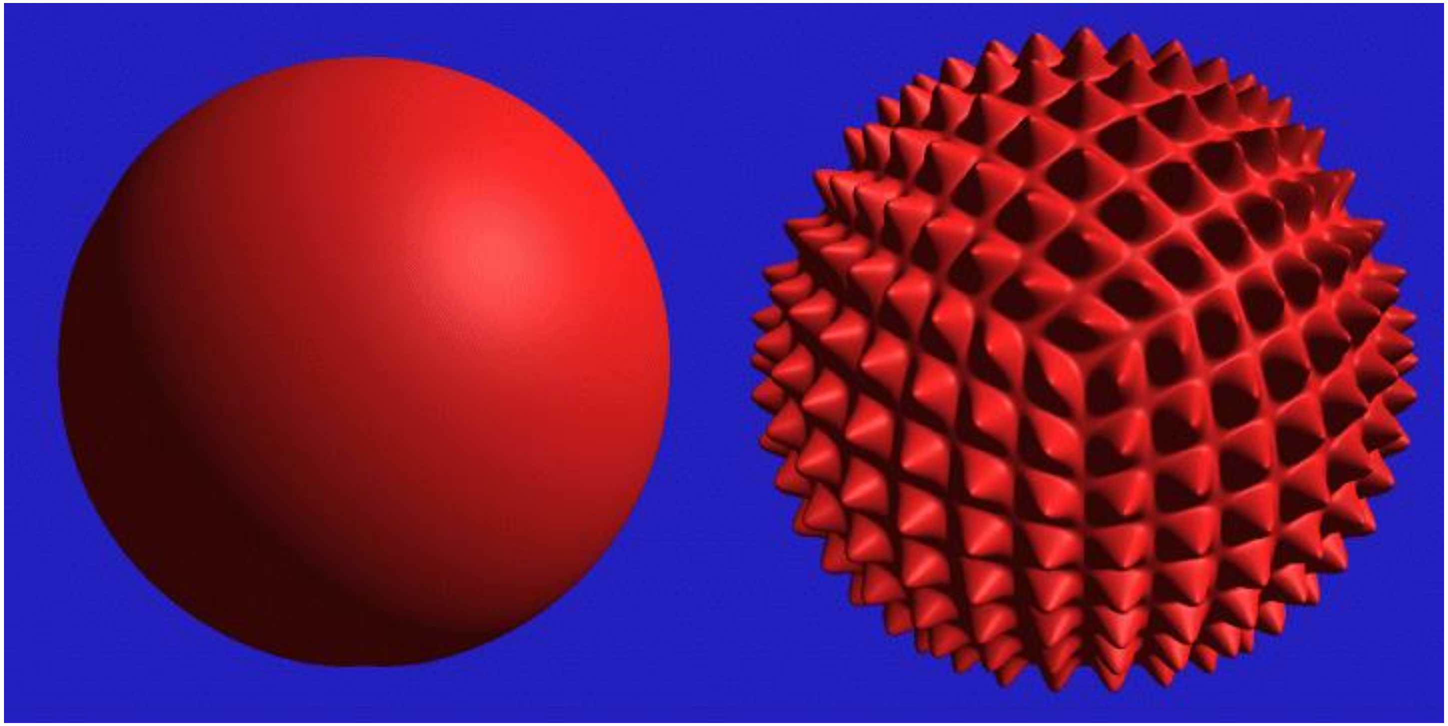
Another Bump Map Example





Displacement Mapping

- Use texture map to *move* surface points



Light Maps

- In Quake *texture* and *light maps* are used
  - Light map contains precomputed illumination at low resolution
  - Multiply light map with texture map at run-time (and cache it)

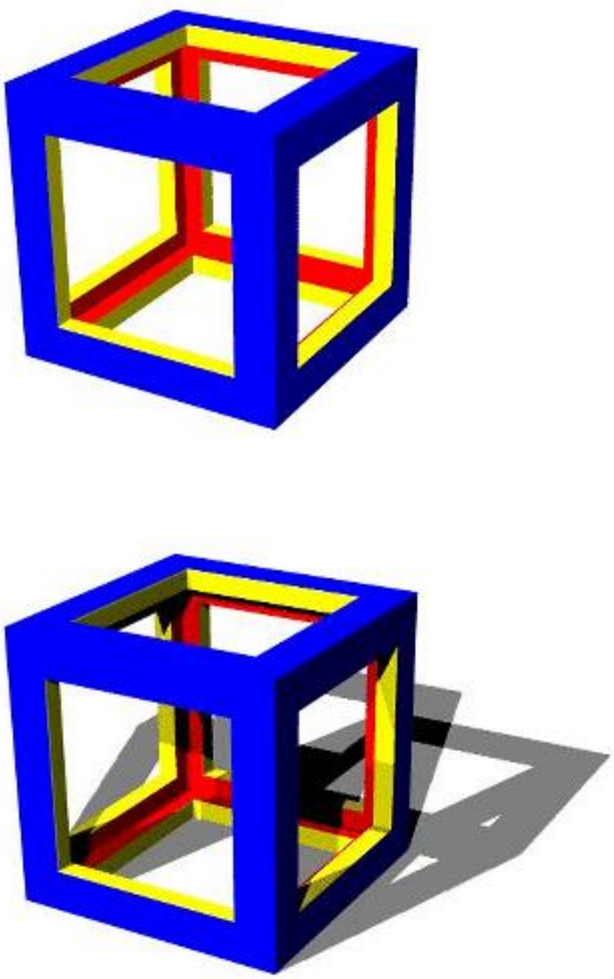
Only Texture Map

Texture and Light Map

Shadow Maps

- Generate *shadows* using texture maps
  - Render scene from the *viewpoint of each light source* and only keep *depth buffer* values in *shadow buffers*
  - When shading each pixel (illumination computation per pixel):
    - Compute vector *L* from visible point to light source (needed for illumination computation)
    - Compute the length of *L*
    - *Compare* this length with corresponding value in the *shadow buffers*
    - If the shadow buffer value is *less*, then the point is in the shadow and we can ignore the light source

Shadow Map Example



Texture Mapping in OpenGL

- Use Texture and TextureIO to apply a texture
  1. Create a texture object using TextureIO
     - TextureIO.newTexture(File, boolean);
  2. Indicate how the texture is to be applied to each pixel
     - Texture.setTexParameteri (…)
  3. Draw the scene, supplying both texture and geometric coordinates; send the coordinates to vertex shader, and send texture sampler to fragment shader
     - Texture.getImageTexCoords().top() …

Texture Mapping in OpenGL

- Using OpenGL Core functions to apply a texture
  1. Create a texture object and specify a texture for that object
     - glGenTextures(…)
     - glBindTexture(…)
     - glTexImage2D(…)
  2. Indicate how the texture is to be applied to each pixel
     - glTexParameteri(…)
  3. Enable texture mapping
     - glEnable(GL_TEXTURE_2D)
  4. Draw the scene, supplying both texture and geometric coordinates; send the coordinates to vertex shader, and send texture sampler to fragment shader

- Step 0: Read in texture image

Texture Object

- Texture objects store texture data and keep it readily available for usage. Many texture objects can be generated.
- Generate identifiers for texture objects first

```
int texids[n];
glGenTextures(n, texids)
```

  - `n`: the number of texture objects identifiers to generate
  - `texids`: an array of unsigned integers to hold texture object identifiers
- Bind a texture object as the current texture

```
glBindTexture(target, identifier)
```

  - target: can be GL_TEXTURE_1D, GL_TEXTURE_2D, or GL_TEXTURE 3D
  - identifier: a texture object identifier
- Specify texture image

```
glTexImage2D(target, level, internalFormat,
             width, height, border, format, type, data);
```

Texture Object Example Code

```
int texids[] = new int[1];
ByteBuffer texImg = readImage("textures/Day.png");

glGenTextures(1, texids, 0);
glBindTexture(GL_TEXTURE_2D, texids[0]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB,
             texWidth, texHeight,0, GL_BGR,
             GL_UNSIGNED_BYTE, texImg);
```

Texture Parameters

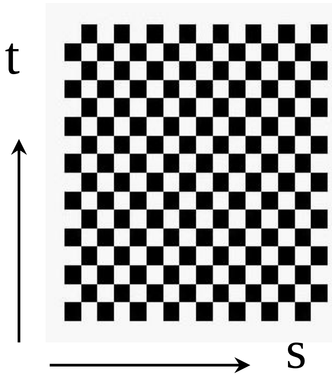
- OpenGL has a variety of parameters that determine how texture is applied.
  - Wrapping parameters determine what happens if s and t are outside the (0,1) range
  - Filter modes allow us to use area averaging instead of point samples
  - Environment parameters determine how texture mapping interacts with shading
  - Mipmapping allows us to use textures at multiple resolutions
- OpenGL Command

  glTexParameterf(target, pname, param);

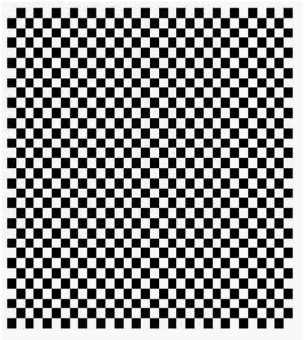
  - target: Specifies the target texture
  - pname: Specifies the symbolic name of a single-valued texture parameter
  - param: Specifies the value of pname.

Wrapping Modes

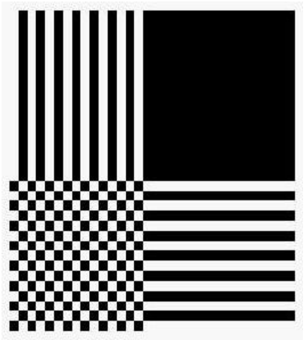
- Repeat: use s,t modulo 1
- Clamp: if s,t > 1 use 1, if s,t <0 use 0
    - glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT)
    - glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE)
    - GL_CLAMP_TO_BORDER, GL_MIRRORED_REPEAT…



texture



GL_REPEAT wrapping



GL_CLAMP_TO_EDGE wrapping

Texture Filtering

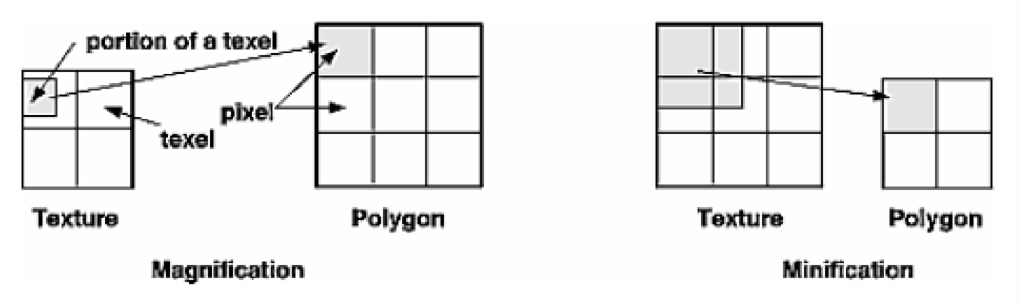
- A pixel may be mapped to a small portion of a texel or a collection of texels from the texture map. How to determine the color of the pixel?
- Magnification: when a pixel mapped to a small portion of a texel

  glTexParameteri(GL_TEXTURE_2D,
  GL_TEXTURE_MAG_FILTER, type);

  - type: GL_NEAREST or GL_LINEAR
- Minification: when a pixel mapped to many texels

  glTexParameteri(GL_TEXTURE_2D,
  GL_TEXTURE_MIN_FILTER, type);

  - type: GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_LINEAR, …



Shading and Texture Interaction

- You can specify how the texture-map colors are used to modify the pixel colors by setting environment parameters in old version of OpenGL

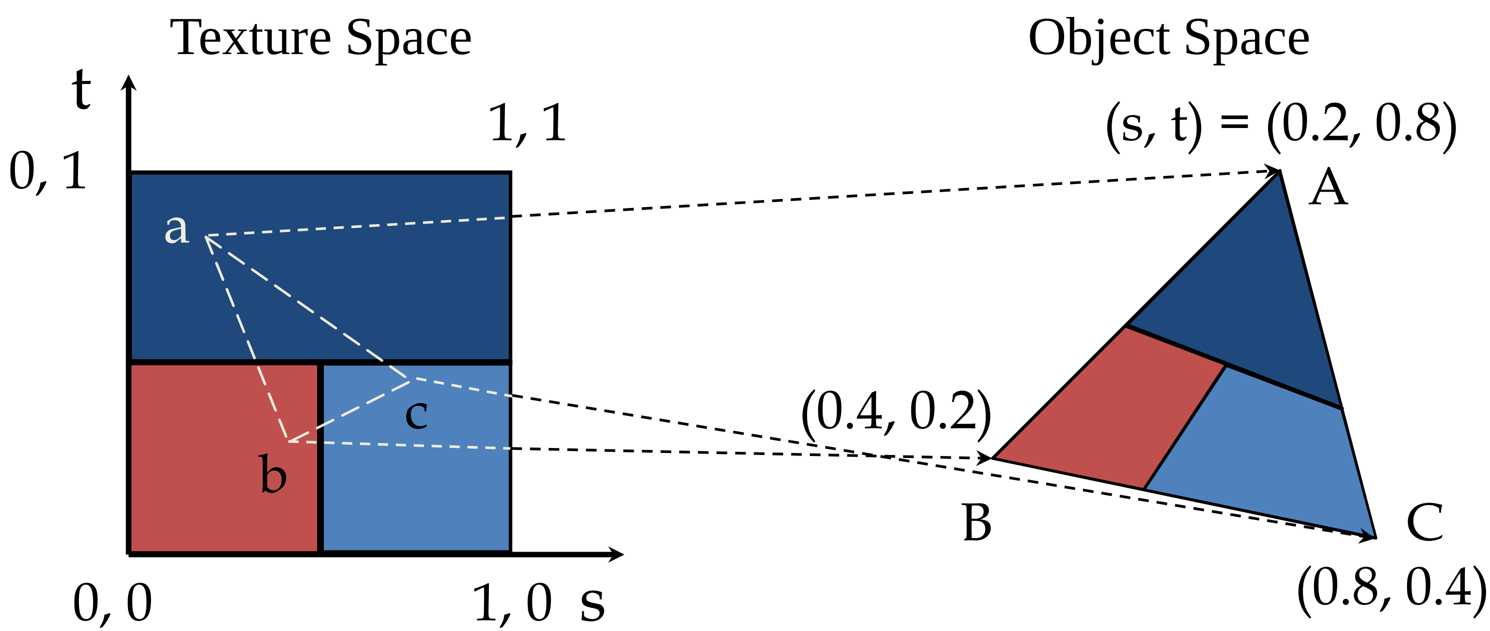
  glTexEnvi(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE, mode);

  mode values:

  - GL_REPLACE : replace pixel color with texture color
  - GL_BLEND : $C = C_f(1-C_t) + C_cC_t$,
    - C_f is the pixel color, C_t is the texture color, and C_c is some constant color
  - GL_MODULATE : $C = C_fC_t$ (Default)
  - More on OpenGL programming guide
- In the shader version of OpenGL, the interaction should be implemented in the fragment shader.

Assign Texture Coordinates

- Every point on a surface should have a texture coordinate (s, t) in texture mapping
- We often specify texture coordinates to polygon vertices and interpolate texture coordinates with the polygon
- Texture.getImageTexCoords() can be used to retrieve texture coordinates



Typical Code in Main Program

```
// Set the texture to be used
try {
    texture = TextureIO.newTexture(new File("WelshDragon.jpg"), false);
  } catch (IOException ex) {
    Logger.getLogger(getClass().getName()).log(Level.SEVERE, null, ex);
  }

  // Set texture coordinates
  float[] texCoord ={…};
  FloatBuffer textures = FloatBuffer.wrap(texCoord); gl.glGenBuffers(…);
  gl.glBindBuffer(…);
  gl.glBufferData(…);
  gl.glBufferSubData(…);

  // Send texture coordinates to vertex shader
  vTexCoord = gl.glGetAttribLocation( program, "vTexCoord" );
  gl.glEnableVertexAttribArray( vTexCoord );
  gl.glVertexAttribPointer( vTexCoord, 2, GL_FLOAT, false, 0, offsetSize);

  // Set the fragment shader texture sampler variable
  gl.glUniform1i( gl.glGetUniformLocation(program, "tex"), 0 );
```

Vertex Shader

```
#version 330 core

layout(location = 0) in vec4 vPosition;
layout(location = 1) in vec3 vColour;
layout(location = 2) in vec2 vTexCoord;

out vec4 color;
out vec2 texCoord;

uniform mat4 ModelView;
uniform mat4 Projection;

void main()
{
    gl_Position = Projection * ModelView * vPosition;
    texCoord    = vTexCoord;

    color.rgb = vColour;
    color.a = 1.0;
}
```

Fragment Shader

```
#version 330 core

in  vec4 color;
in  vec2 texCoord;

out vec4 fColor;

uniform sampler2D tex;

void main()
{
    fColor = color* texture( tex, texCoord );
}
```

More details in the Labs...

Summary

- Describe the principle of texture maps. What are texture coordinates and how are they related to 3D coordinates?
- What options do exist to generalise texture maps? For what other effects are they useful and what are the advantages and disadvantages of these techniques?
- How to program texture mapping in OpenGL?

CMT107 Visual Computing

VI.2 Curves

Xianfang Sun

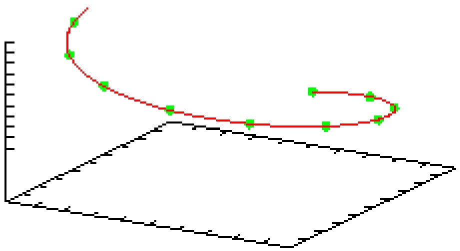
School of Computer Science & Informatics
Cardiff University

Overview

- Curve representations
  - Explicit representation
  - Implicit representation
- Parametric representation of curves
  - Piecewise polynomial curves (spline curves)
  - Bézier curves

Curves

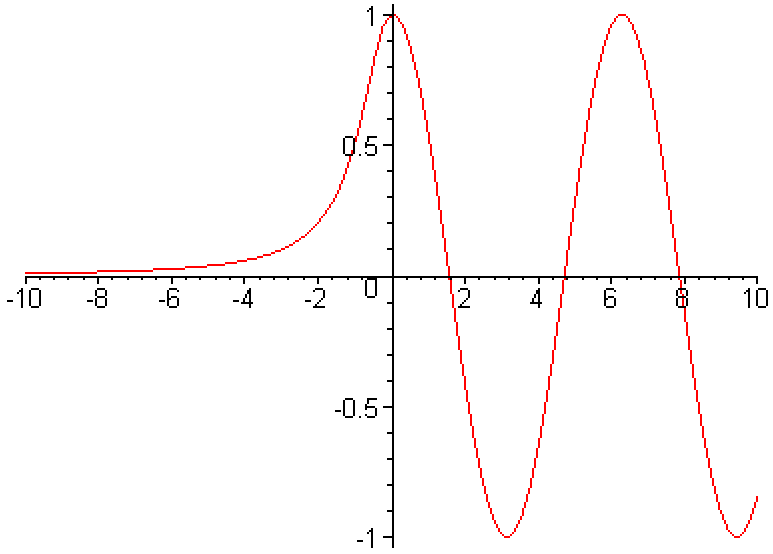
- A curve is a set of positions of a point moving with one degree of freedom



- Useful to describe shapes on a *higher level*
  - Not only straight lines or curved shapes approximated by short line segments
  - Simpler to create, edit and analyse
  - More accurate rendering and less storage (compared to linear approximation)

Explicit Representation

- *Explicit curve*: $y = f(x)$
  - Essentially a *function plot* over some interval $x \in [a, b]$



- Properties:
  - Simple to compute points and plot them
  - Simple to check whether a point lies on curve
  - Cannot represent closed or multi-valued curves:

    Only one y value for each x value (a function)

Implicit Representation

- Define curves *implicitly* as *solution of an equation system*
  - Straight line in 2D: $Ax + By + C = 0$
  - Circle of radius R in 2D: $x^2 + y^2 - R^2 = 0$
  - Conic section: $Ax^2 + 2Bxy + Cy^2 + Dx + Ey + F = 0$
  - Matrix/vector representation up to order two:

$$\boldsymbol{x}^T M \boldsymbol{x} + v^T \boldsymbol{x} + s = 0 \quad (\boldsymbol{x} = [x\ y]^T)$$

- In 3D, *two* equations are needed
  (*1 equation restricts 1 variable*, but there are 3 variables)
  - Straight line: $Ax + By + Cz + D = 0,$

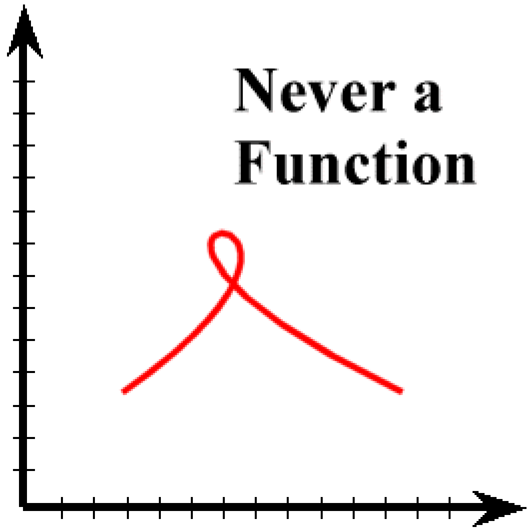
$$Ex + Fy + Gz + H = 0$$

  - A circle in x-y plane: $x^2 + y^2 = r^2,$

$$z = 0$$

Properties of Implicit Curves

- Mainly use polynomial or rational functions
- Coefficients determine geometric properties
- *Properties:*
  - Hard to render (have to solve non-linear equation system)
  - Simple to check whether a point lies on curve
  - Can represent closed or multi-valued curves



Parametric Curves

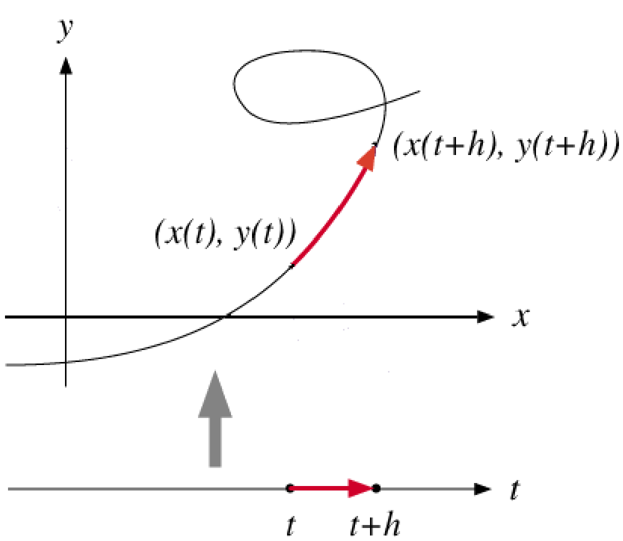
- Describe the position on the curve by a parameter $u \in R$

$$c(u) = \begin{pmatrix} x(u) \\ y(u) \\ z(u) \end{pmatrix}$$

  - $x(u)$, $y(u)$, $z(u)$ are usually polynomial or rational functions in u
  - $u \in [a, b]$, usually $u \in [0, 1]$
- Parameter function maps parameter to model coordinates
  - Parameter space: u (parameter domain)
  - Model space: x, y, z (Cartesian coordinates)

Properties of Parametric Curves

- *Properties*:
  - Simple to render (evaluate parameter function)
  - Hard to check whether a point lies on curve (must compute inverse mapping from (x, y, z) to u; involves solving non-linear equations)
  - Can represent closed or multi-valued curves

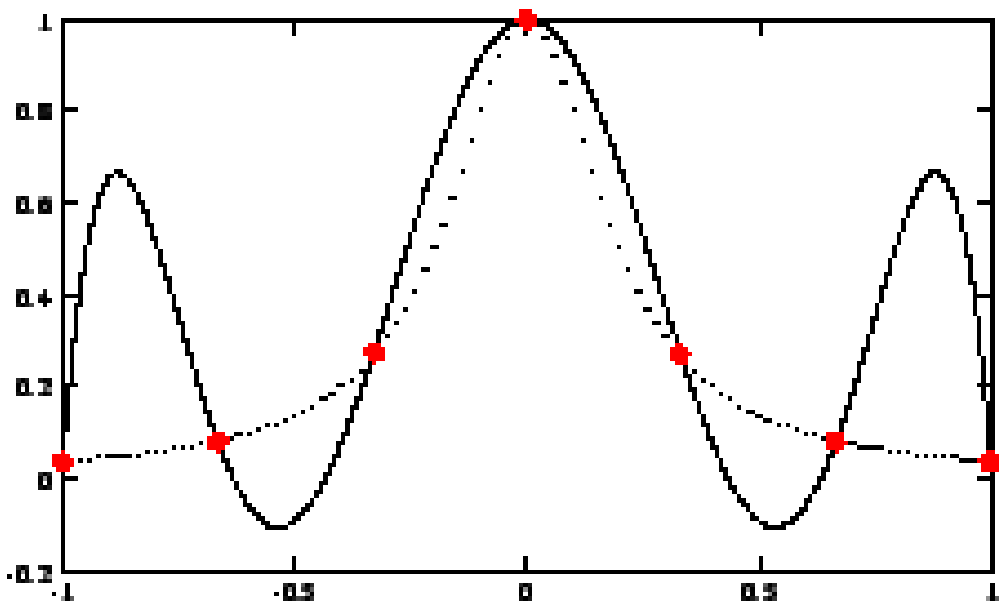


Parametric Polynomial Curves

- Describe coordinates by *polynomials*:

$$x(u) = \sum_{l=0}^{d} A_l u^l, \quad y(u) = \sum_{l=0}^{d} B_l u^l, \quad z(u) = \sum_{l=0}^{d} C_l u^l$$

- *Smooth* (infinitely differentiable)
- Higher order curves (say > 4) cause *numerical problems*
- Hard to control shape by *interpolation*



Bernstein Polynomials

- Bernstein *basis* polynomials

$$b_l^d(u) = \binom{d}{l} u^l (1-u)^{d-l}, \; l = 0, 1, \cdots, d.$$

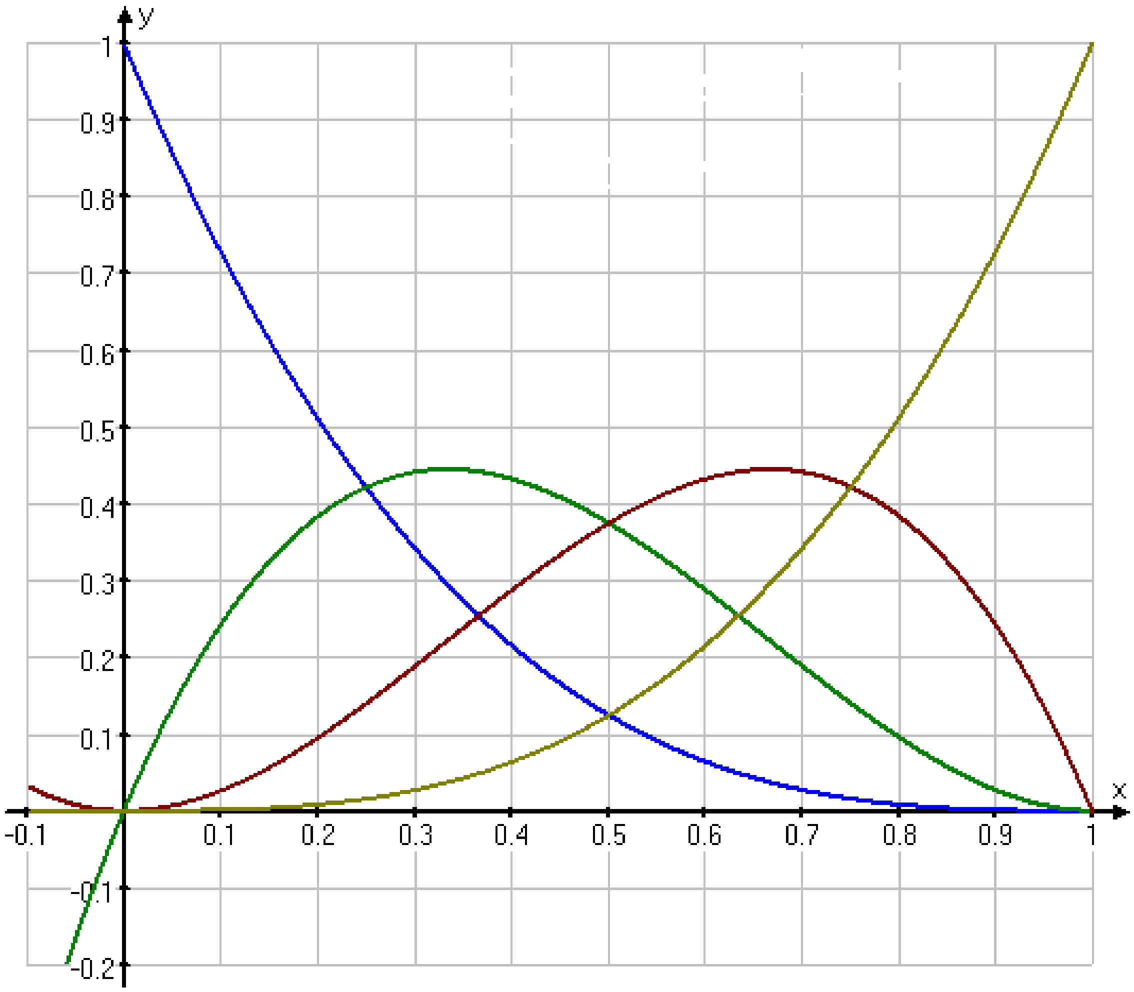
- $\binom{d}{l} = \dfrac{d!}{l!(d-l)!}$ is binomial coefficient.
- Property: $\sum_{l=0}^{d} b_l^d(u) = 1 \text{ for } u \in [0, 1]$

- A Bernstein polynomial is a linear combination of Bernstain basis polynomials

$$B(u) = \sum_{l=0}^{d} \beta_l b_l^d(u), \; u \in [0, 1].$$

Cubic Bernstein Basis Polynomials

- There are 4 cubic Bernstein basis polynomials

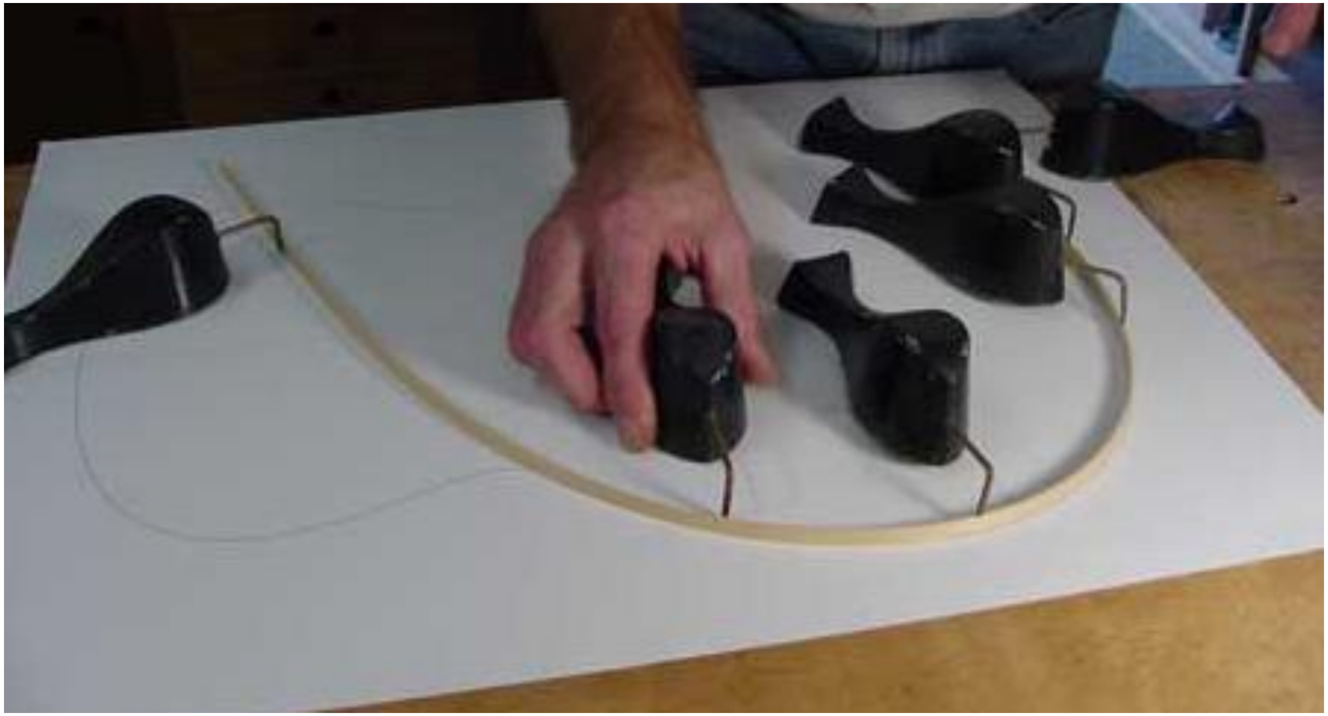


Piecewise Polynomial Curves

- Cut curve into *segments* and represent each segment as *polynomial* curve
- Can use *low-order polynomial* curves, e.g. cubic (order 3)
- But how to guarantee *smoothness at the joints*?
  - Continuity problem

Spline Curves

- In general, piecewise polynomial curves are called splines
  - Motivated by loftsman's spline
    - Long narrow strip of wood or plastic
    - Shaped by lead weights (called ducks)
  - Gives curves that are smooth or fair

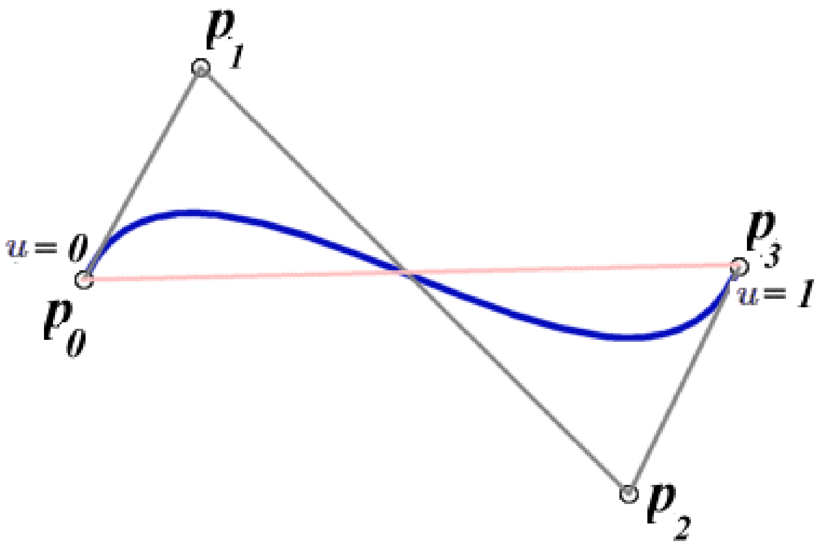


Bézier Curves

- Represent a polynomial segment as

$$\boldsymbol{Q}(\boldsymbol{u}) = \sum_{l=0}^{d} \boldsymbol{p}_l \boldsymbol{b}_l^d(\boldsymbol{u}),\ \boldsymbol{u} \in [0, 1]$$

$$\boldsymbol{Q}(\boldsymbol{u}) = \sum_{l=0}^{d} \boldsymbol{p}_l \binom{\boldsymbol{d}}{\boldsymbol{l}} \boldsymbol{u}^l (1-\boldsymbol{u})^{d-l},\ \boldsymbol{u} \in [0, 1].$$



  - Control points $p_l \in R^3$ or R^2 determine segment's shape
  - $\boldsymbol{b}_l^d(\boldsymbol{u})$: l^{th} *Bernstain basis polynomial* of degree d.

- Cubic Bézier curve ($d = 3$) has four control points
  - Note that $\sum_{l=0}^{d} \boldsymbol{b}_l^d(\boldsymbol{u}) = 1$ for $\boldsymbol{u} \in [0, 1]$

    ➡ *Convex combination* of control points

Properties of Bézier Curves

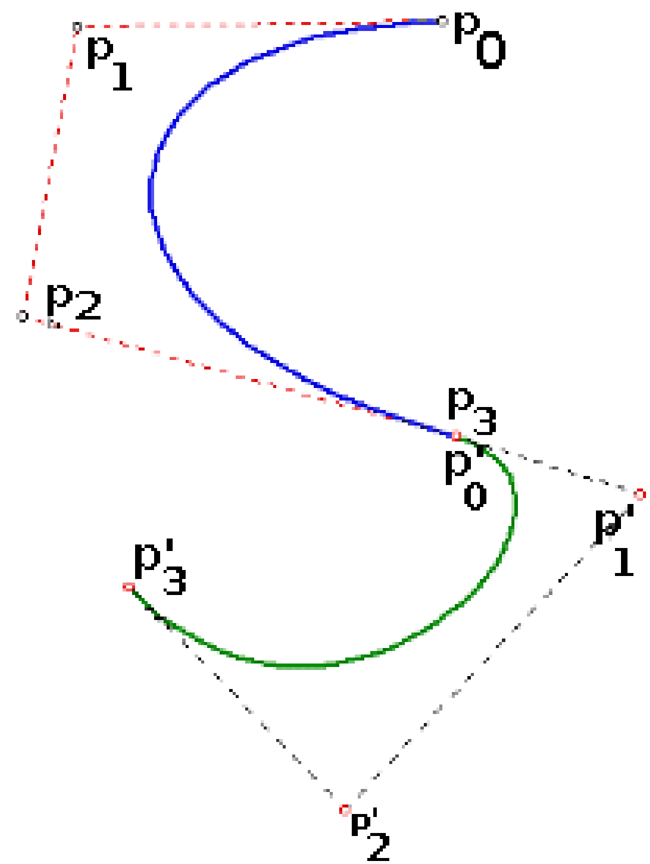
- *Convex hull*:
  - curve lies inside the convex hull of its control points
- *Endpoint interpolation:*

$$Q(0) = p_0$$
$$Q(1) = p_d$$

- *Tangents*

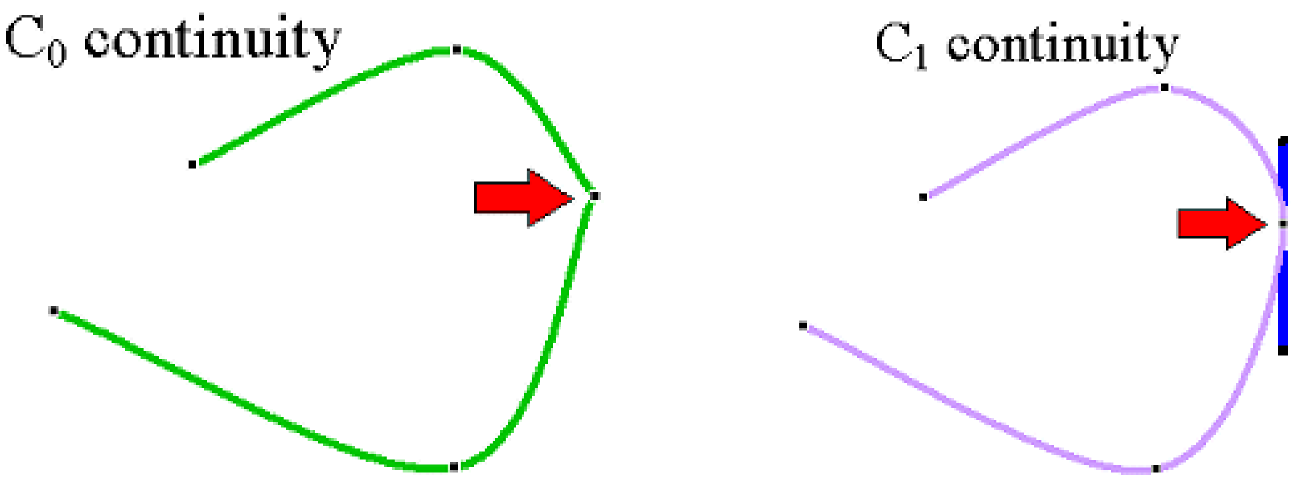
$$Q'(0) = d(p_1 - p_0)$$
$$Q'(1) = d(p_d - p_{d-1})$$

- *Symmetry*
  - $Q(u)$ defined by $p_0, \ldots, p_d$ is equal to $Q(1 - u)$ defined by $p_d, \ldots, p_0$



Smooth Bézier Curves

- *Smooth joint* between two Bézier curves of order d with control points $\{p_0, \ldots, p_d\}$, $\{p'_0, \ldots, p'_d\}$ respectively
  - C_0: same end-control-points at joints: $p_d = p'_0$ (due to end-point interpolation)
  - C_1: control points p_{d-1}, $p_d = p'_0$, p'_1 must be collinear (due to tangent property)



➡ Continuity conditions create restrictions on control points

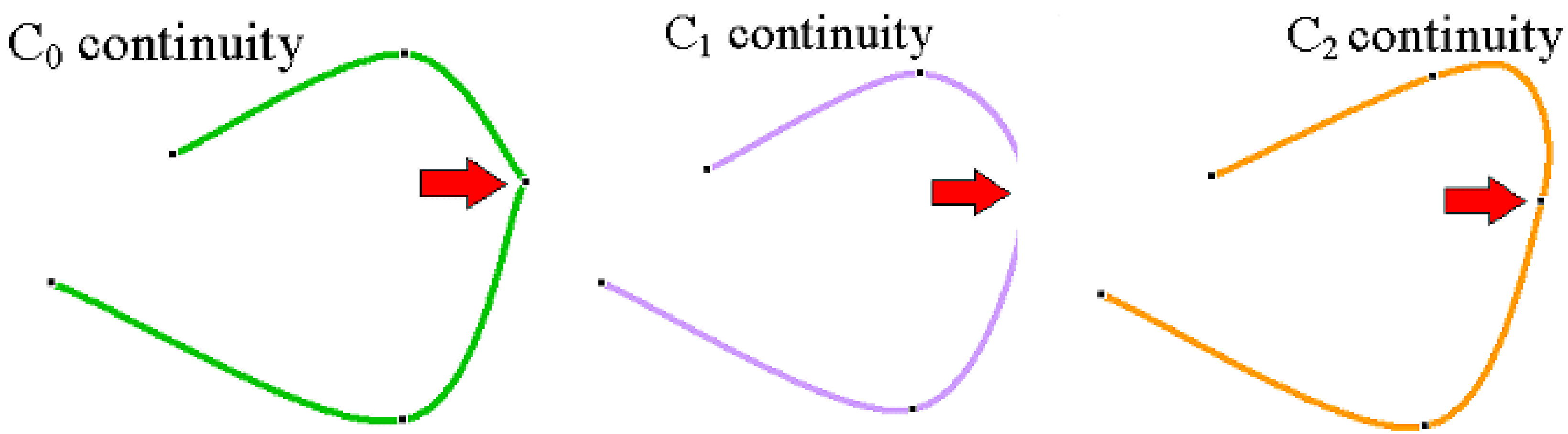
Parametric/Geometric Continuity

- Parametric continuity:
  - C^0: curves are joined
  - C^1: first derivatives are equal at the joint points
  - C^2: first *and* second derivatives are equal

    …
  - C^n: first *through* n^{th} derivatives are equal
- Geometric continuity:
  - G^0: The curves touch at the joint points
  - G^1: The curves also share a common tangent direction at the joint points (first derivatives are *proportional*)
  - G^2: The curves also share a common centre of curvature at the joint points (first and second derivatives are *proportional*)

Smoothness / Continuity

- Curve should be *smooth* to some order at joints
- Different types of *continuity at joints*
- *Geometric continuity*: from the geometric viewpoint
- *Parametric continuity*: for parametric curves



- Parametric continuity of order *n* implies geometric continuity of order n, but not vice versa.

Summary

- What is the implicit and explicit representation of a curve? What are the advantages and disadvantages of these representations?
- What are piecewise parametric polynomial curves (splines)? What is the advantage of this representation? What is the main problem?
- What are Bézier Curves and how are they defined? What properties do they have?
- What is the major problem when using piecewise polynomial curves? What conditions do the control points of a Bézier Curve have to fulfil in order to get C_0/C_1 continuous curves?