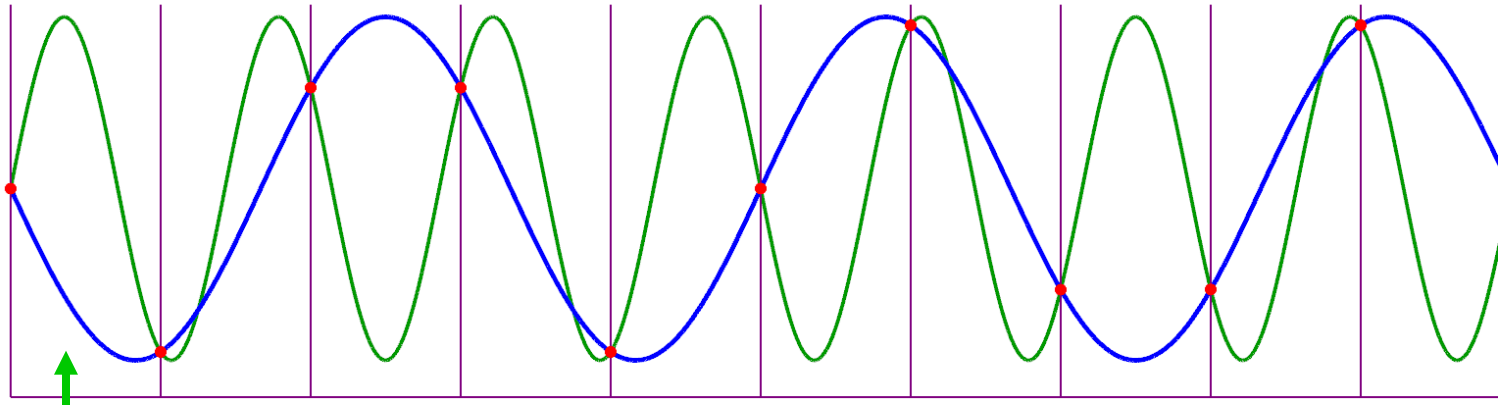
The Sampling Theorem

To avoid aliasing, the sampling rate must exceed twice the highest frequency of the sampled signal

(see Wikipedia: Nyquist-Shannon sampling theorem)



press

a seven-cycle signal (green) is sampled ten times, producing a three-cycle alias (7 is 2 greater than 5, reflects to 3)

Audio Demo

Undersampled frequencies reflect around the Nyquist frequency

To remove high frequencies: low-pass filter

(convolve with a low-pass kernel)

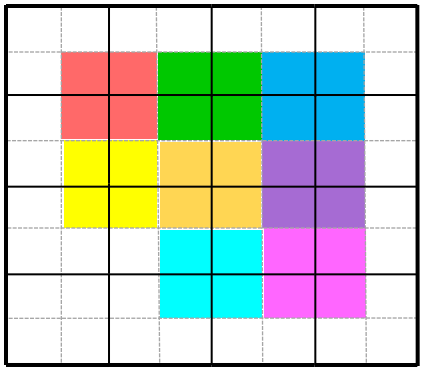
A Pixel Is *Not* A Little Square

(Alvy Ray Smith)

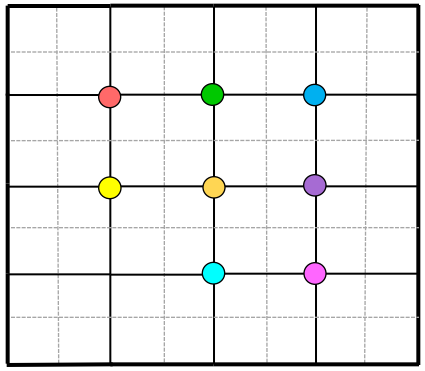
A pixel is a *discrete* sample that represents a region of a *continuous* image

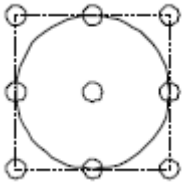
The region may be a circle or ellipse, but just about never a square (or rectangle)

NOT THIS

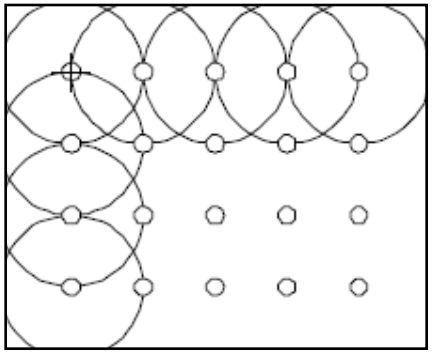


THIS





a pixel footprint



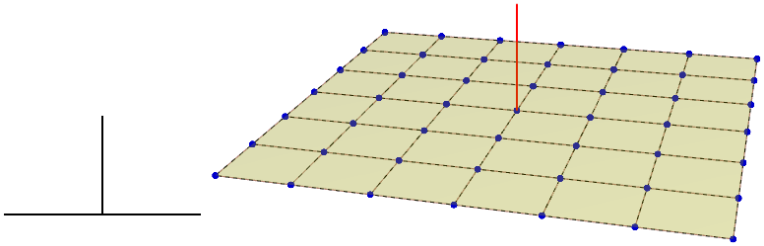
pixel footprints overlap

Filtering the Pixel Footprint

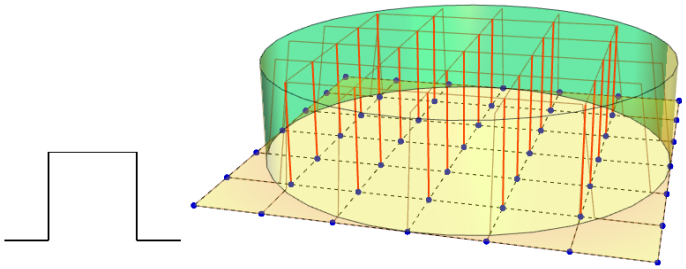
The pixel is a single measure (i.e., a *sample*) of a continuous image (*i.e.,* signal), assuming the upper frequency of the image is below half that of the pixels

If this requirement is not met, aliasing (moirés, scintillation, stair-casing) can occur

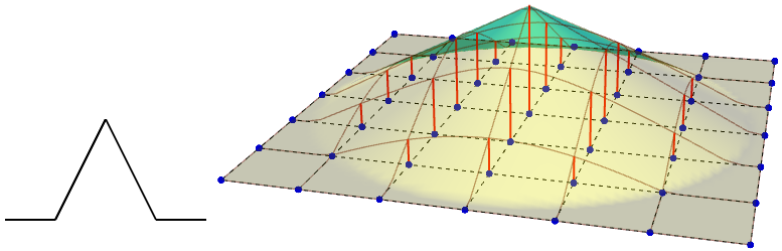
(press any one!)



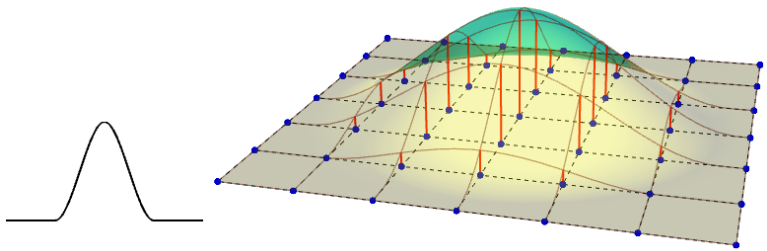
Point



Box



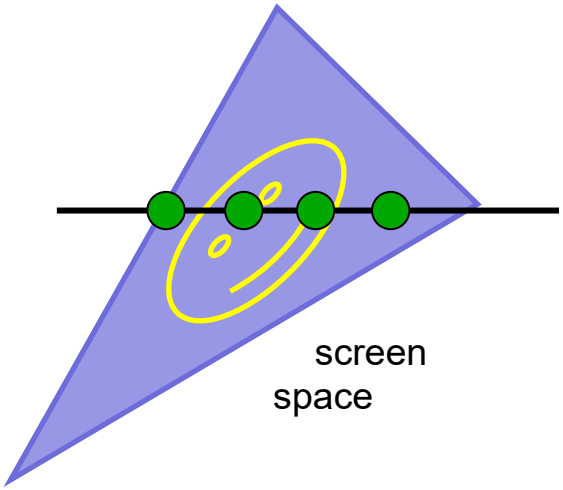
Triangle



Cubic

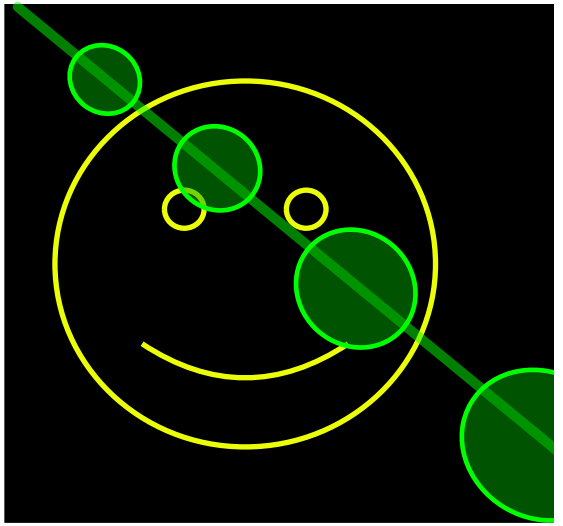
Pixel Footprints

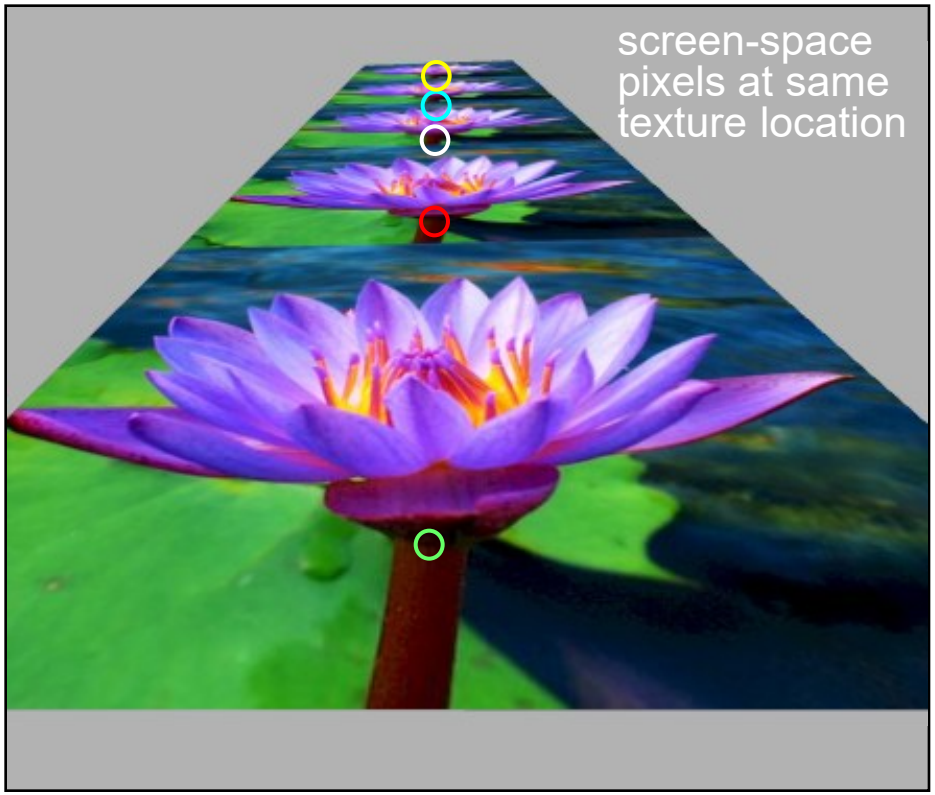
in Texture Space

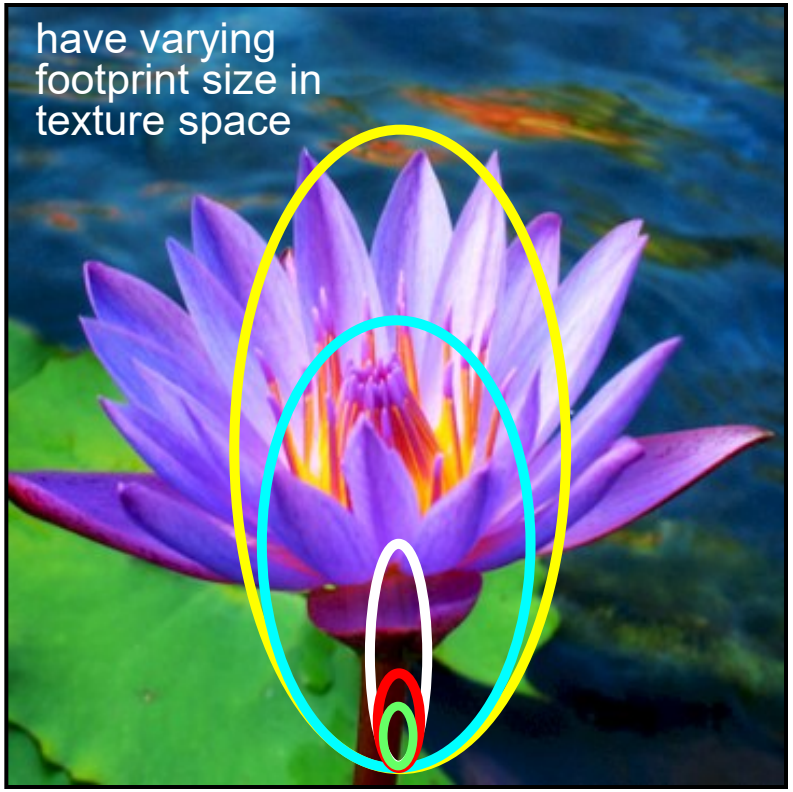


screen
space

texture
space







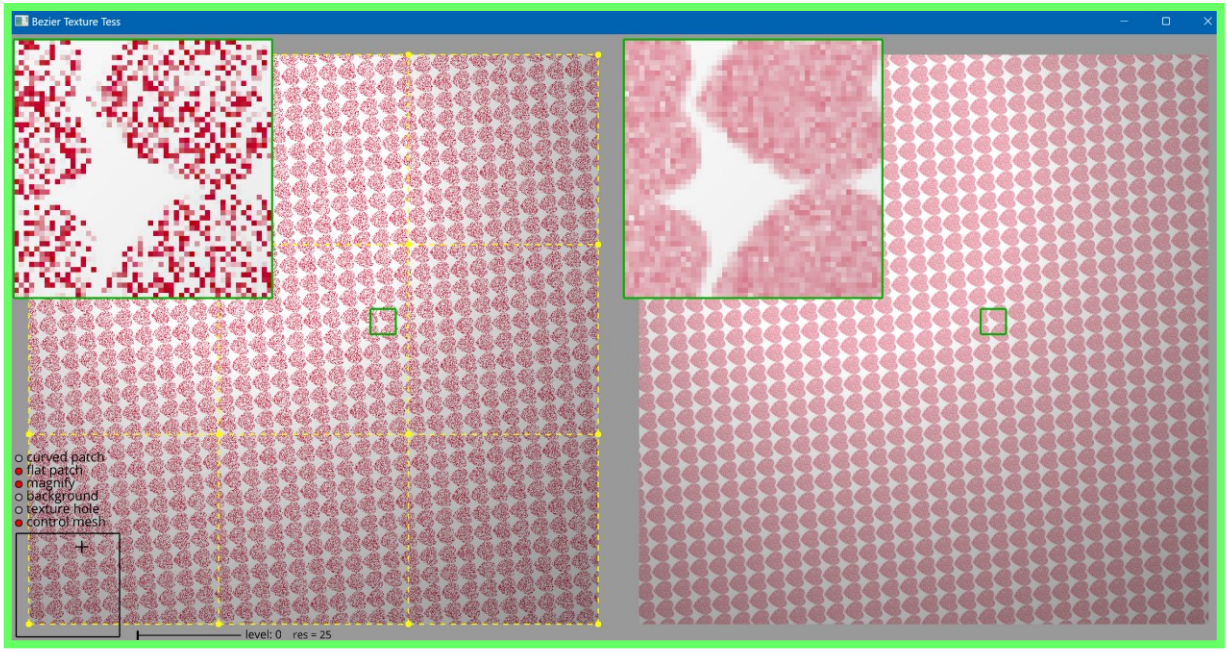
Mipmap Demo

Aliasing occurs if the texture contains a frequency that exceeds the *Nyquist* frequency (half the sampling rate)

OpenGL tracks the change in *uv* from pixel to pixel

when calling *texture* from the pixel shader

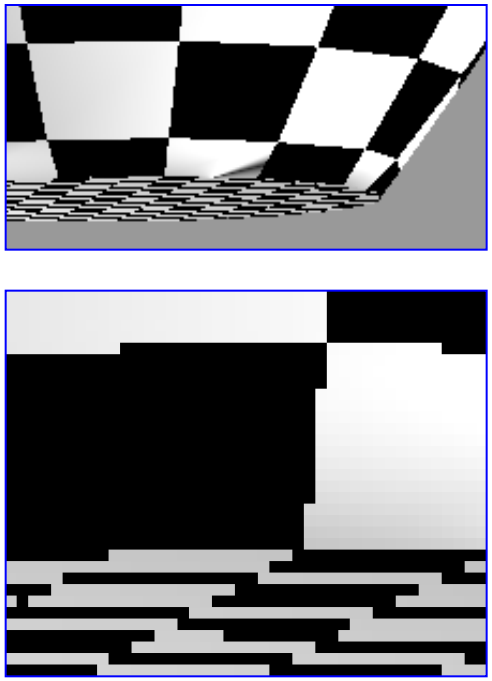
OpenGL knows the sampling rate across the texture image and chooses the appropriate mipmap level



PRESS FOR DEMO

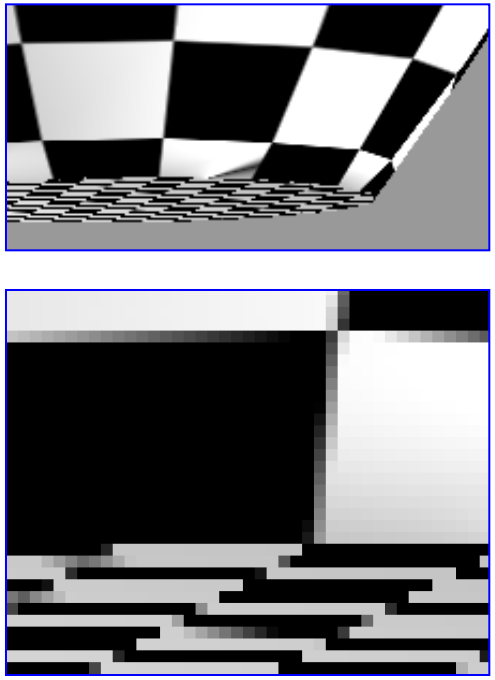
Texture Anti-aliasing

Aliased



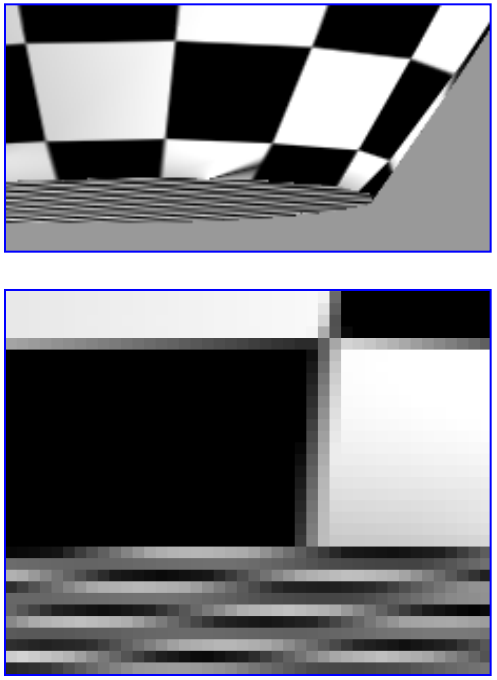
```
glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_FILTER, GL_NEAREST);
```

Bilinearly Interpolated

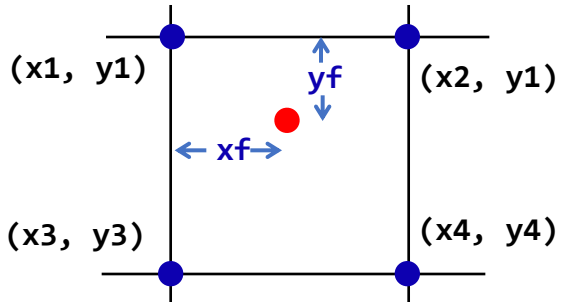


```
glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_FILTER, GL_LINEAR);
```

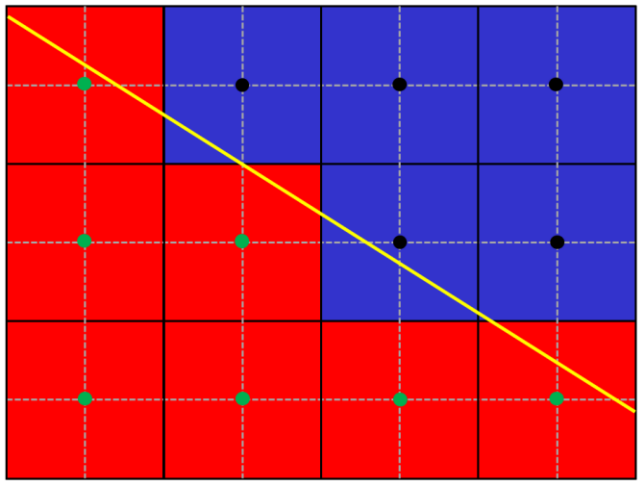
Mip-mapped

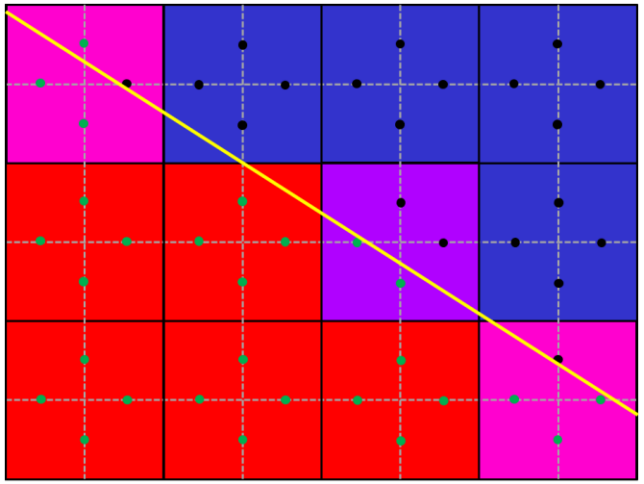


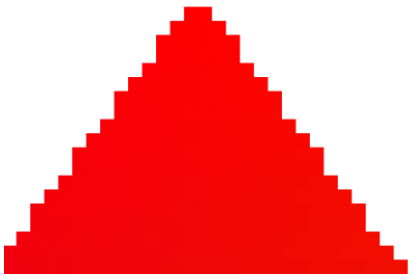
```
glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_FILTER,
GL_LINEAR_MIPMAP_LINEAR
```



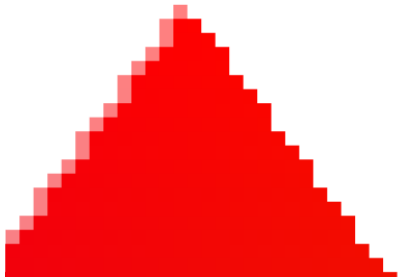
Multi-sampling for Silhouette Edges



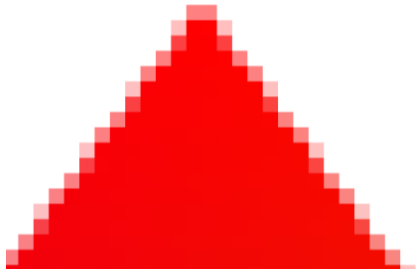




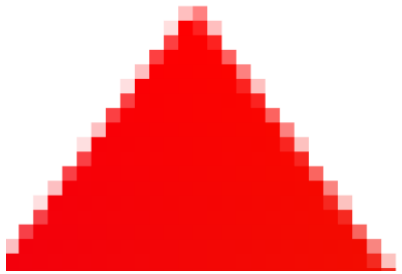
No Sub-Pixels



2x2 Sub-Pixels



4x4 Sub-Pixels

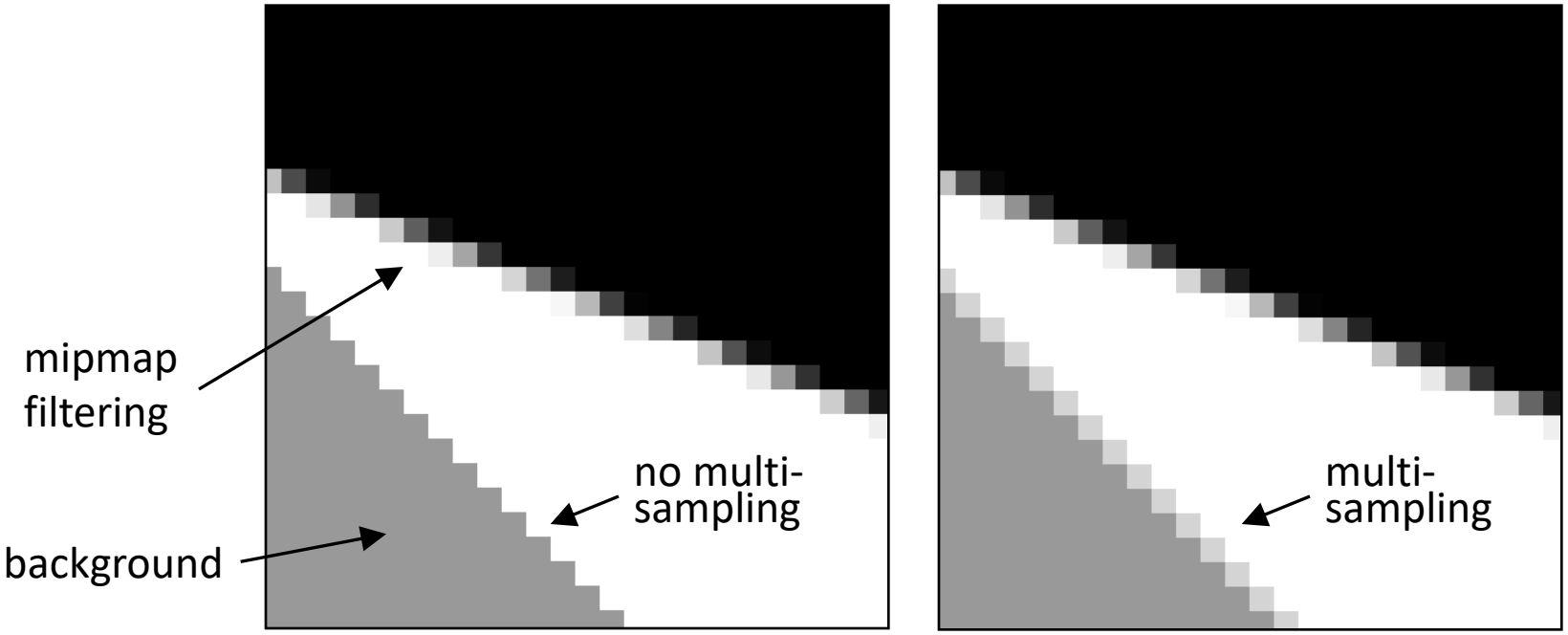


8x8 Sub-Pixels

Implementation

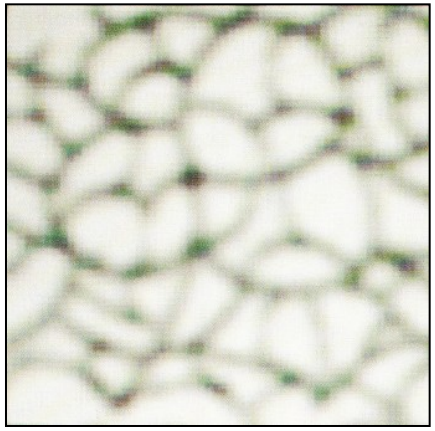
To reduce the effects of staircasing, the call to glfwInit includes:

```
glfwWindowHint(GLFW_SAMPLES, 4); // anti-alias edges
```

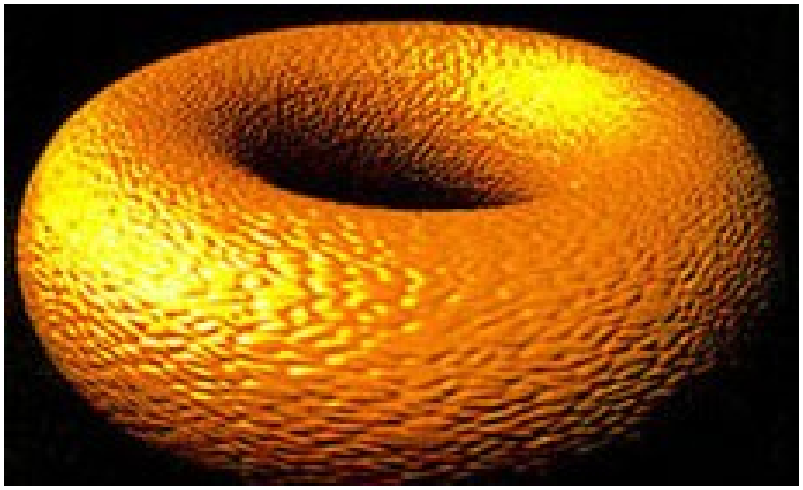


Bumps

Jim Blinn addressed this class in 2012



Hand-drawn bump texture



Tektronix 4010

ACKNOWLEDGEMENTS

The author would like to thank the following people whose assistance has proven invaluable in the production of this thesis.

Ed Catmull, Bui-Tuong Phong, and Frank Crow upon whose work much of this work is directly based.

Jim Kajiya, Lance Williams, and Larry Evans for first leading me to believe that a scan line based algorithm for patches was even possible.

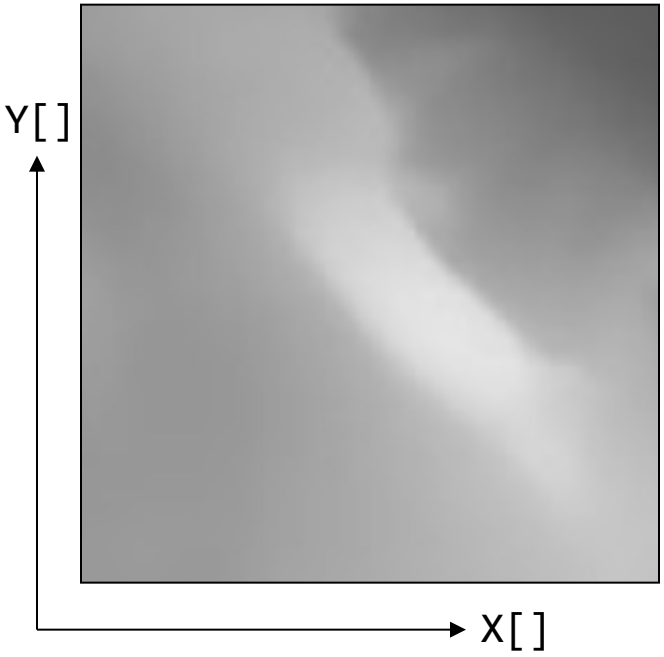
Gary Demos for keeping me honest.

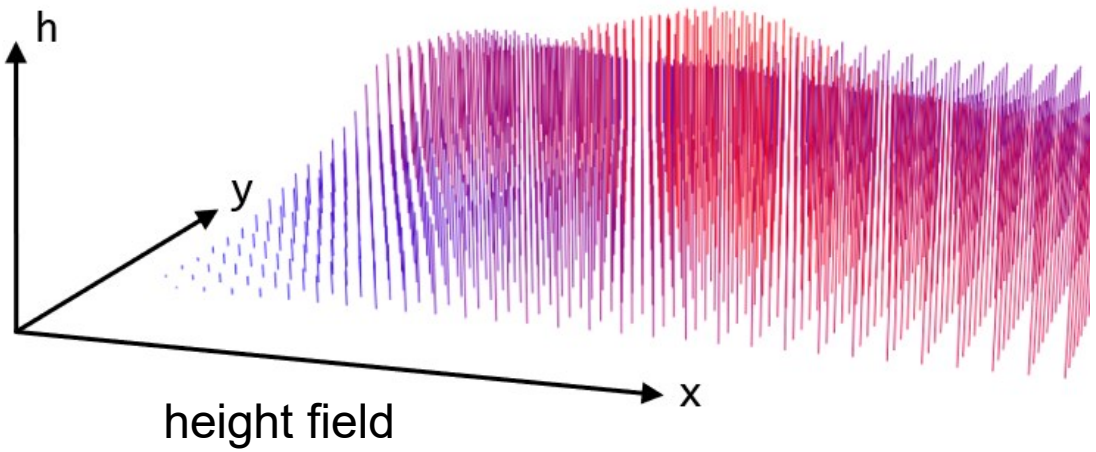
The Directors and Staff of The New York Institute of Technology and Information International Incorporated for employment, encouragement, and use of facilities.

Jules Bloomenthal, Allan Cima, Mike Wilson, and David DiFrancisco for use of their photographic equipment, and especially to Jules for allowing me to steal so much of his computer time.

Finally, to my advisor, Martin Newell, for allowing me to take over his office for the past three years.

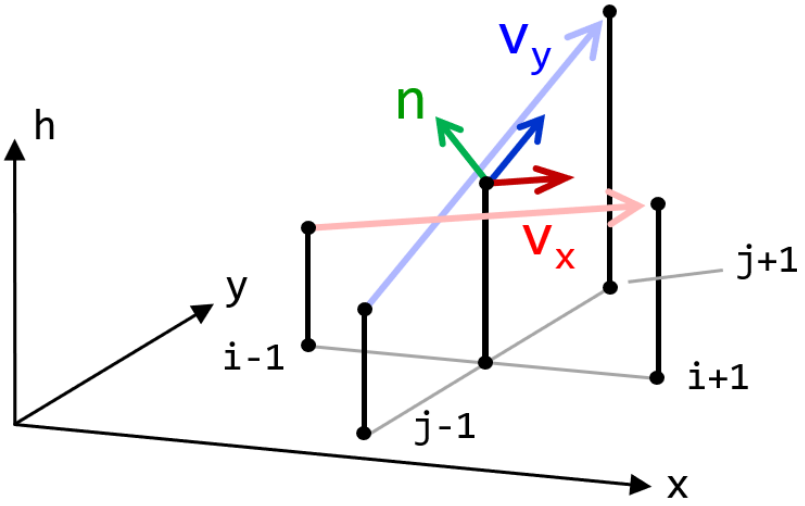
depth map of
Yosemite Valley

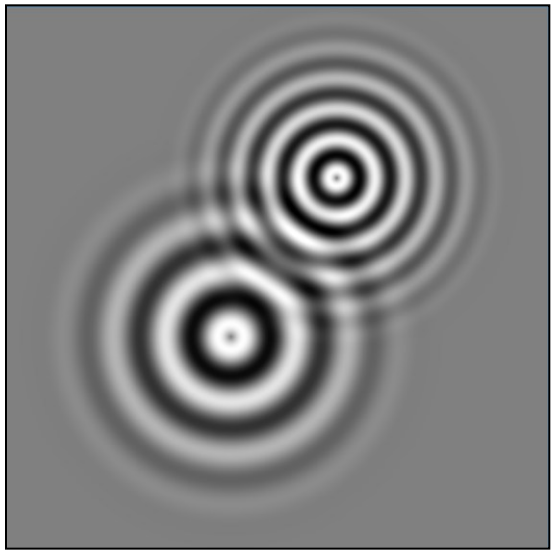




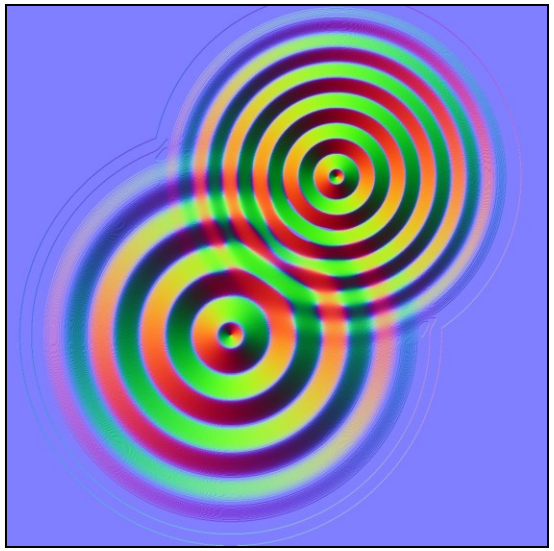
height field

computing
normal n

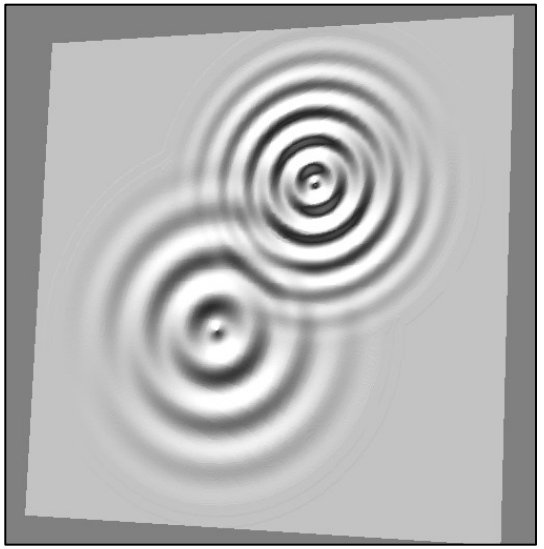




Depth Map



Normal/Bump Map

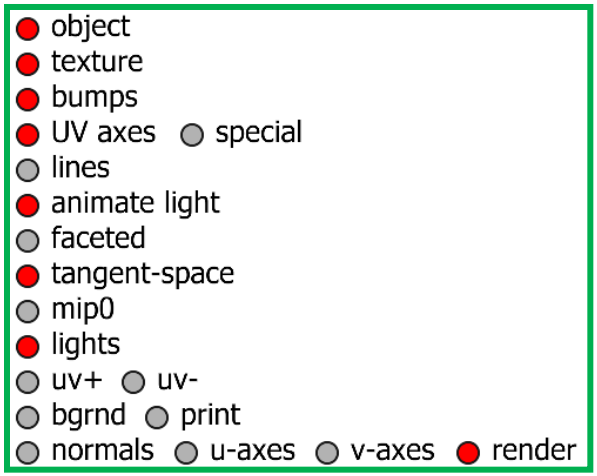


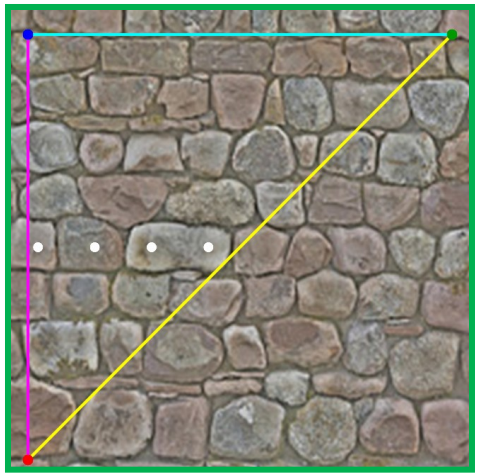
Bump-mapped Image

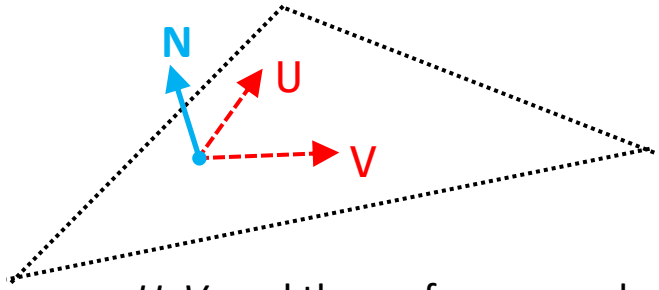
Applying the bump map to a flat object such as above is straightforward.

For a curved, irregular surface, however, the vector given by the bump map must be oriented locally.

This is a bit complicated, as described in the next few slides.







U, *V*, and the surface normal constitute a reference frame

For a curved surface, however, the vector given by the bump map must be oriented locally.

This requires each vertex have an associated *reference frame* consisting of the surface normal and two bi-tangents. The bi-tangents are in the plane of the triangle and thus are at right angles with the surface normal. They are not necessarily at right angles to each other.

The bi-tangents must be stored in the vertex buffer along with the surface normal, and are sent to the vertex shader.

The previous section applies to a single triangle; the reference frame *UVN* is constant across the triangle. For an arbitrary mesh, the local *UV* coordinate system at a vertex can be set by averaging the *U* and *V* axes computed for each triangle surrounding the vertex. This is calculated by the application, not the GPU.

The *U* and *V* axes for each vertex are stored in the vertex buffer; during rendering they are transformed by the vertex shader and interpolated by the rasterizer. At each pixel, the interpolated *U*, *V*, and *N* (normal) vectors produce a local *UVN* reference frame. Section 15.6 (Bump-Mapped Mesh) provides code.

As there is now a *UVN* frame per vertex, a frame can be interpolated across a triangle. Thus, the *UVN* frame can change throughout a mesh.

Advanced Shading

Cook-Torrance Model

Distributed Ray-Tracing

Radiosity

The Rendering Equation

Shadows

Sub-Surface Scattering

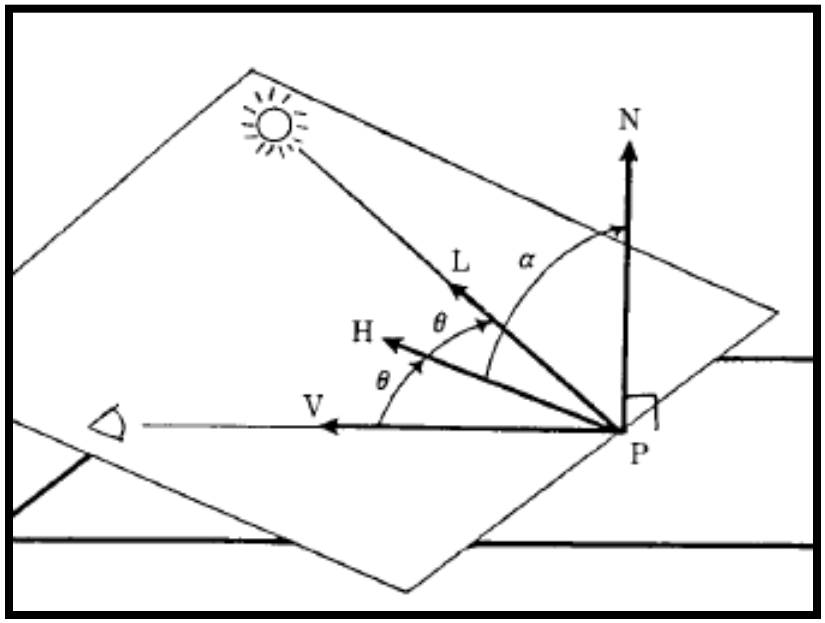
Solid Texture

Texture Generation

Artistic Shading

Cook-Torrance Shading Parameters

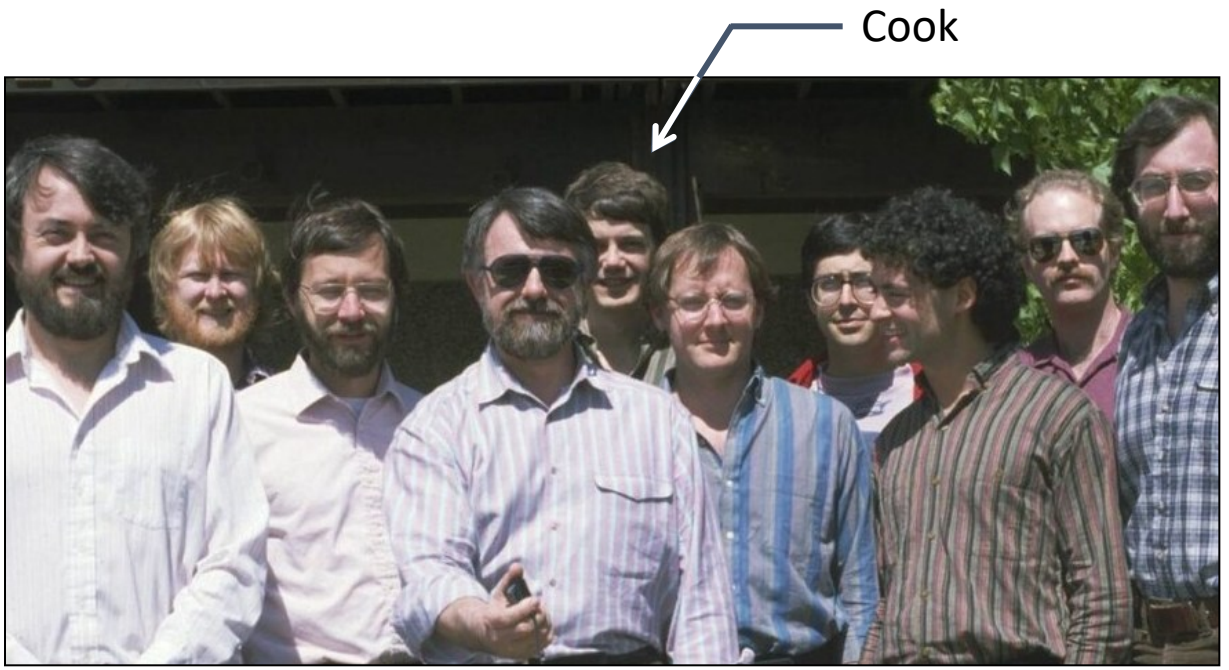
a list of variables considered in determining reflectivity at point P



A Reflectance Model for Computer Graphics

Cook & Torrance SIGGRAPH 1982

α	Angle between N and H
θ	Angle between L and H or V and H
λ	Wavelength
D	Facet slope distribution function
d	Fraction of reflectance that is diffuse
$d\omega_i$	Solid angle of a beam of incident light
E_i	Energy of the incident light
F	Reflectance of a perfectly smooth surface
f	Unblocked fraction of the hemisphere
G	Geometrical attenuation factor
$\mathbf{H}$	Unit angular bisector of V and L
I_i	Average intensity of the incident light
I_{in}	Intensity of the incident ambient light
I_r	Intensity of the reflected light
I_{rs}	Intensity of the reflected ambient light
k	Extinction coefficient
$\mathbf{L}$	Unit vector in the direction of a light
m	Root mean square slope of facets
$\mathbf{N}$	Unit surface normal
n	Index of refraction
R_a	Ambient reflectance
R	Total bidirectional reflectance
R_d	Diffuse bidirectional reflectance
R_s	Specular bidirectional reflectance
s	Fraction of reflectance that is specular
$\mathbf{V}$	Unit vector in direction of the viewer
w	Relative weight of a facet slope

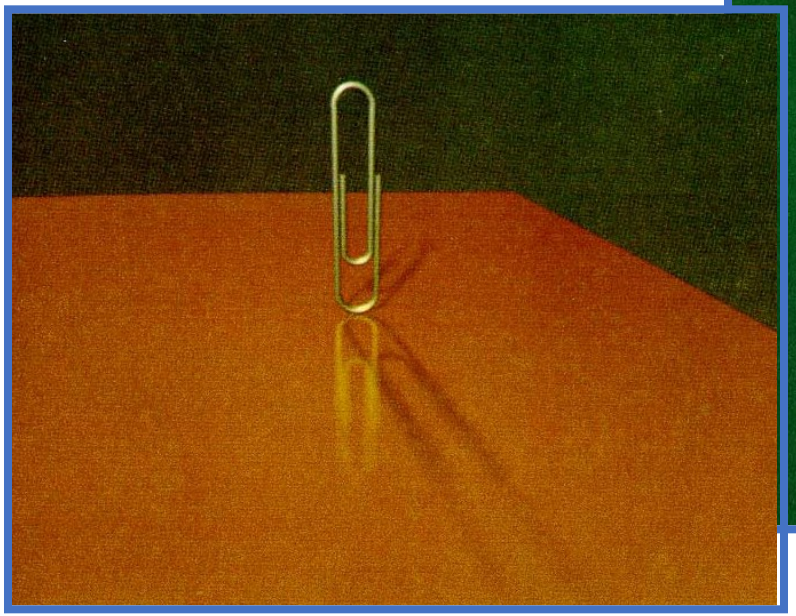


subtle differences and range of surface properties simulated by the Cook-Torrance model

Distributed Ray-Tracing

Distributed Ray Tracing is a numerical solution that supports:

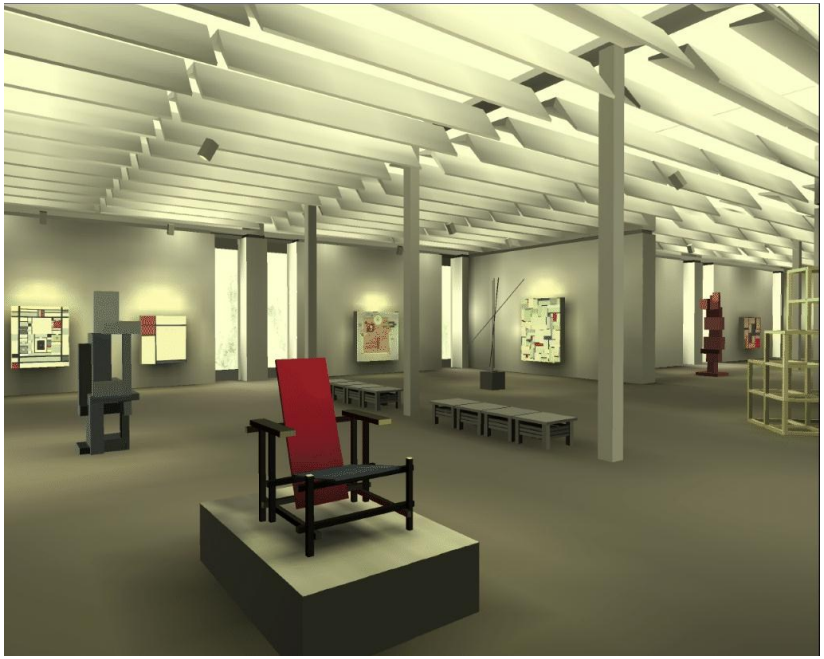
gloss
translucency
penumbra
depth of field
motion blur



Cook, Porter and Carpenter 1984

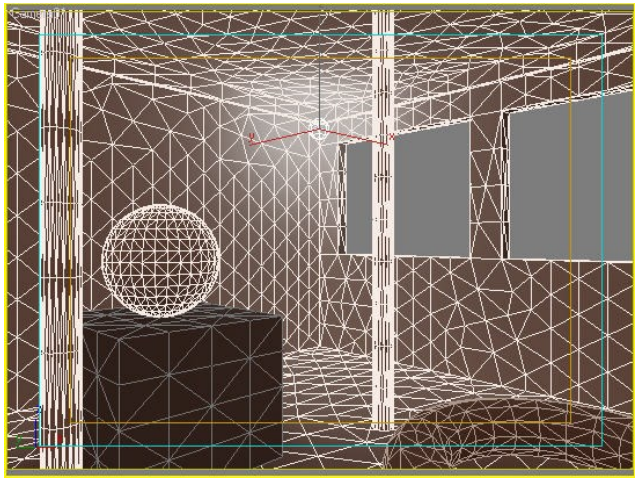
Radiosity

heat transfer: total radiation leaving a surface
3D computer graphics: **a global illumination method**

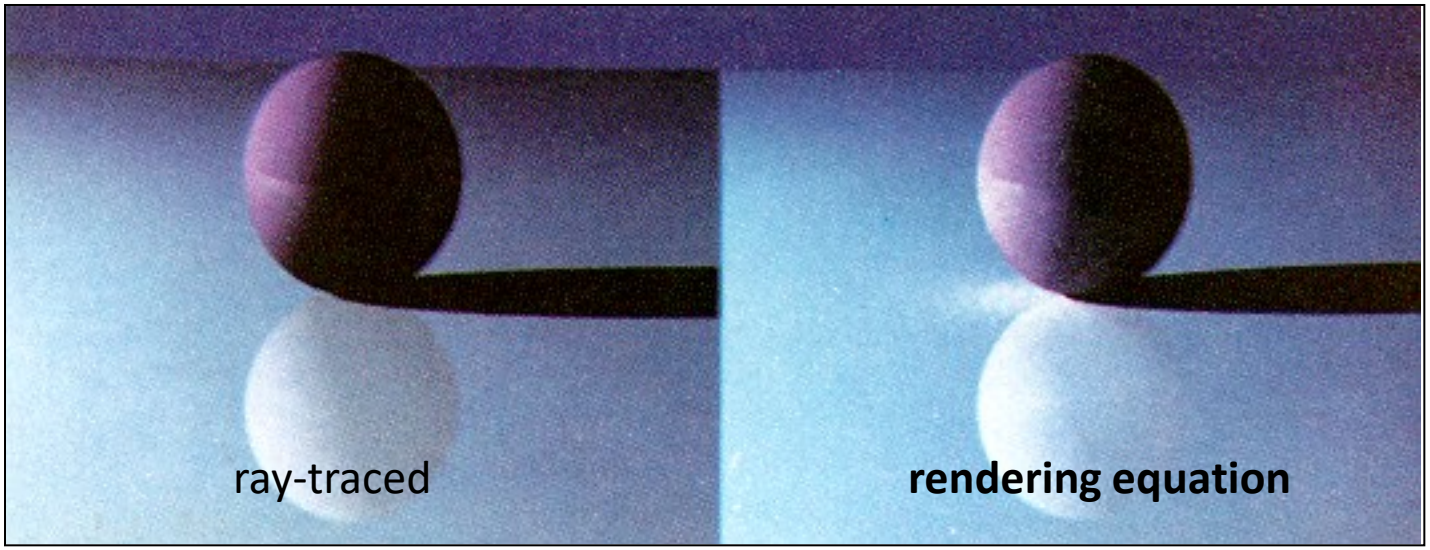


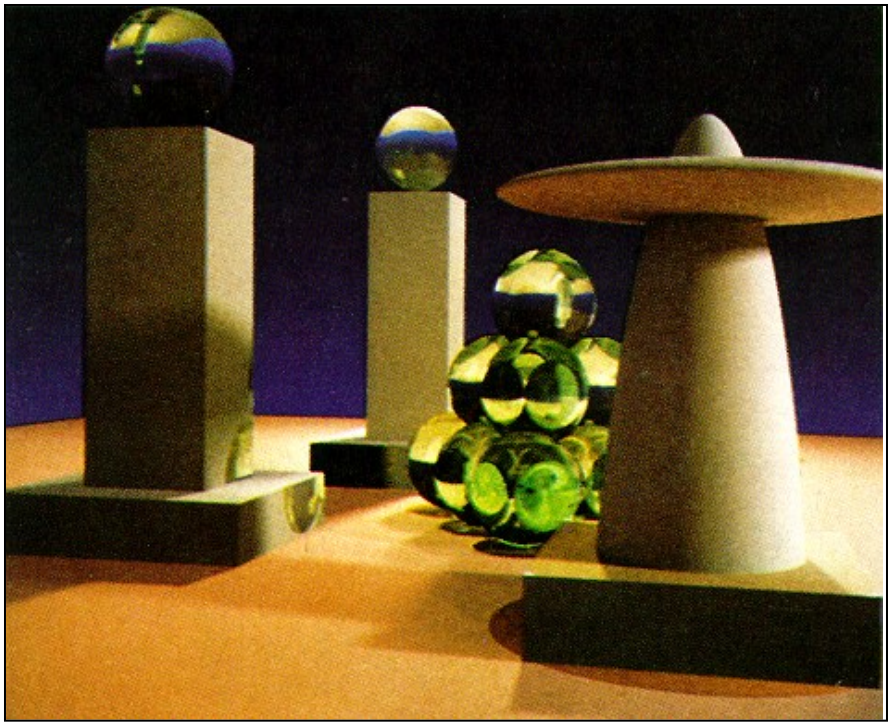
Cornell University

meshing used for radiosity calculations



www.arch.ttu.edu





right image shows improved table/ball inter-reflections

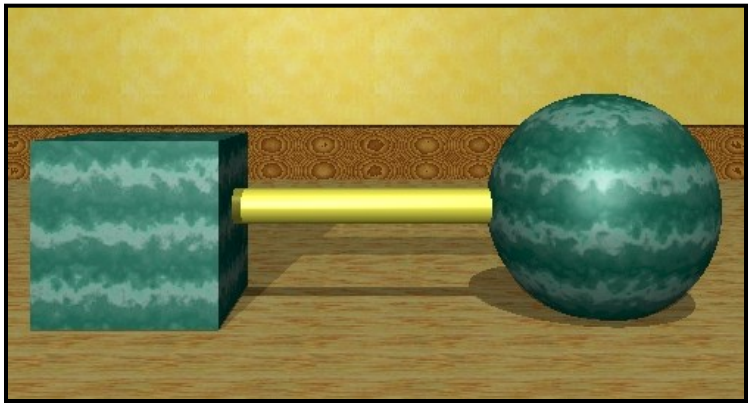
Kajiya, ‘86

Shadows

Casting Curved Shadows on Curved Surfaces
Williams, SIGGRAPH 1978

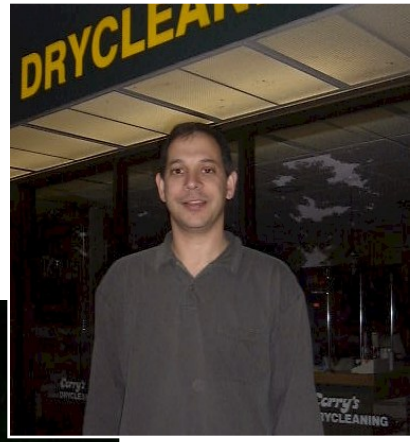
3D Texture

Computer Graphics
Laboratory ETH Zurich



Peachey, SIGGRAPH 1985

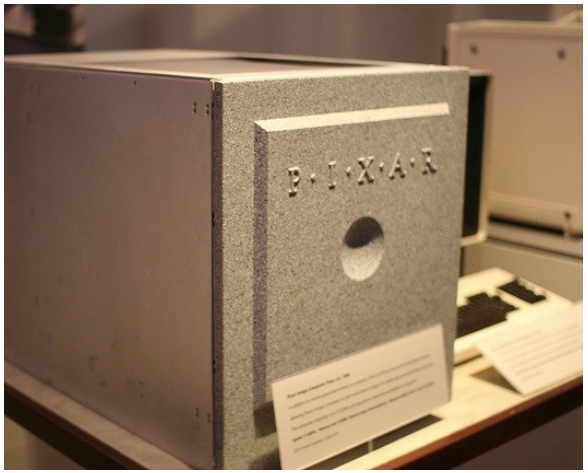
Perlin, SIGGRAPH 1985



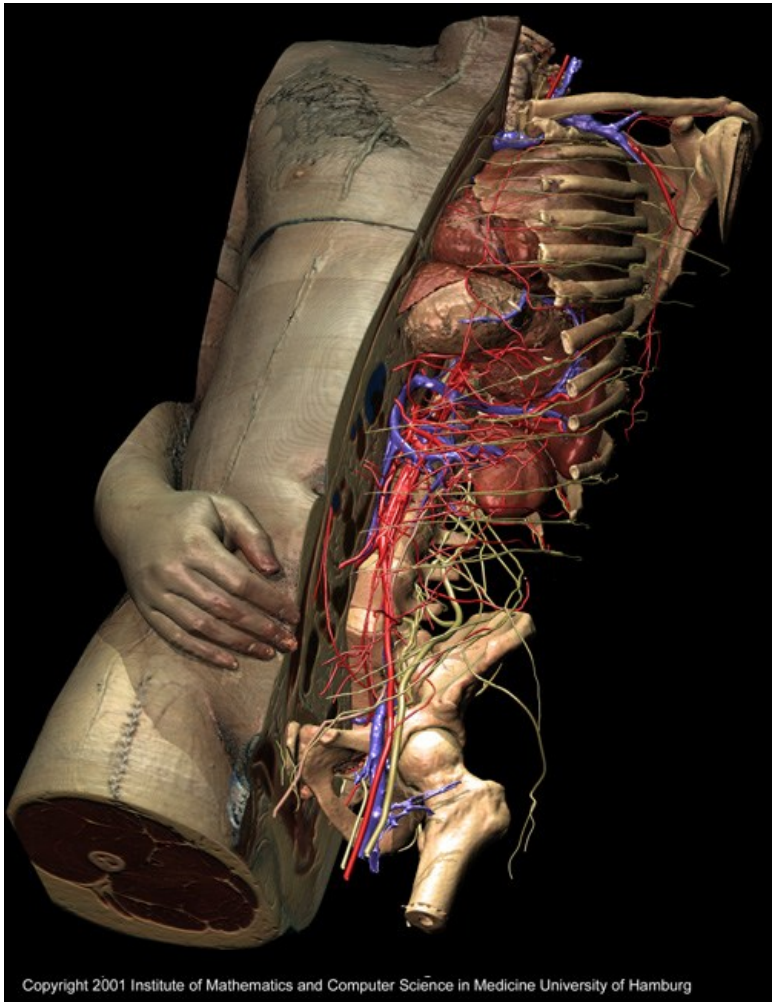
Subsurface Scattering

Henrik Jensen 2002

Stanford University
2001



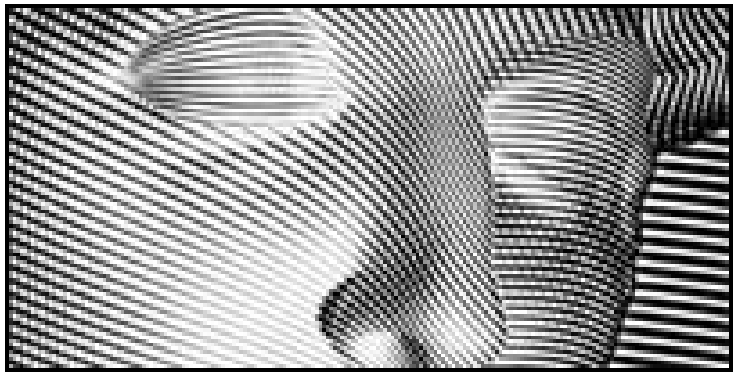
For CAT, MRI or PET scans, multiple slices are transformed to the screen and rendered with semi-transparency as a diagnostic or surgical aid. By detecting differences in density, surfaces can be estimated and shaded for added clarity. The Pixar Image Computer was built for this purpose.



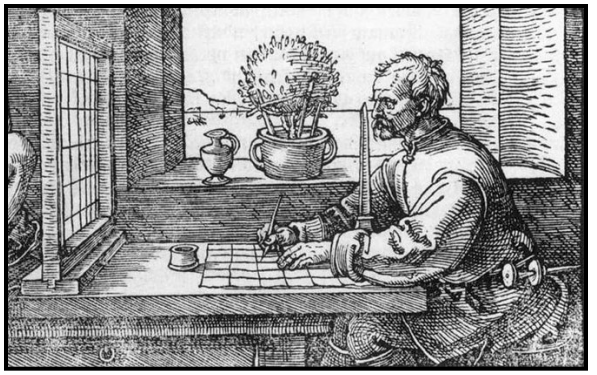
Artistic Shading

http://www.red3d.com/cwr/npr/

Barbara Meier

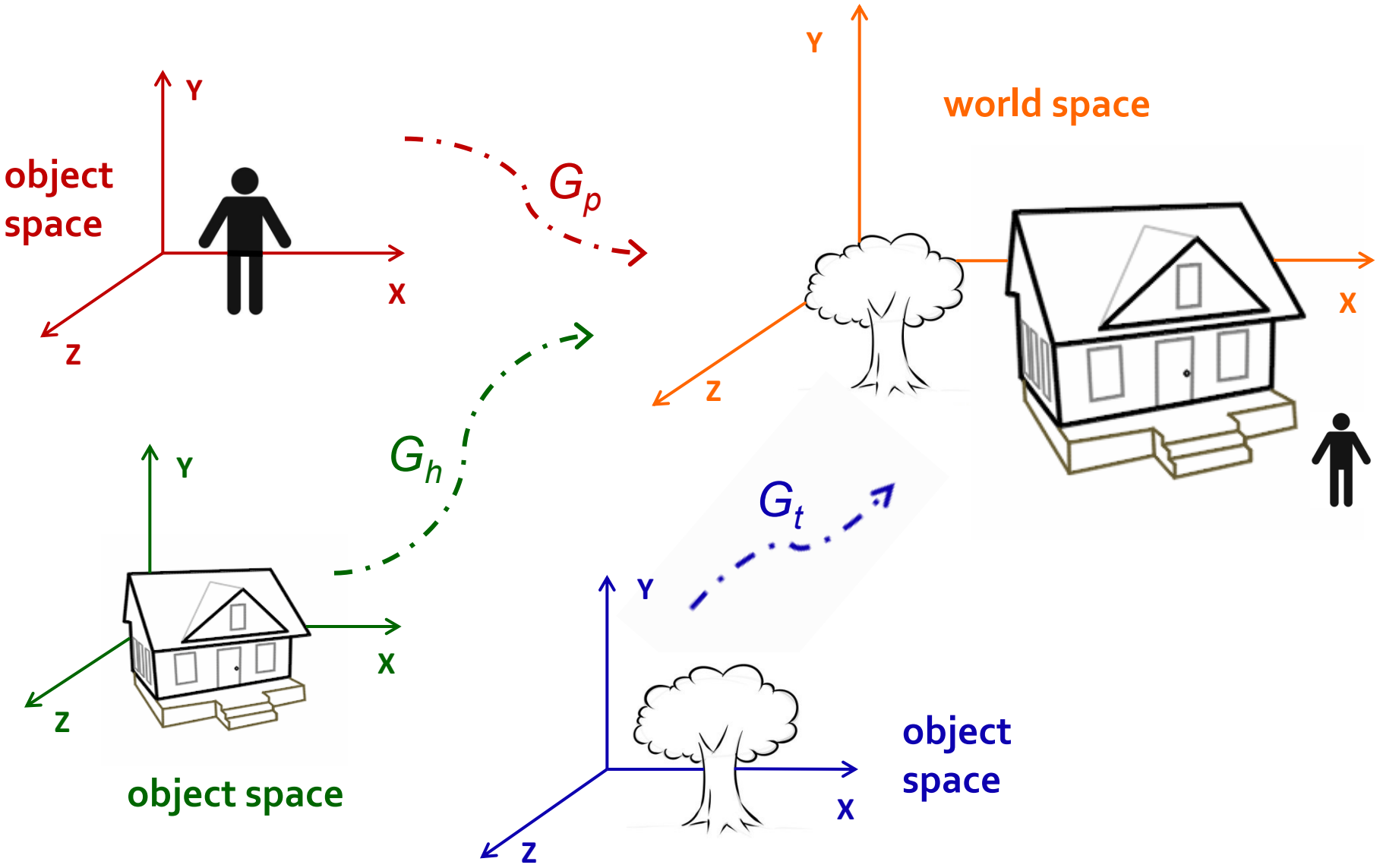


Victor Ostromoukhov

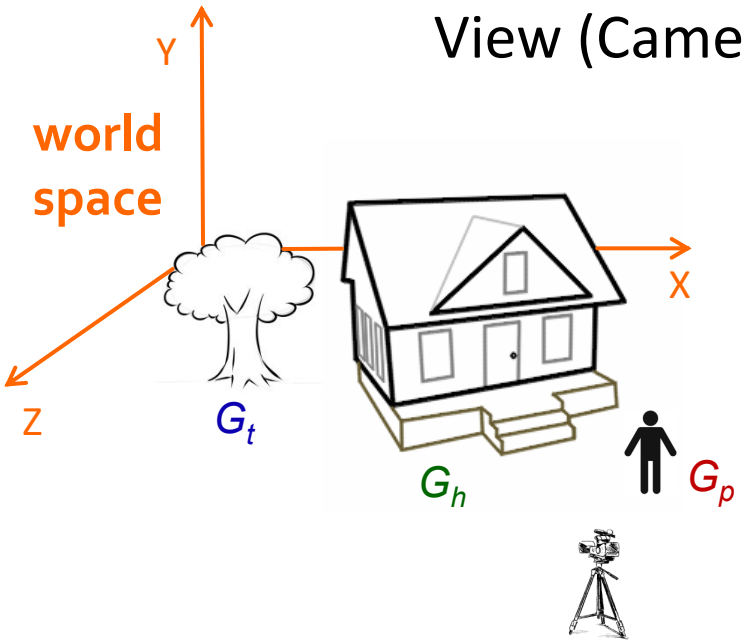


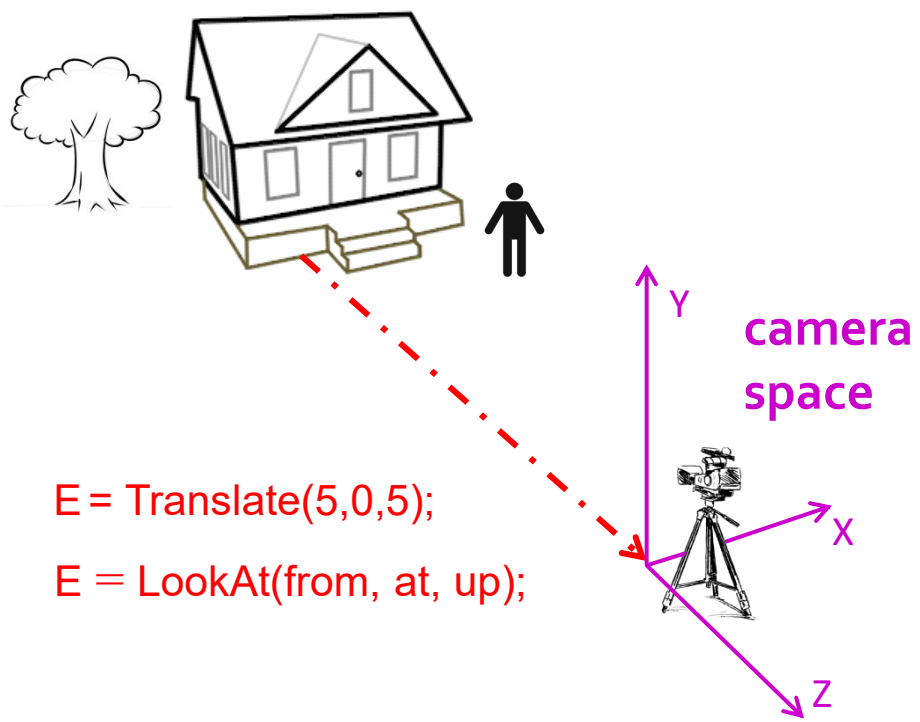
Albrecht Durer

Object (or Model) Space (or Coordinates) and World (or Scene) Space



World (Scene) Space and View (Camera or Eye) Space





G_t = Scale(.8)RotateX(45)Translate(2,2,0);

G_h = Scale(1.5)Translate(3,2,0);

G_p = Scale(.2)Translate(4,2,0);

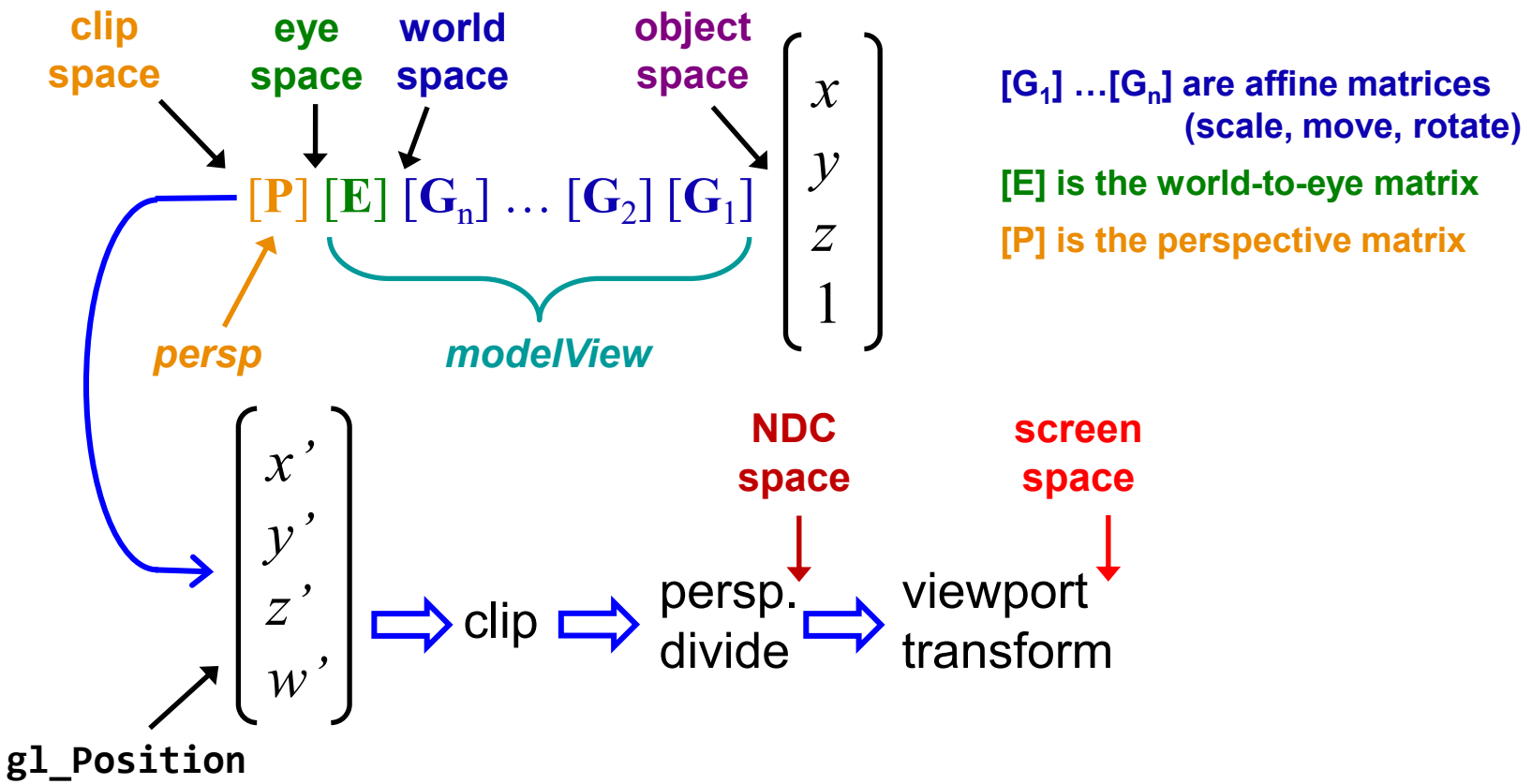
E = Translate(5,0,5);

E = LookAt(from, at, up);

A point **p** is transformed from object space to camera space via a compound transformation, with the geometry matrix closest to **p**

p' = E G p

Before the Rasterizer

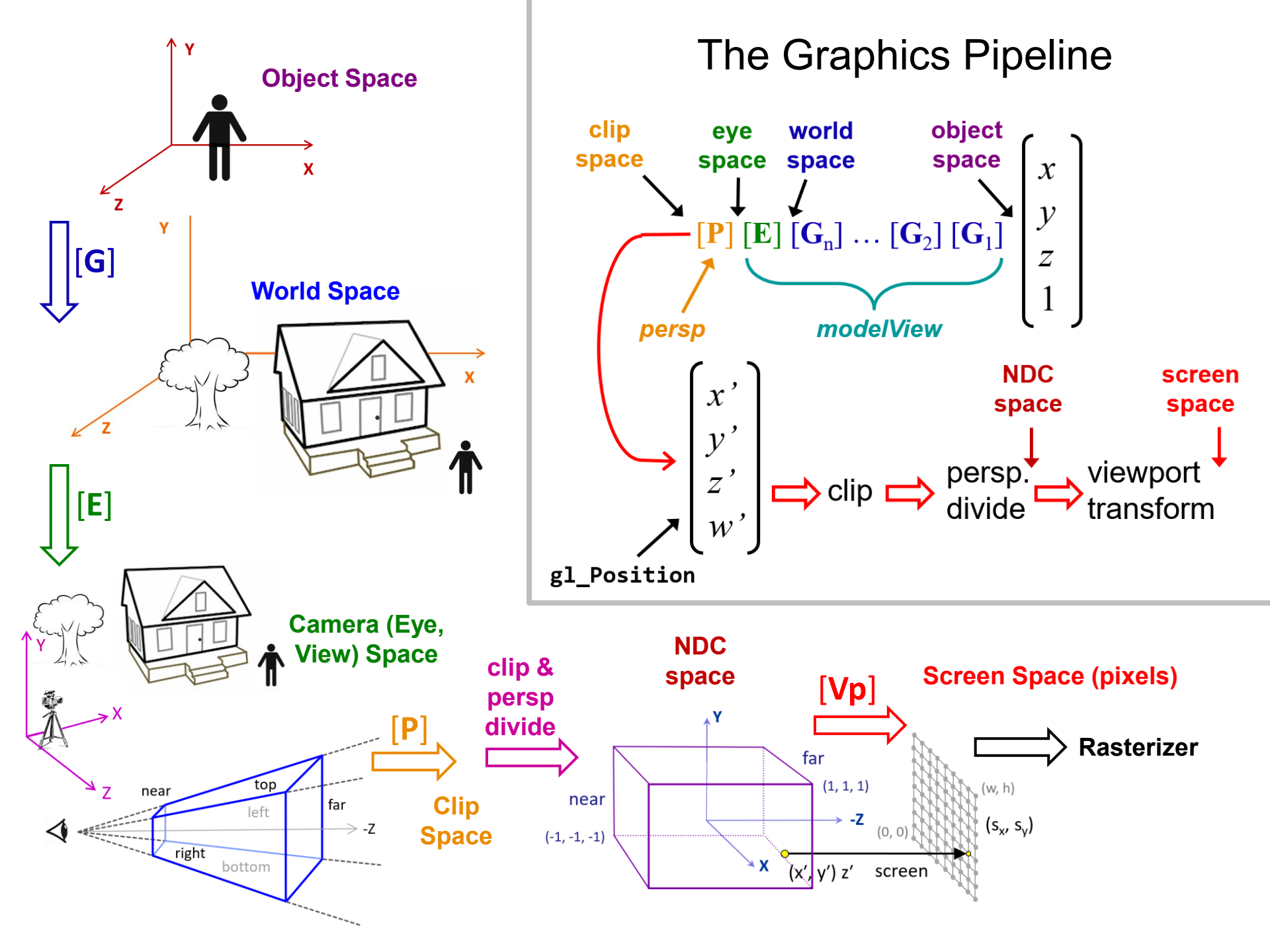


(x, y, z) transformed by **[G]** and **[E]** is in *eye space* and used by the pixel shader for *shading*

(x', y', z') includes transformation by **persp** for use by the rasterizer to determine pixels covered

In code: `gl_Position = persp*modelView*vec4(x,y,z,1)`

```
// the effect on (x,y,z) is accumulated right to left
```


Object Space
Y
X
Z
[G]
World Space
[E]
Camera (Eye, View) Space
[P]
Clip Space
clip & persp divide
NDC space
[Vp]
Screen Space (pixels)
Rasterizer
near
top
left
far
right
bottom
-Z
(-1, -1, -1)
(1, 1, 1)
(0, 0)
(w, h)
(x', y') z'
screen
The Graphics Pipeline
clip space
eye space
world space
object space
[P] [E] [Gn] … [G2] [G1]
persp
modelView
gl_Position
clip
persp. divide
viewport transform
NDC space
screen space

Concatenations

Each individual object can be assigned its own geometry matrix.

The geometry [G] and camera [E] matrices may be concatenated into a single “modelview” matrix

[MV] = [E][G], and

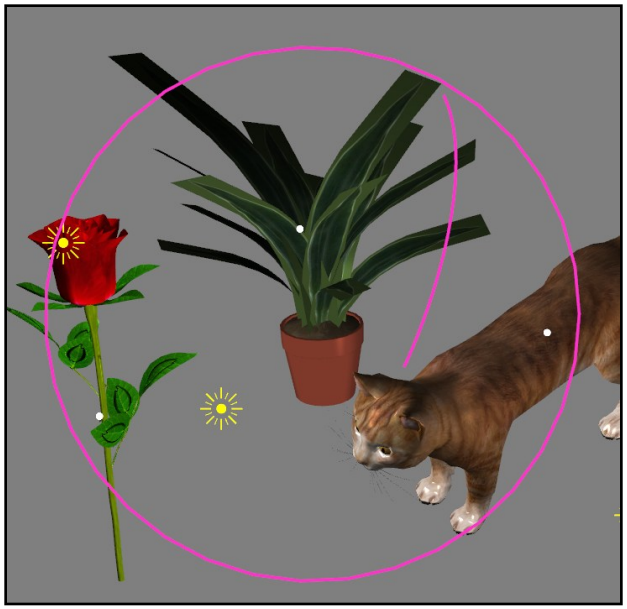
p’ = [MV]p

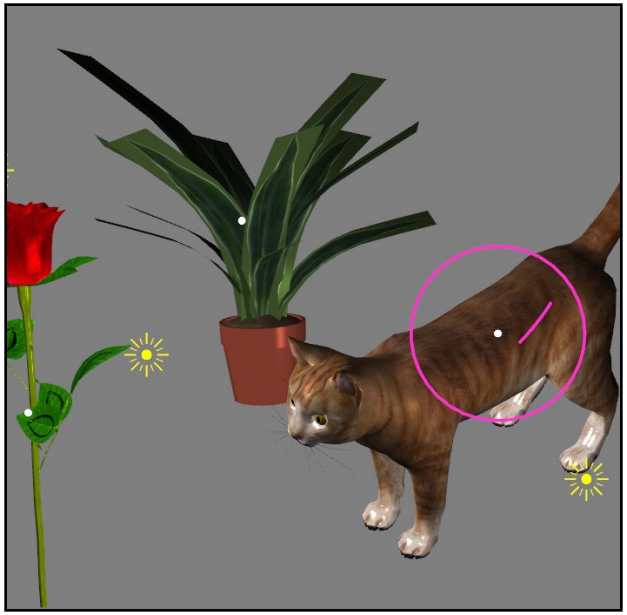
In code:

```
mat4 modelview = E*G;

vec4 pTransform = modelview*vec4(p, 1);
```

Multiplication of matrices left to right is the reverse order of the transformations applied to a point. Rule of thumb: the matrix closest to the point is the first transformation applied to the point.

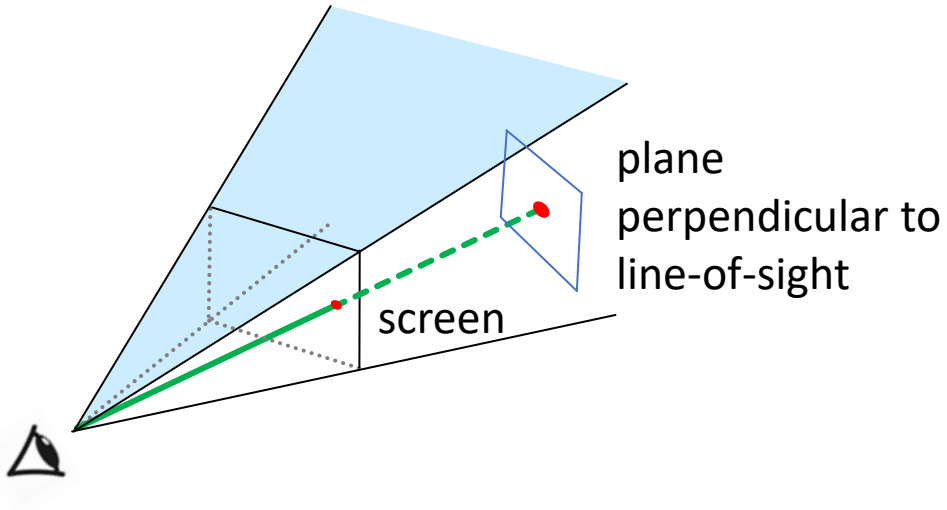
Arcball





```
vec3 Arcball::BallV(vec2 mouse) {
    // return point on sphere with radius in pixels
    vec2 dif(mouse-center);
    float difLen = length(dif);
    if (difLen > .97f*radius)
        dif *= (.97f*radius/difLen);
    float sq = radius*radius-dot(dif, dif);
    return normalize(vec3(dif.x, dif.y, sqrt(sq)));
}

void Arcball::Down(int x, int y, mat4 modelview) {
    modelviewInverse = Invert(modelview);
    mouseDown = vec2((float) x, (float) y);
    qstart = Quaternion(*m);
    scale = MatrixScale(*m);
}

Quaternion Arcball::Drag(int x, int y) {
    mouseMove = vec2((float) x, (float) y);
    vec3 v1 = BallV(mouseDown), v2 = BallV(mouseMove);
    vec3 axis = cross(v2, v1);
    if (dot(axis, axis) > .000001f) {
        if (use == Use::Body)
            axis = Vec3(modelviewInverse*vec4(axis, 0));
        Quaternion qrot(axis, (float) acos(dot(v1, v2)));
        qq = qstart*qrot;
        mat3 m3 = scale*qq.Get3x3();
        for (int i = 0; i < 3; i++)
            for (int j = 0; j < 3; j++)
                (*m)[i][j] = m3[i][j];
    }
    return qrot;
}
```

Mover



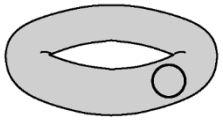

```
vec3 Unproject(float xscreen, float yscreen, float zscreen, mat4 &inv, int4 vp) {
    vec4 ndc(2*(xscreen-vp[0])/vp[2]-1, 2*(yscreen-vp[1])/vp[3]-1, 2*zscreen-1, 1);
    vec4 q = inv*ndc;
    return vec3(q.x, q.y, q.z)/q.w;
}

void ScreenLine(float xscreen, float yscreen, mat4 modelview, mat4 persp, vec3 &p1, vec3 &p2) {
    // compute 3D world space line, given by p1 and p2, that transforms
    // to a line perpendicular to the screen at (xscreen, yscreen)
    int4 vp = VP();
    mat4 fullview = persp*modelview, inv = Invert(fullview);
    p1 = Unproject(xscreen, yscreen, .25f, inv, vp);
    p2 = Unproject(xscreen, yscreen, .50f, inv, vp);
}
```

A mesh can be convex or concave (at least one cavity)

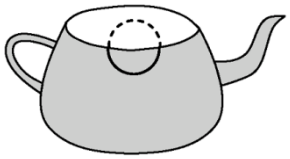
Closed (concave or convex) meshes may be rendered using glFrontFacing

A mesh can be manifold, manifold-with-boundary, or non-manifold
Most CG applications deal with manifolds

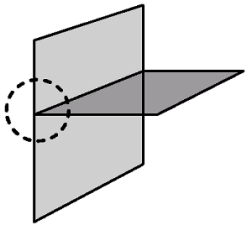
A manifold mesh can be displayed by rendering forward faces only. if the manifold mesh is genus 0, it can be displayed without a z-buffer by rendering forward faces only, in any order



manifold



manifold-with-boundary



non-manifold

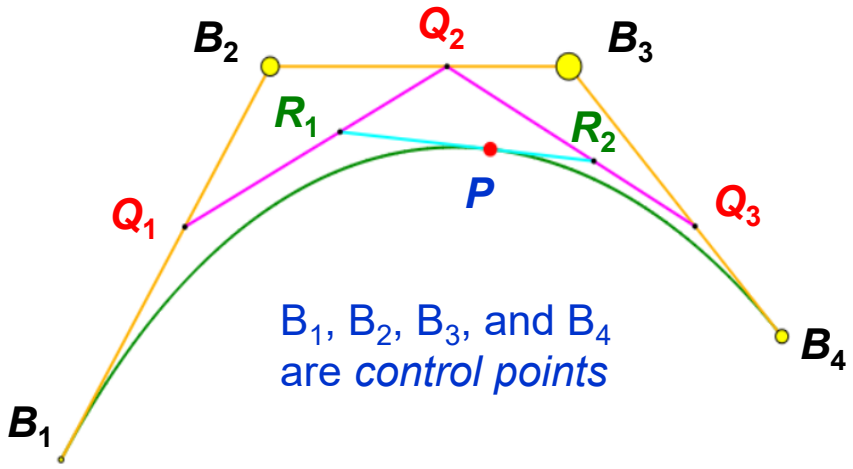
A mesh can be
- connected as in “points-polys” (eg, OBJ)
  - structured mesh with edges, permitting queries:
    - vertices of a triangle
    - edges of a triangle
    - vertices of an edge
    - triangles to each side of edge
    - triangles around a vertex
- disconnected as in “triangles” (eg, STL).

de Casteljau Construction of Bézier Curve

$Q_1 = B_1+t\,(B_2-B_1)$ ← linear

$Q_2 = B_2+t\,(B_3-B_2)$

$Q_3 = B_3+t\,(B_4-B_3)$



$$R_1 = Q_1+t(Q_2-Q_1)$$
$$= B_1+t(B_2-B_1)+t(B_2+t(B_3-B_2))-t(B_1+t(B_2-B_1))$$
$$= B_1+tB_2-tB_1+tB_2+t^2B_3-t^2B_2-tB_1-t^2B_2+t^2B_1$$
$$= (1-2t+t^2)B_1+(2t-2t^2)B_2+t^2B_3$$

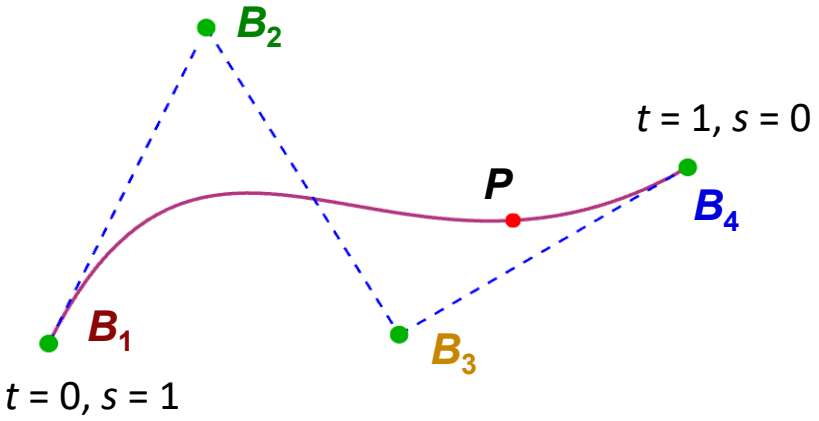
$$R_2 = Q_2+t(Q_3-Q_2)$$
$$= B_2+t(B_3-B_2)+t(B_3+t(B_4-B_3))-t(B_2+t(B_3-B_2))$$
$$= B_2+tB_3-tB_2+tB_3+t^2B_4-t^2B_3-tB_2-t^2B_3+t^2B_2$$
$$= (1-2t+t^2)B_2+(2t-2t^2)B_3+t^2B_4$$ ← quadratic

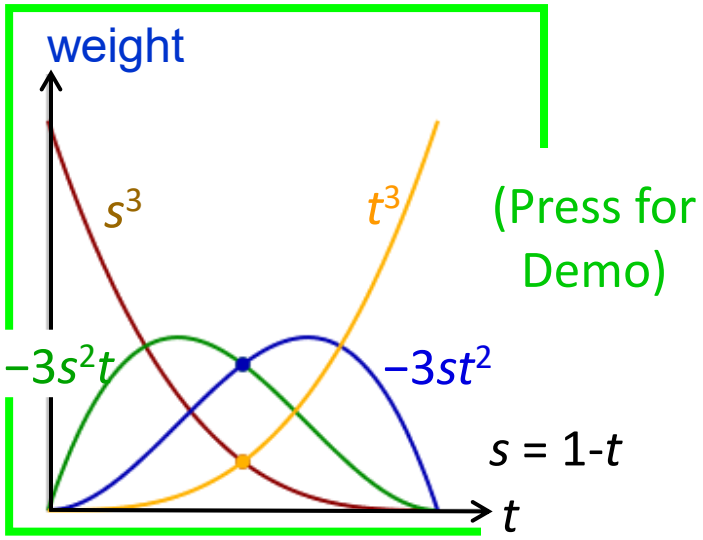
$$P = R_1+t(R_2-R_1) = R_1-tR_1+tR_2$$
$$= (1-2t+t^2)B_1+(2t-2t^2)B_2+t^2B_3-t((1-2t+t^2)B_1+(2t-2t^2)B_2+t^2B_3)+t((1-2t+t^2)B_2+(2t-2t^2)B_3+t^2B_4)$$

$$= (-t^3+3t^2-3t+1)B_1+(3t^3-6t^2+3t)B_2+(-3t^3+3t^2)B_3+t^3B_4$$ ← cubic

$$= s^3B_1-3s^2tB_2-3st^2B_3+t^3B_4, \text{ where } s = 1-t$$

The Bézier Basis Functions





Bernstein basis functions for four Bézier control points

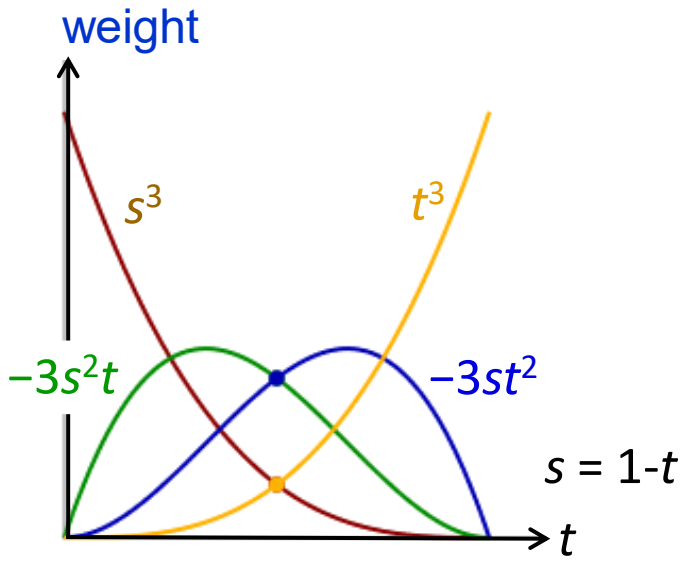
$$P = s^3 B_1 - 3ts^2 B_2 - 3st^2 B_3 + t^3 B_4$$

The basis functions are cubic and polynomial, so the Bézier curve is a *cubic polynomial curve*.

The coefficients are “binomial coefficients” and may be computed are an arbitrary number of control points using the binomial theorem. Four points permit an inflection but avoid control issues of higher order curves.

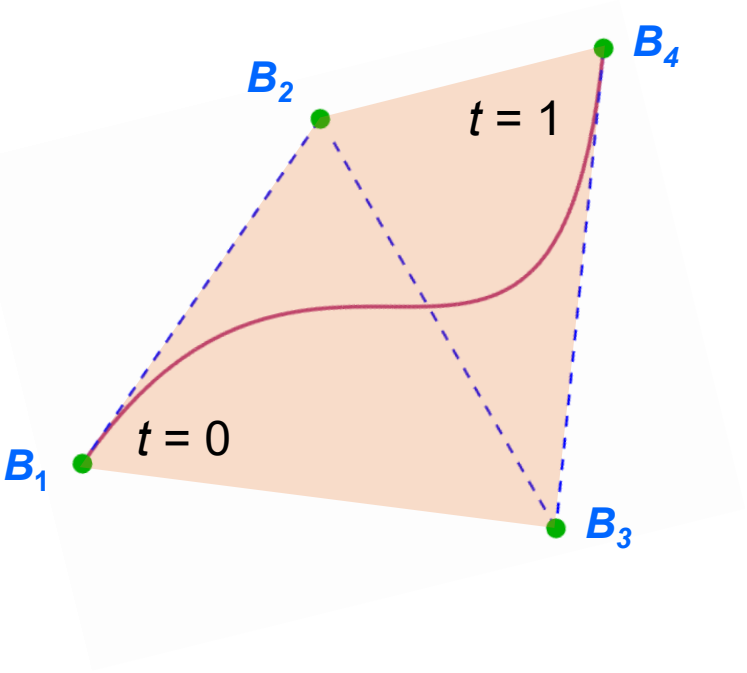
These basis (*i.e.*, *weighting*) functions belong to the family of *Bernstein polynomials*. The Bézier curve is one type of *B-spline curve* (*B* for *Bernstein*).

Interpolation/Approximation, Convex Hull

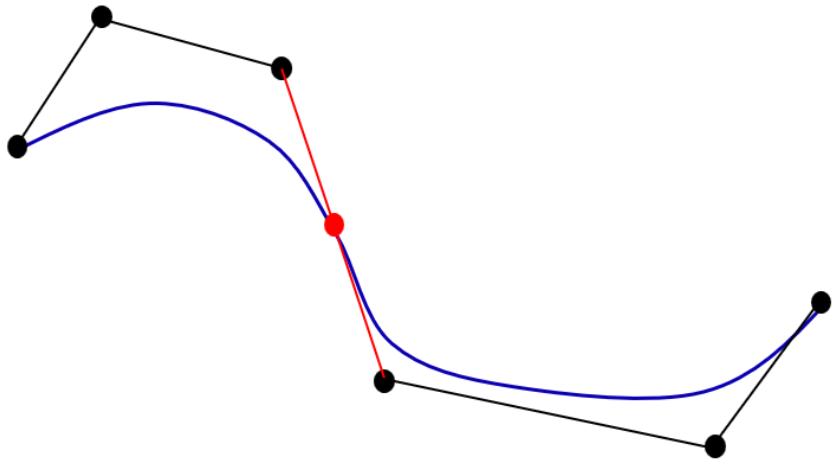


These functions form a *partition of unity* (the sum of the four functions is 1 for any *t*), which implies the entire curve is contained within the *convex hull* of the control points

Because only the first and last basis functions reach 1, the Bézier curve interpolates B_1 and B_4 (the outer control points) and approximates B_2 and B_3 (the inner points)



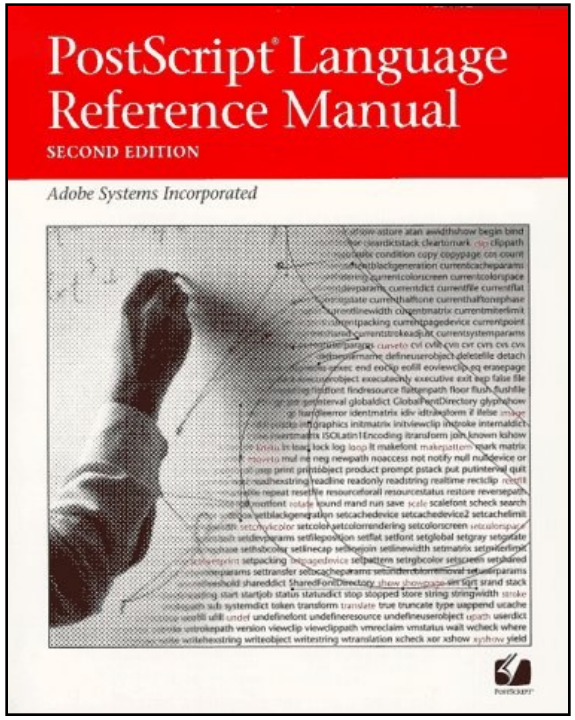
Piecewise, Continuous Bézier Curves

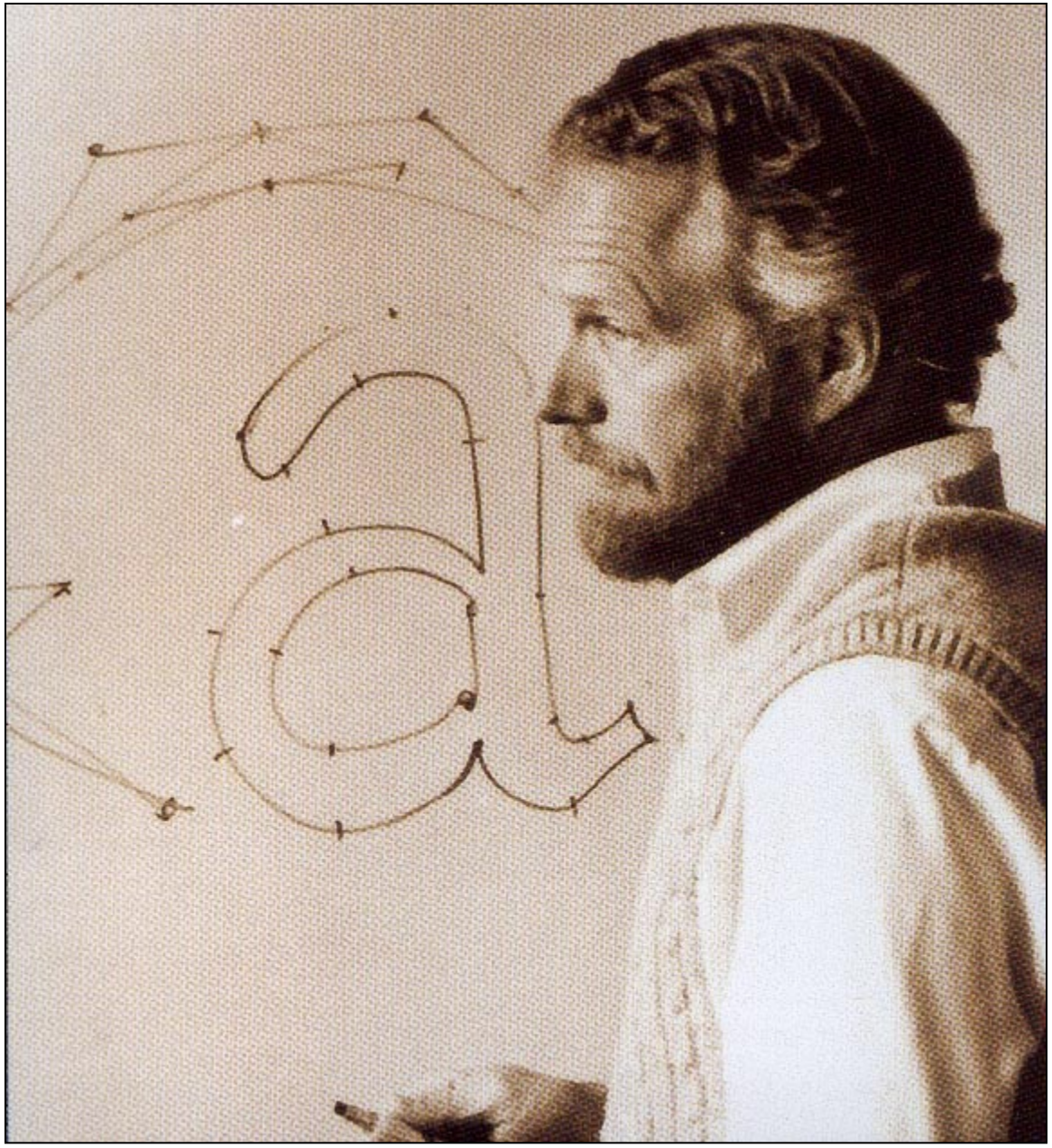


Two Bézier curves may be combined if the last control point of one is coincident with the first of the second.

Tangency (*C^1 continuity*) can be maintained if the coincident point is midway between the 3rd point of the 1st curve and the 2nd point of the 2nd curve.

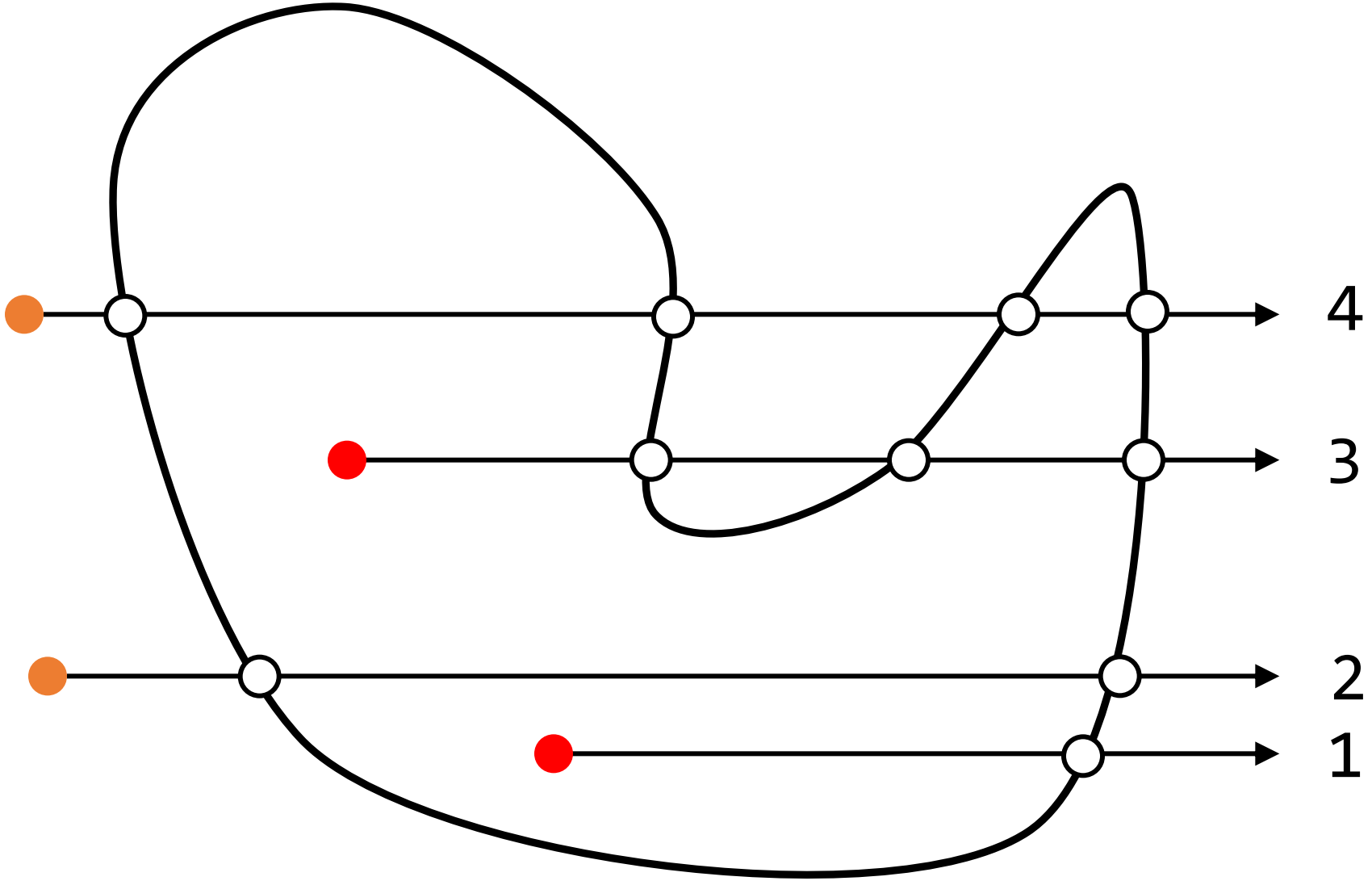
Text





John Warnock, Adobe founder

Insidedness



Text

Preparation

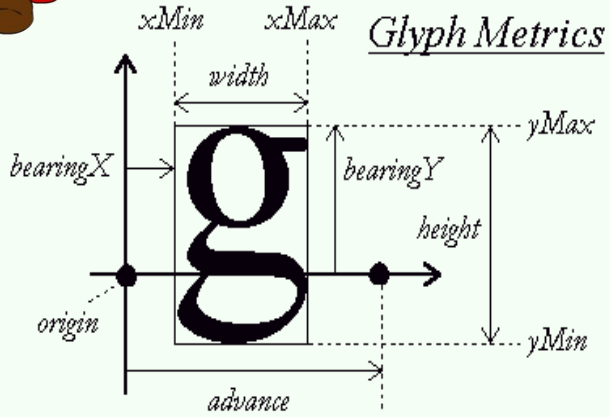
In Text::SetCharacterSet(), for each character in set:

```
FT_Load_Char(face, c, FT_LOAD_RENDER);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RED, g->bitmap.width,
             g->bitmap.rows, 0, GL_RED, GL_UNSIGNED_BYTE,
             g->bitmap.buffer);
```

Rendering

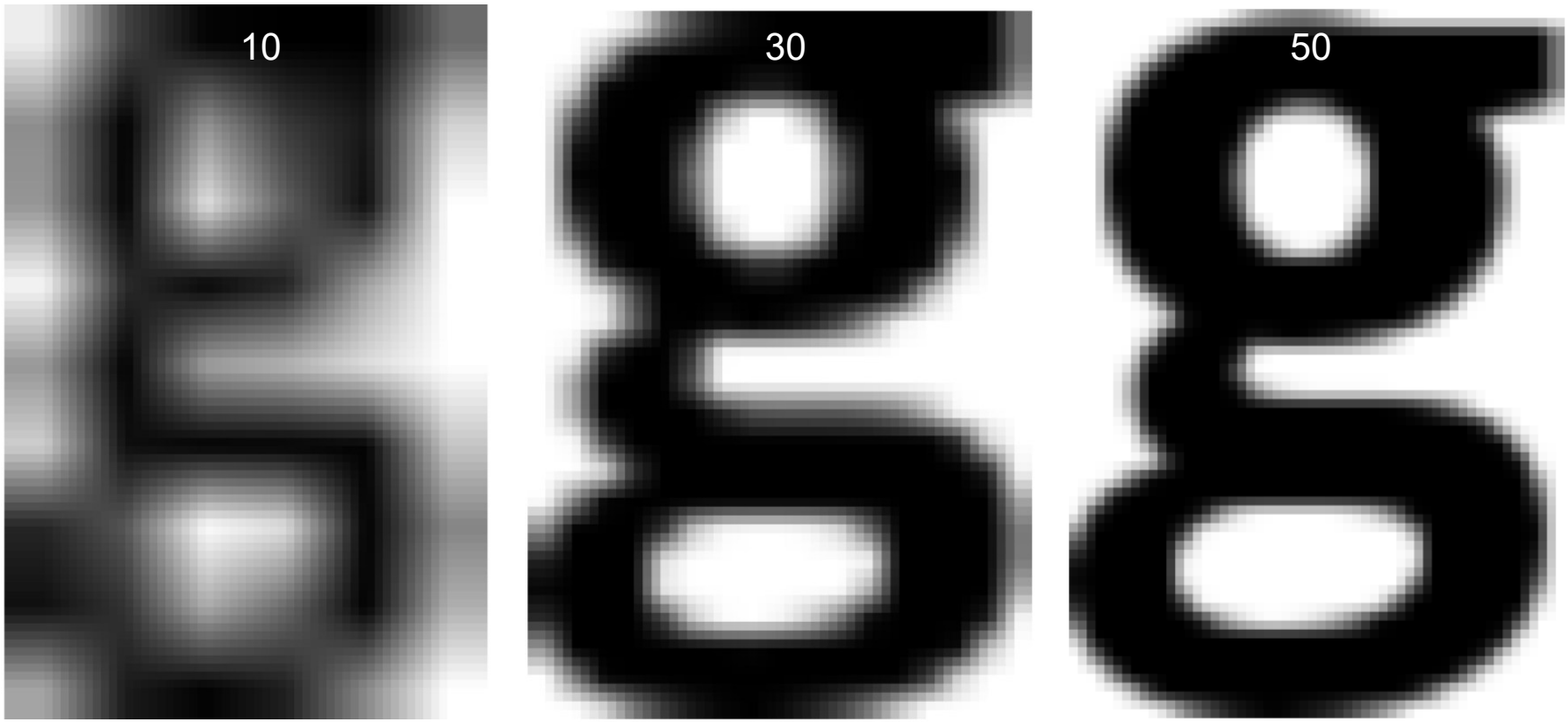
In Text::RenderText(), for each character in string:

```
for (const char *c = text; *c; c++) {
    // render glyph texture with quad
    Character ch = currentFont->characters[(int)*c];
    float xpos = x+ch.bearing.i1*scale;
    float ypos = y-(ch.gSize.i2-ch.bearing.i2)*scale;
    float w = ch.gSize.i1*scale, h = ch.gSize.i2*scale;
    glBindTexture(GL_TEXTURE_2D, ch.textureID
    // 4 vertices, each {x, y, u, v}
    float vrts[][4] = {
        {xpos, ypos+h, 0, 0},
        {xpos+w, ypos+h, 1, 0},
        {xpos+w, ypos, 1, 1},
        {xpos, ypos, 0, 1}
    };
    glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vrts), vrts);
    glDrawArrays(GL_QUADS, 0, 4);
};
```




```
const char *textPixelShader = R"(
    #version 130
    in vec2 vUv;
    out vec4 pColor;
    uniform sampler2D image;
    uniform vec3 color;
    void main() {
        float a = texture(image, vUv).r;
        pColor = vec4(color, a);
    }
)";
```

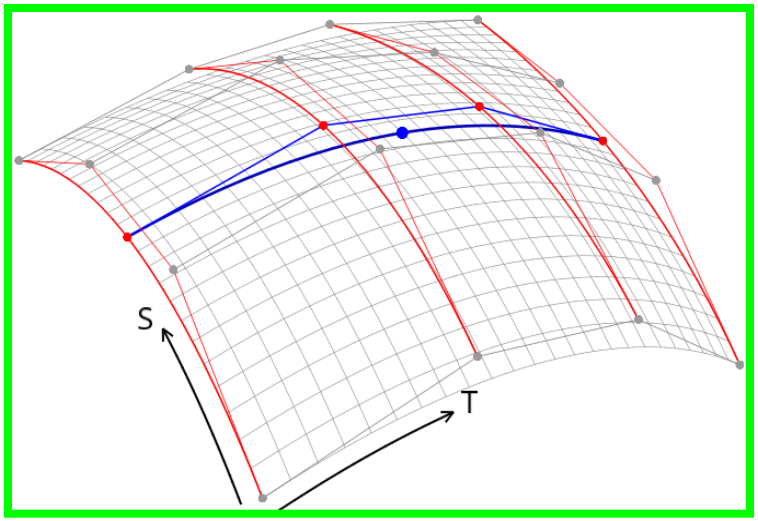
bitmap.buffer resolution



FontSpiral.cpp

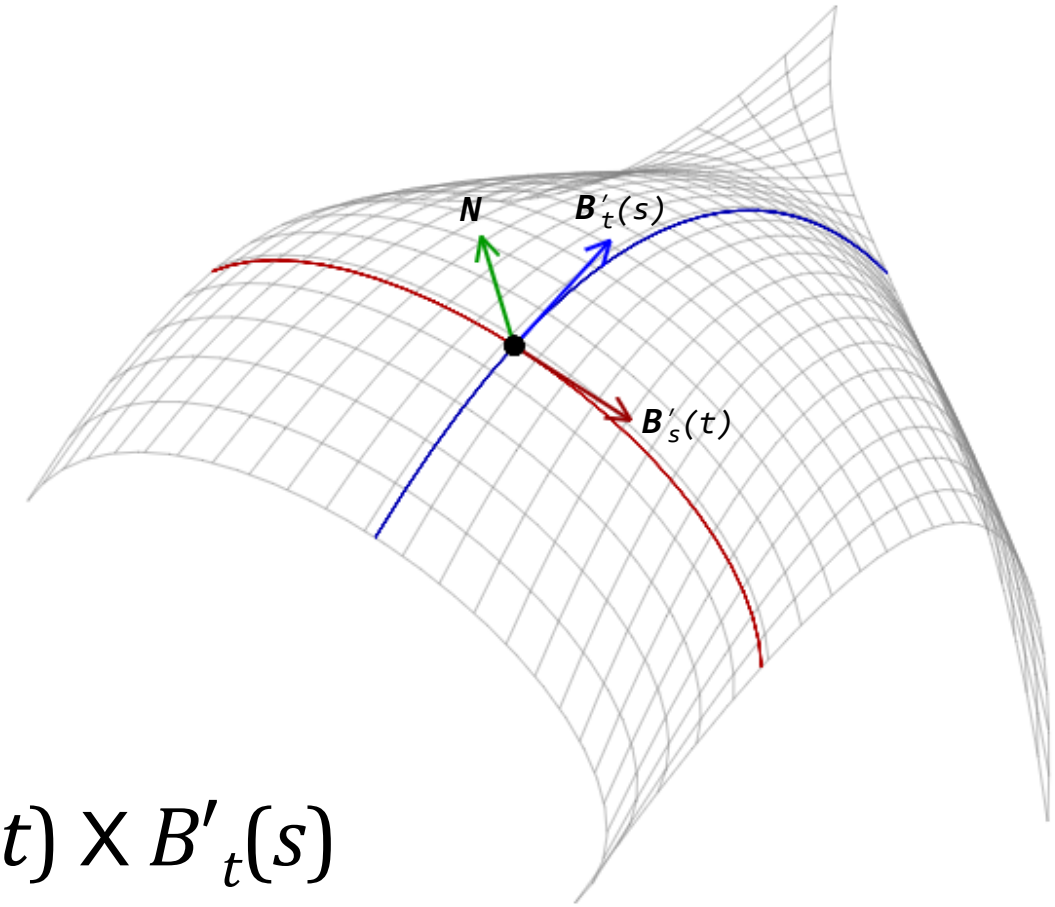
```
vec2 Spiral(float t) {
     float alpha = t/50, mag = .3f*alpha*alpha;
     return mag*vec2(cos(alpha), sin(alpha));
}

void SpiralText(const char *text, float x, vec3 color, float scale, mat4 view) {
     // create quad vertex buffer and build characters
     glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
     glBufferData(GL_ARRAY_BUFFER, sizeof(float)*6*4, NULL, GL_DYNAMIC_DRAW);
     VertexAttribPointer(shaderProgram, "point", 4, 4*sizeof(float), 0);
     SetUniform(shaderProgram, "view", view);
     SetUniform(shaderProgram, "color", color);
     glActiveTexture(GL_TEXTURE0);
     glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
     for (const char *c = text; *c; c++) {
         Character ch = font->characters[(int)*c];
         float xpos = x+ch.bearing.i1*scale, w = ch.gSize.i1*scale;
         glBindTexture(GL_TEXTURE_2D, ch.textureID);
         vec2 cen((float) winWidth/2, (float) winHeight/2);
         vec2 p1 = Spiral(xpos), p4 = 1.2f*p1;
         vec2 p2 = Spiral(xpos+w), p3 = 1.2f*p2;
         vec4 vertices[] = {vec4(cen+p1, 0, 0), vec4(cen+p2, 1, 0),
                            vec4(cen+p3, 1, 1), vec4(cen+p4, 0, 1)};
         glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vertices), vertices);
         glDrawArrays(GL_QUADS, 0, 4); // render glyph texture with quad
         x += (ch.advance >> 6)*scale; // advance in terms of 1/64 pixel
     }
}
```

The Bézier Patch



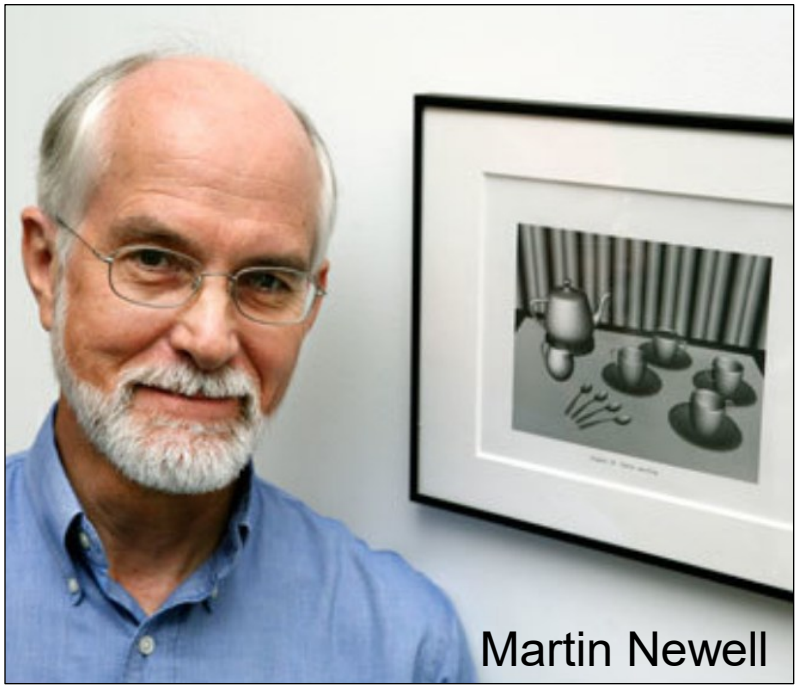
t-curve made from s-points
s-curve made from t-points

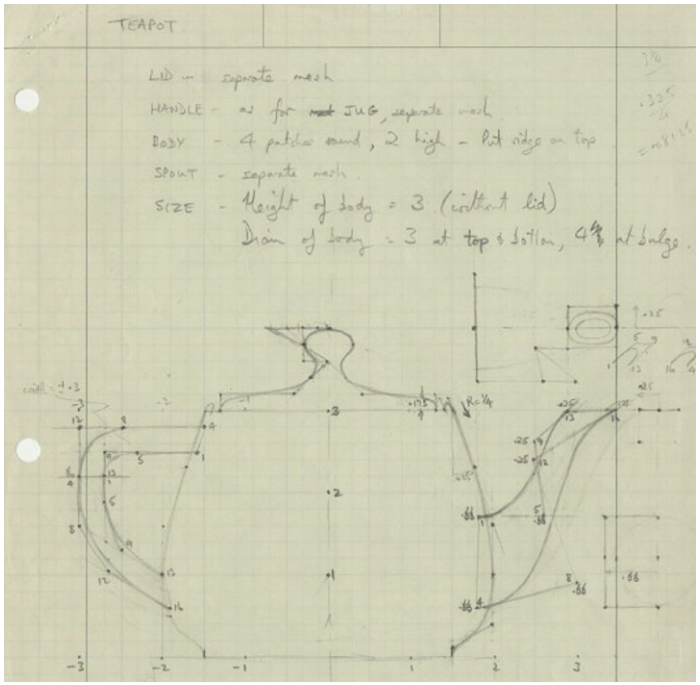
Computing the Normal

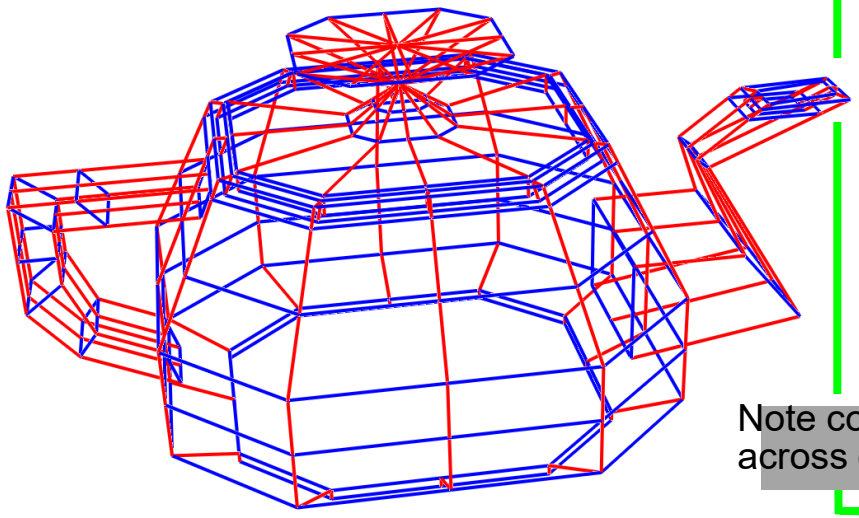


$$N(s,t) = B'_s(t) \times B'_t(s)$$

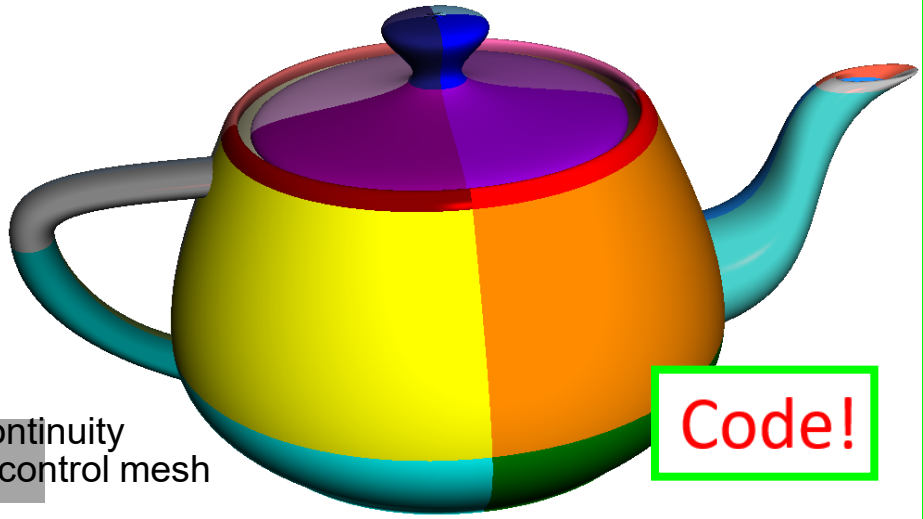
The Utah Teapot





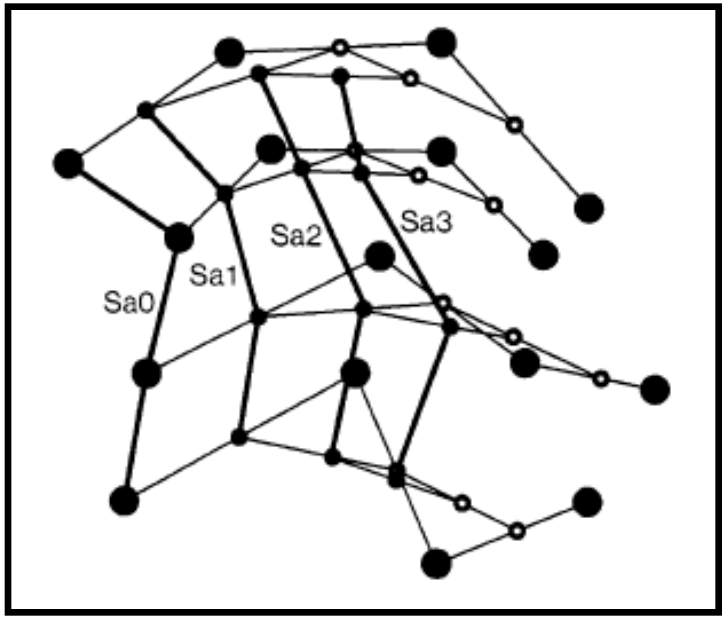


Note continuity
across control mesh



Patch Splitting

A bi-cubic Bezier patch can be subdivided into two sub-patches along the *s*-direction; these two sub-patches are then subdivided along the *t*-direction:



Continuity

Continuity along the boundary of Bezier patches follows the constraints imposed for continuity of Bezier curves.

Tessellation (control, evaluation) Shader(s)

Shader architecture permits a programmer access to data throughout the graphics pipeline

It also permits the pipeline to be reconfigured - for example: the tessellation shader

Although a hardware effort was made as early as 2001, it was not until 2008 (DirectX) and 2010 (OpenGL) that tessellation was available for commodity computers

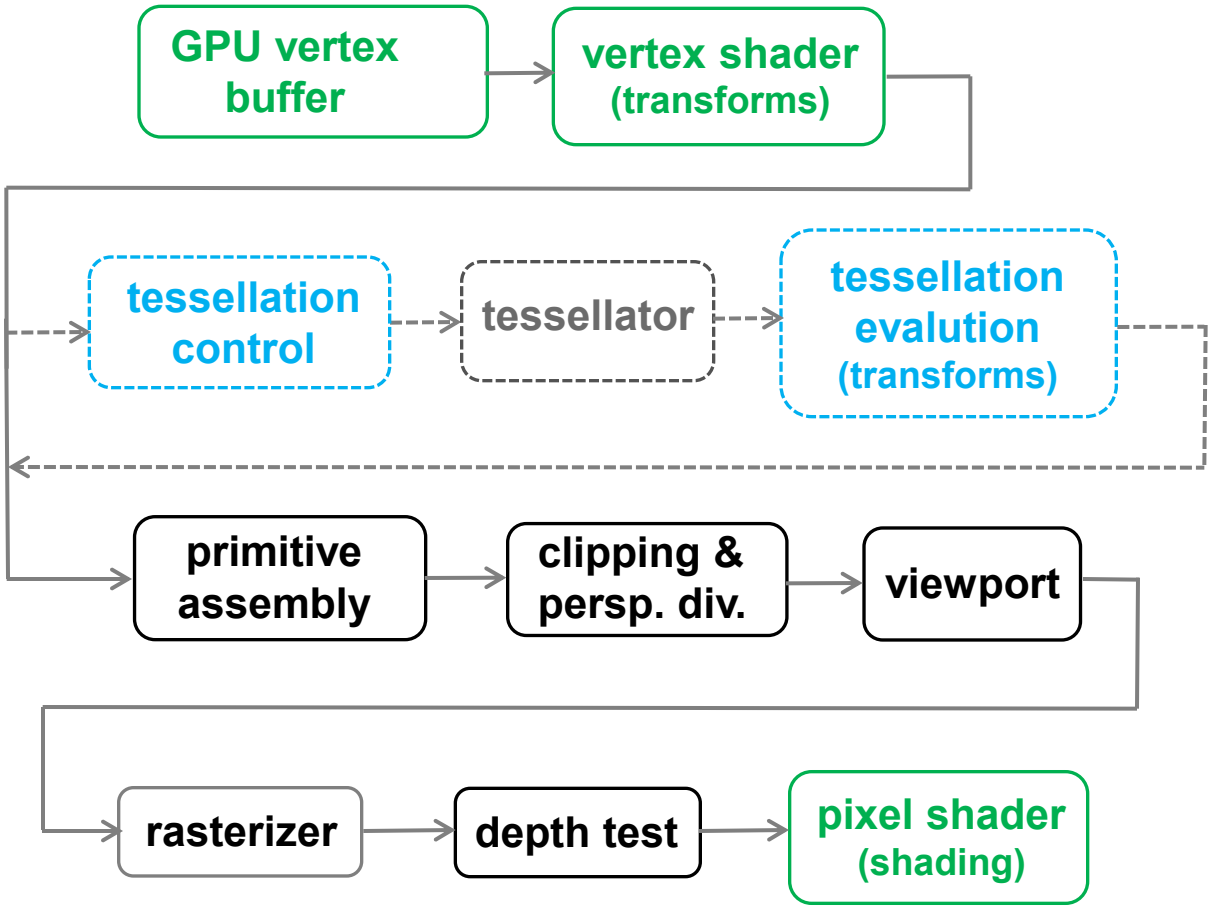
Tessellation increases resolution without burdening the application with additional vertex/triangle complexity or additional CPU/GPU transfers

Conventionally, with the vertex and pixel shaders,

1) a vertex is fetched from a GPU vertex buffer
2) the vertex is processed by the vertex shader
3) vertices are accumulated as triangles (quads)
4) the triangles (quads) are rasterized
5) pixels are shaded

The tessellation shader permits a triangle or quad to become multiple triangles (or quads) before being passed to the rasterizer

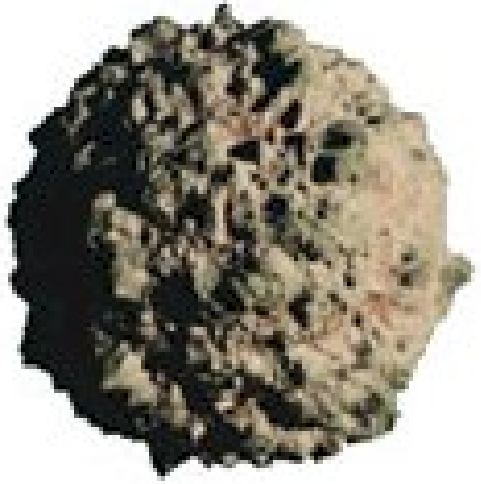
Revised Graphics Pipeline



The (optional) tessellation shader follows the vertex shader; once a primitive (triangle or quad) is assembled, it is subdivided into smaller pieces. The new vertices and primitives are then passed to the rasterizer.

Texture-
Mapped

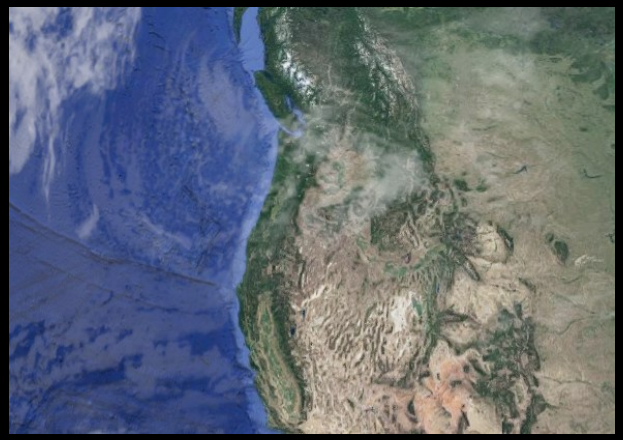
Bump-
Mapped

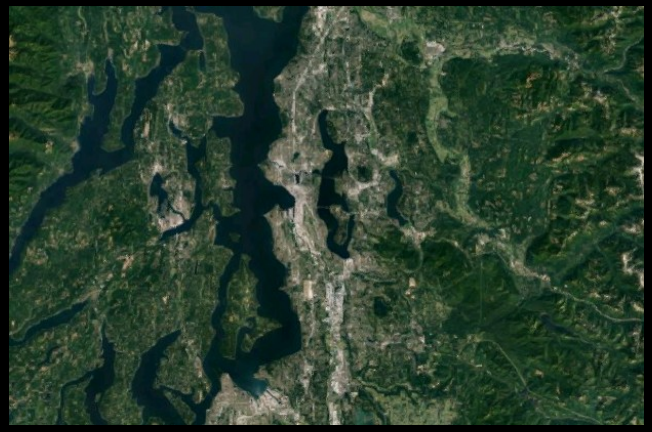


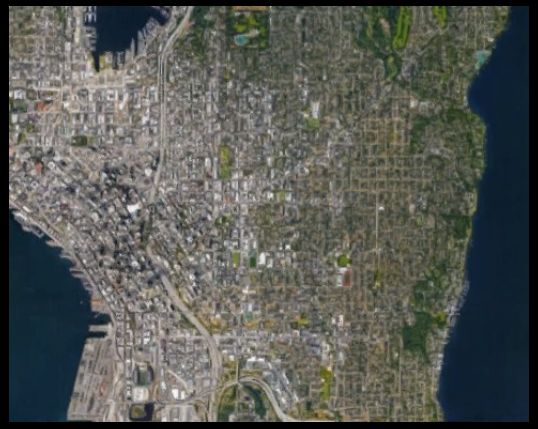
Displacement-
Mapped

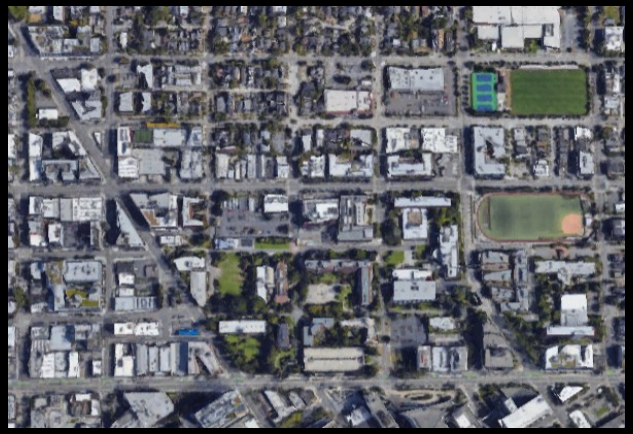
The difference in realism is evident
from the shading and silhouette edges

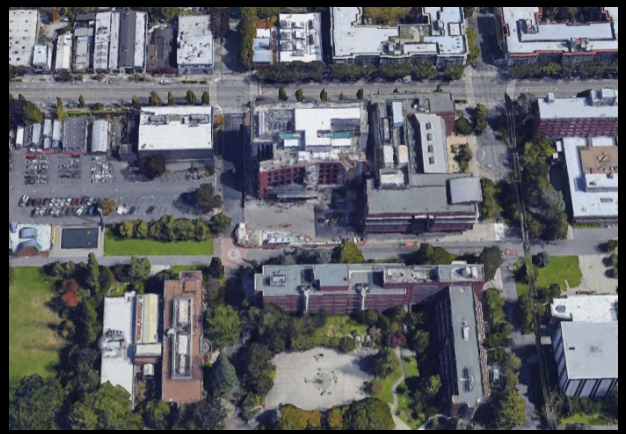
Level of Detail



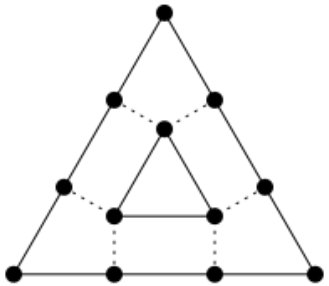




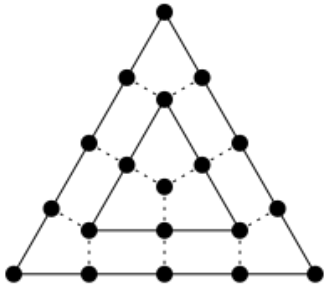




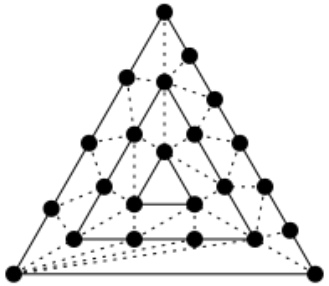
Triangle Inner and Outer Levels, Examples



Inner Tess = 3



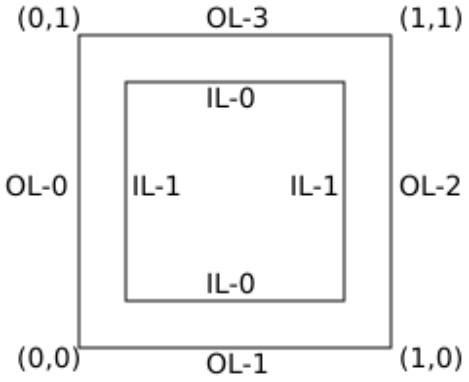
Inner Tess = 4

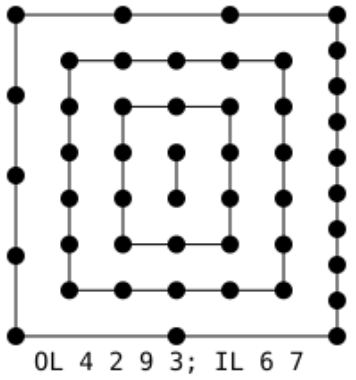


OL 4 1 6; IL 5

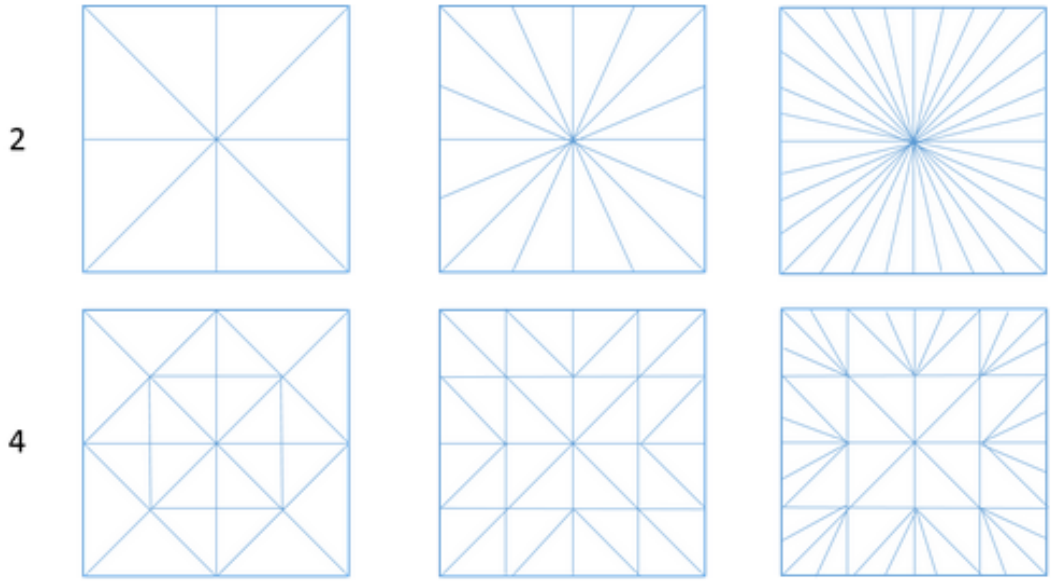
```
glPatchParameteri(GL_PATCH_VERTICES, 3);
float outerLevels[] = {4, 1, 6}, innerLevels[] = {5, 5, 5};
glPatchParameterfv(GL_PATCH_DEFAULT_OUTER_LEVEL, outerLevels);
glPatchParameterfv(GL_PATCH_DEFAULT_INNER_LEVEL, innerLevels);
```

Quad Inner and Outer Levels





Inner Tessellation Level



Example Tessellation Control Shader for a Quad

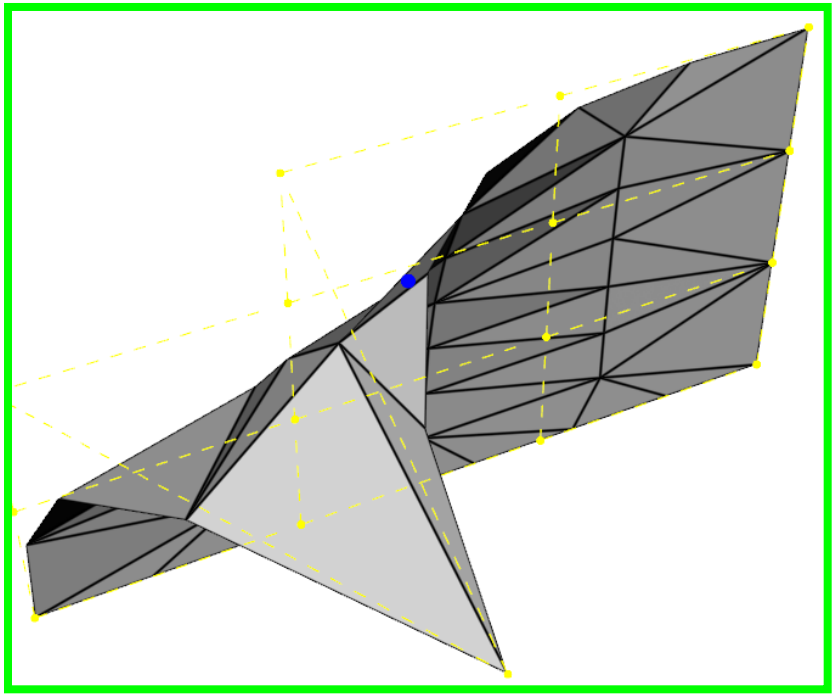
```
#version 420
layout (vertices = 4) out;
uniform mat4 modelview, projection;
void main() {
    if (gl_InvocationID == 0) {
        mat4 m = projection*modelview;
        vec2 pos[4]; // quad points in screen space
        for (int i = 0; i < 4; i++) {
            vec4 h = m*gl_in[i].gl_position;
            pos[i] = h.xy/h.w;
        }
        float t[4];
        for (int i = 0; i < 4; i++)
            t[i] = max(2, 16*distance(pos[i], pos[(i+1)%4]));
        float tmax = max(t[0], max(t[1], max(t[2], t[3])));
        gl_TessLevelInner[0] = tmax;    // set inner to be max of outer
        gl_TessLevelInner[1] = tmax;
        gl_TessLevelOuter[0] = t[0];    // outer res, in ccw order
        gl_TessLevelOuter[1] = t[1];
        gl_TessLevelOuter[2] = t[2];
        gl_TessLevelOuter[3] = t[3];
    }
    gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
}
```

This tessellation control shader allows for view-dependent resolution.

main acts as a pass-through; on the first call, however, is the opportunity to set the tessellation resolution factors given the input array gl_in[].

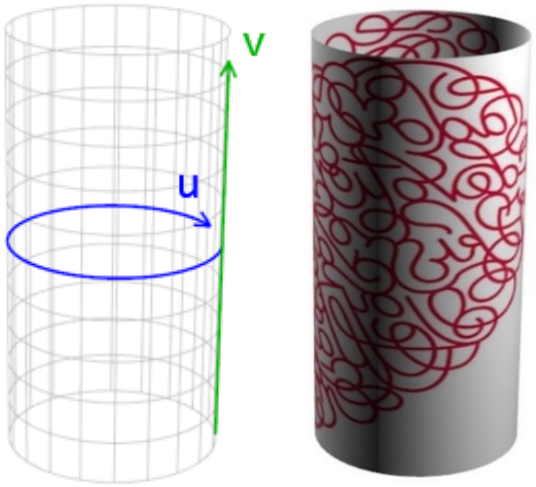
If outer levels of adjacent patches match, no gaps result (assuming subdivision is linear along edge).

BézierLOD

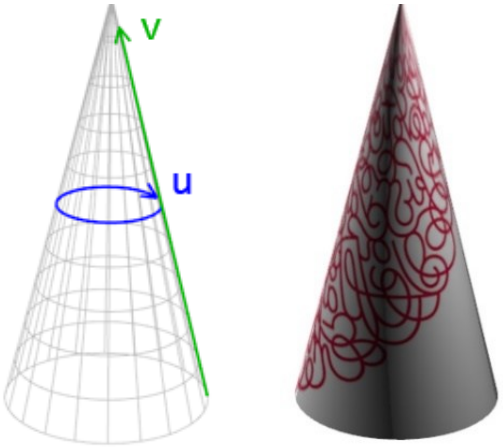


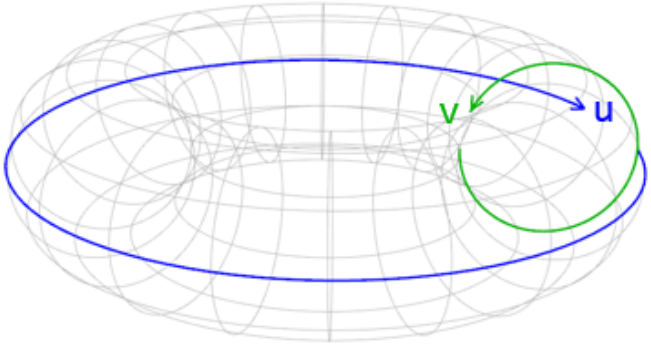
See Code!

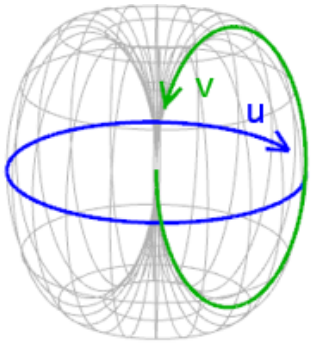
In the case of the Bezier patch, resolution can be established by a tessellation control shader using the 16 control points given by a uniform array

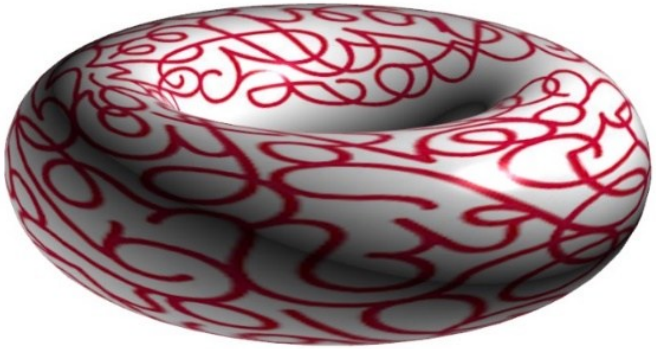


Tessellation
Interpolation









Alternatives to Parametric Surfaces

Subdivision Surfaces

blend via recursive smoothing of a polygonal mesh

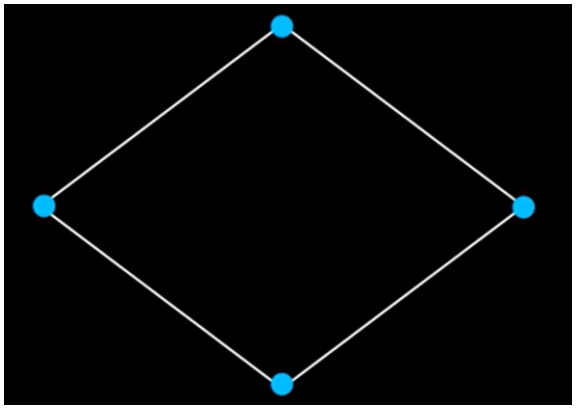
Implicit Surfaces

blend via summation of primitives

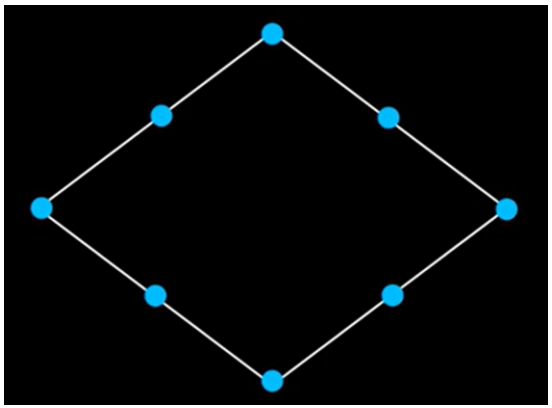
Particle Systems

individual points acting independently, individually displayed

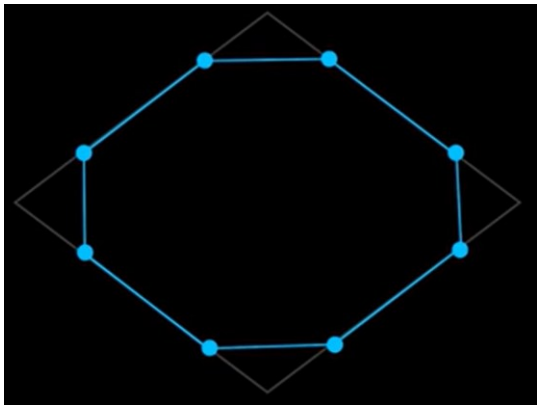
Catmull-Clark Subdivision Surfaces



object defined by 4 points

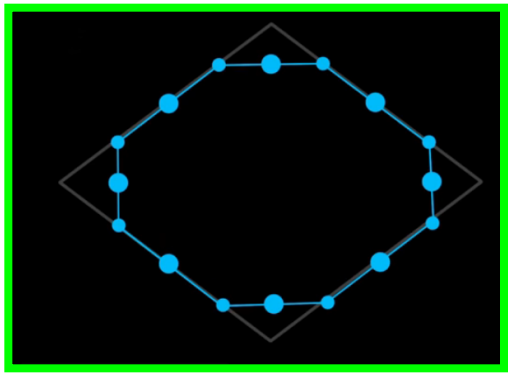


edge midpoints added



move each point halfway towards its cw neighbor

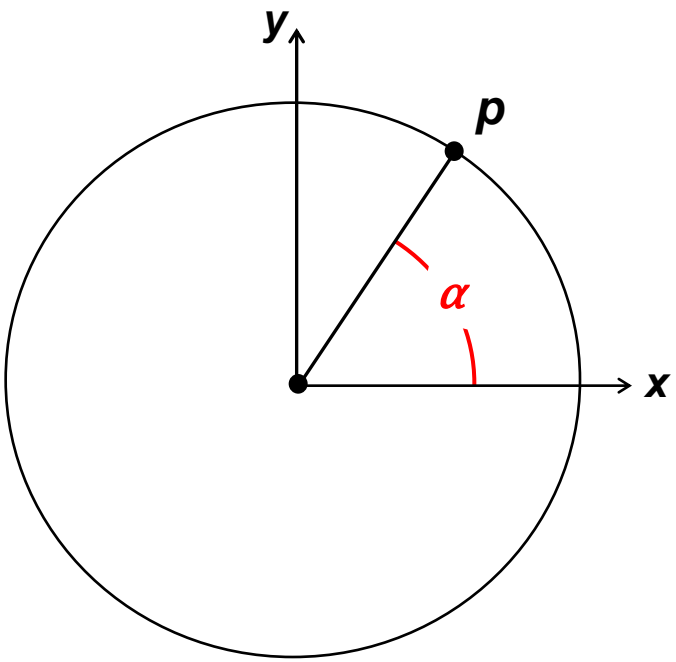
A surface becomes more smooth on each iteration



Khan Academy: Pixar Character Modeling

Implicit Surfaces

Three representations for the unit circle:



parametric

$p = (\cos(\alpha), \sin(\alpha)),\ \alpha \in 0, 2\pi$

equiangular

(transcendental trigonometric)

$p = (\pm(1-t^2)/(1+t^2), \pm 2t/(1+t^2))$

$t \in 0, 1$

non-equiangular

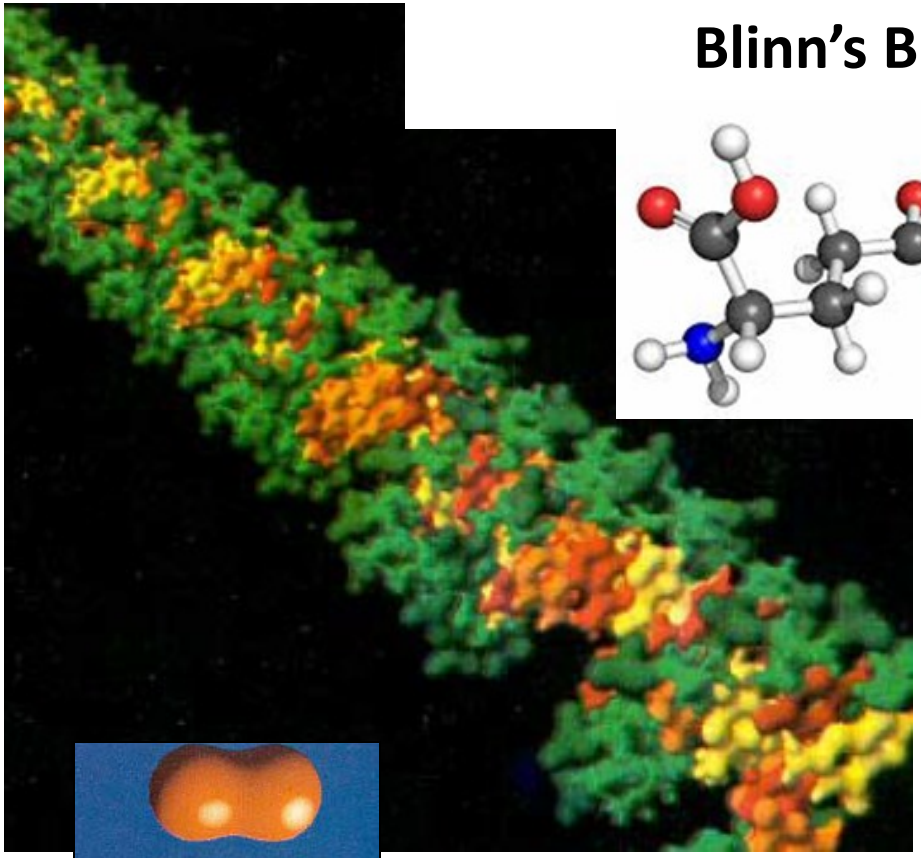
(rational trigonometric)

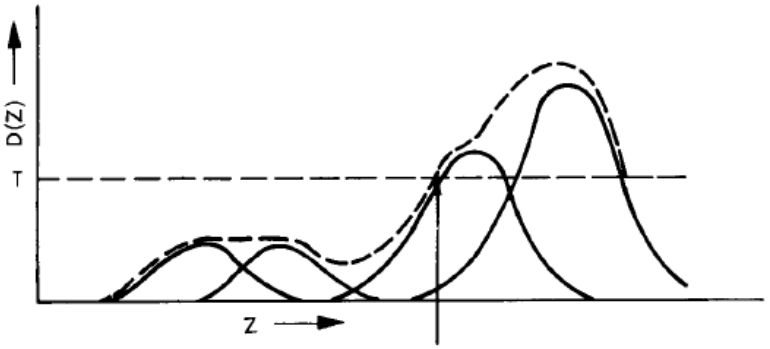
implicit

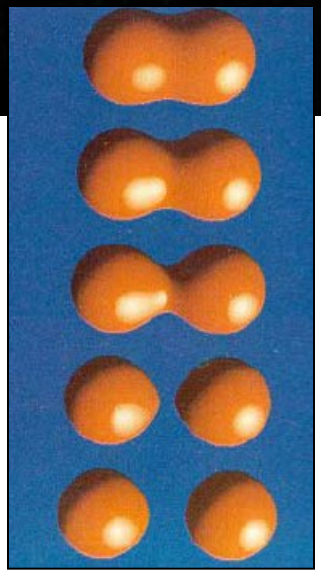
$\{ p \} : p_x^2 + p_y^2 - 1 = 0$

Blinn's Blobbies

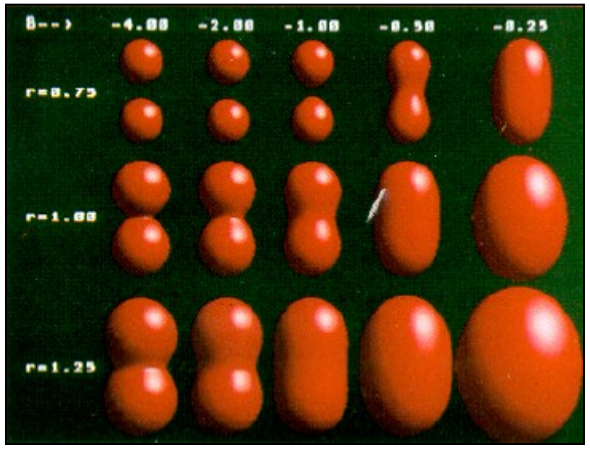
these implicit surfaces are defined by the sum of Gaussian density functions



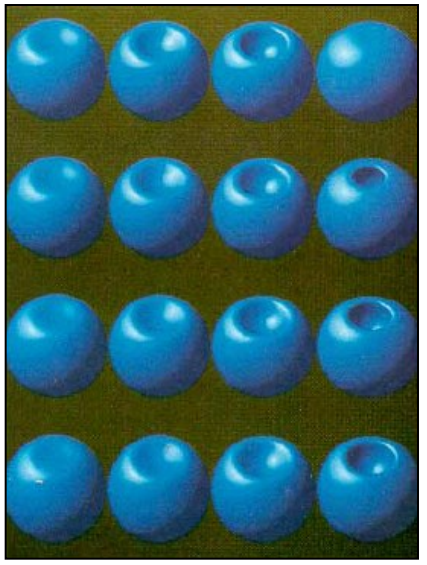




bonds stretch, separate

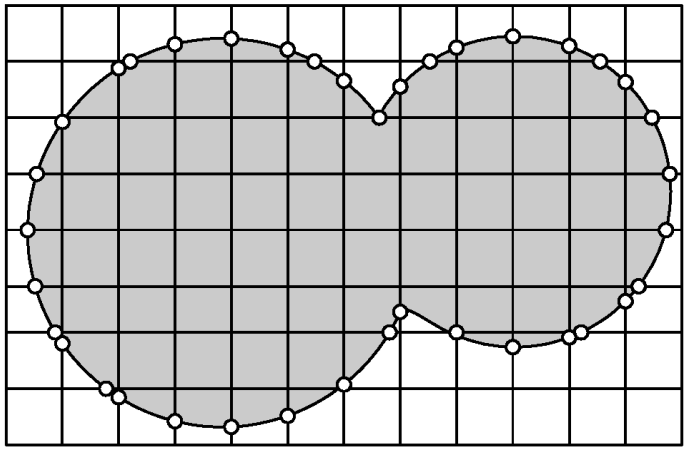


varying radii and blobbiness

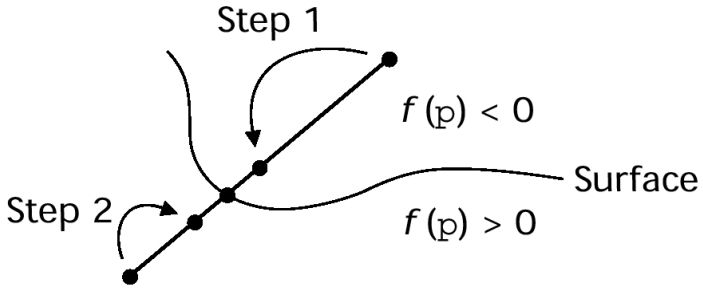


effect of negative density

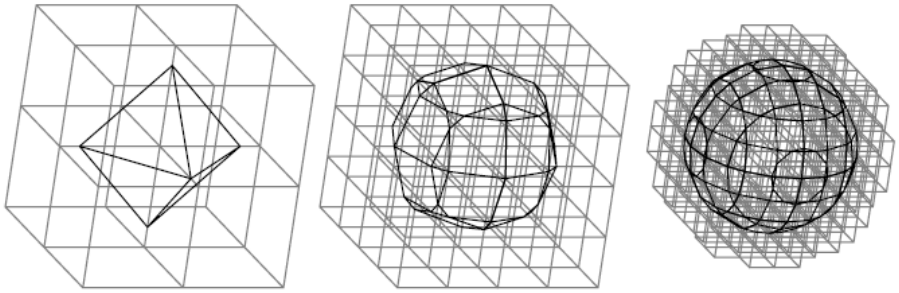
Implicit Surface Polygonization



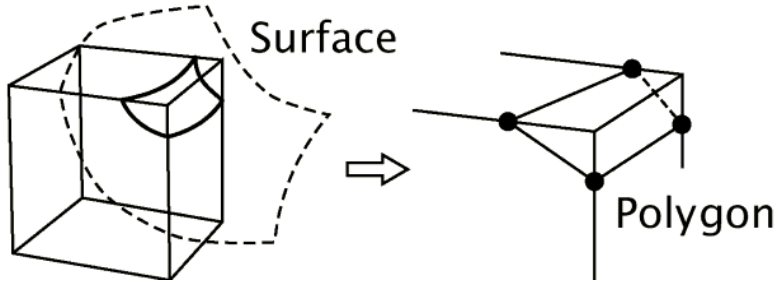
2D cells, intersecting edges, and vertices



edges of terminal nodes that intersect the surface are subdivided to compute a vertex location



3D cells: resolution determines accuracy

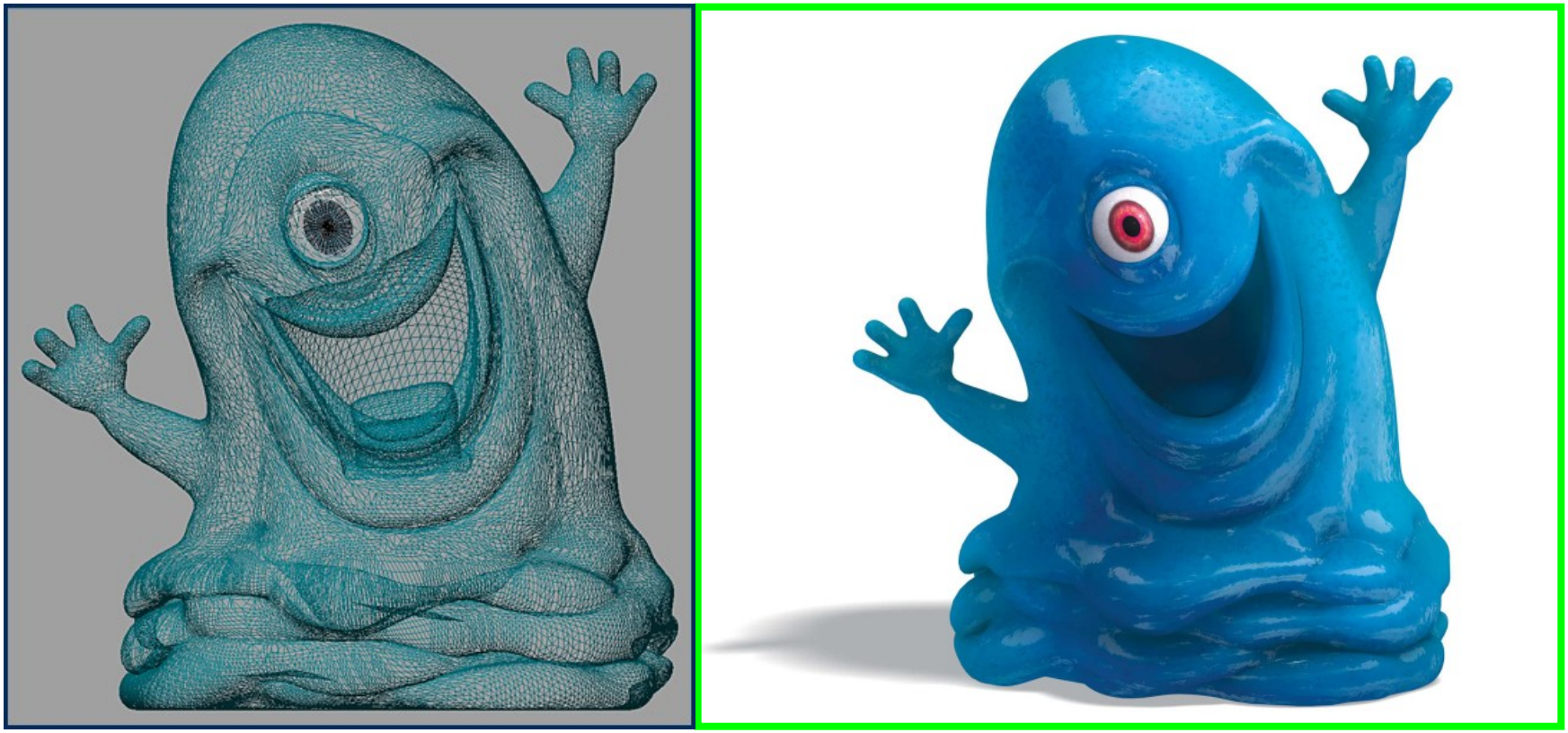


vertices are connected to form a polygon

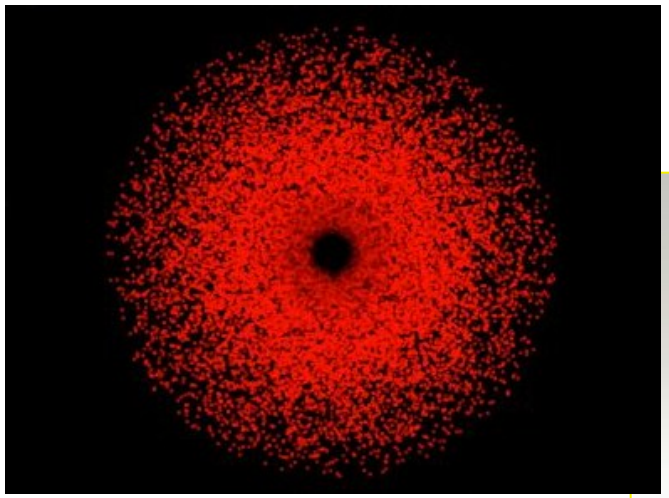
Implicit Surfaces

$$\{ \boldsymbol{p} \} : f(\boldsymbol{p}) = 0$$

(the set of points ***p*** such that ***p*** evaluated by function ***f*** yields 0)



Particle Systems



Earliest Computer Animations

Ed Catmull, 1974

Fred Parke, 1974

Principles of Animation

1. **Squash and Stretch**
defining the rigidity and mass of an object by distorting its shape during an action.

2. **Timing**
spacing actions to define the weight and size of objects and the personality of characters.

3. **Anticipation**
the preparation for an action.

4. **Staging**
presenting an idea so that it is unmistakably clear.

5. **Follow Through & Overlapping Action**
the termination of an action and establishing its relationship to the next action.

6. **Straight Ahead & Pose-To-Pose Action**
the two contrasting approaches to the creation of movement.

7. **Slow In and Out**
the spacing of the in-between frames to achieve subtlety of timing and movement.

8. **Arcs**
the visual path of action for natural movement.

9. **Exaggeration**
accentuating the essence of an idea via the design and the action.

10. **Secondary Action**
the action of an object resulting from another action.

11. **Appeal**
creating a design or an action that the audience enjoys watching.

Slow In and Out

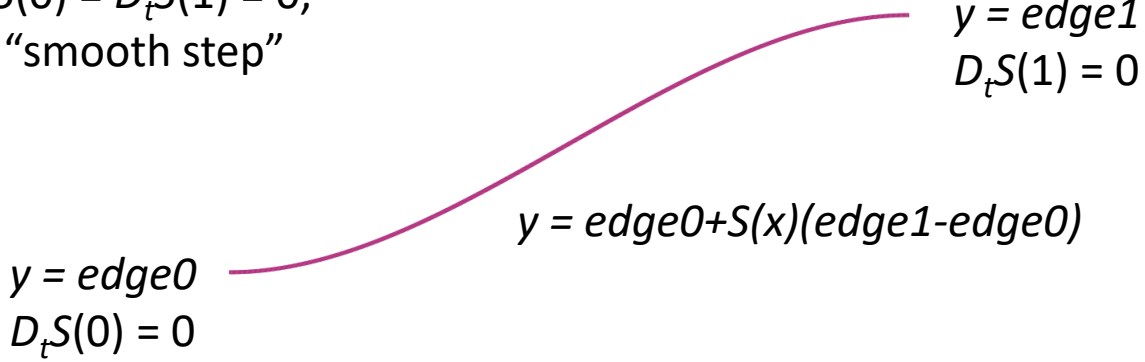
from https://registry.khronos.org/OpenGL-Refpages/gl4/

smoothstep performs smooth Hermite interpolation between 0 and 1 when edge0 < x < edge1. This is useful when a threshold function with a smooth transition is desired. smoothstep ***S*** is equivalent to:

```
t = clamp((x-edge0)/(edge1-edge0), 0, 1);
return t*t*(3-2*t);
```

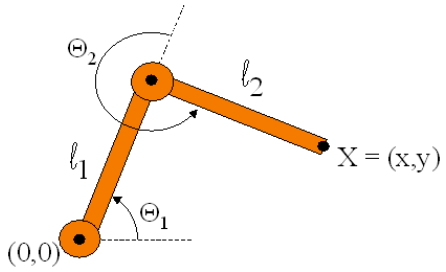
that is, $3t^2-2t^3$

The derivative is $6t-6t^2$
That is, $D_tS(0) = D_tS(1) = 0$,
which is a "smooth step"



Forward Kinematics

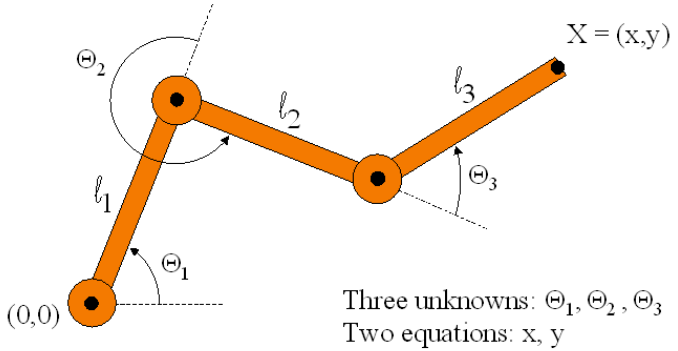
- Animator specifies joint angles: Θ_1 and Θ_2
- Computer finds positions of end-effector: X



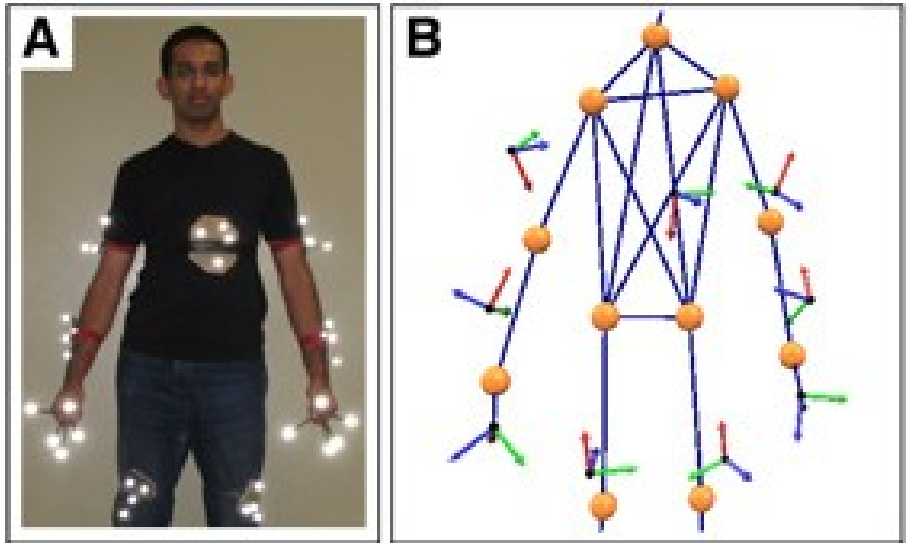
$$X = (l_1 \cos\Theta_1 + l_2 \cos(\Theta_1 + \Theta_2), l_1 \sin\Theta_1 + l_2 \sin(\Theta_1 + \Theta_2))$$

Inverse Kinematics

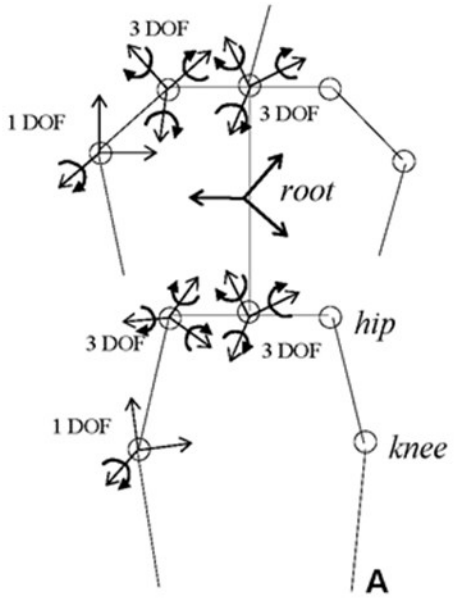
- Problem for more complex structures
  - System of equations is usually under-defined
  - Multiple solutions



Motion Capture

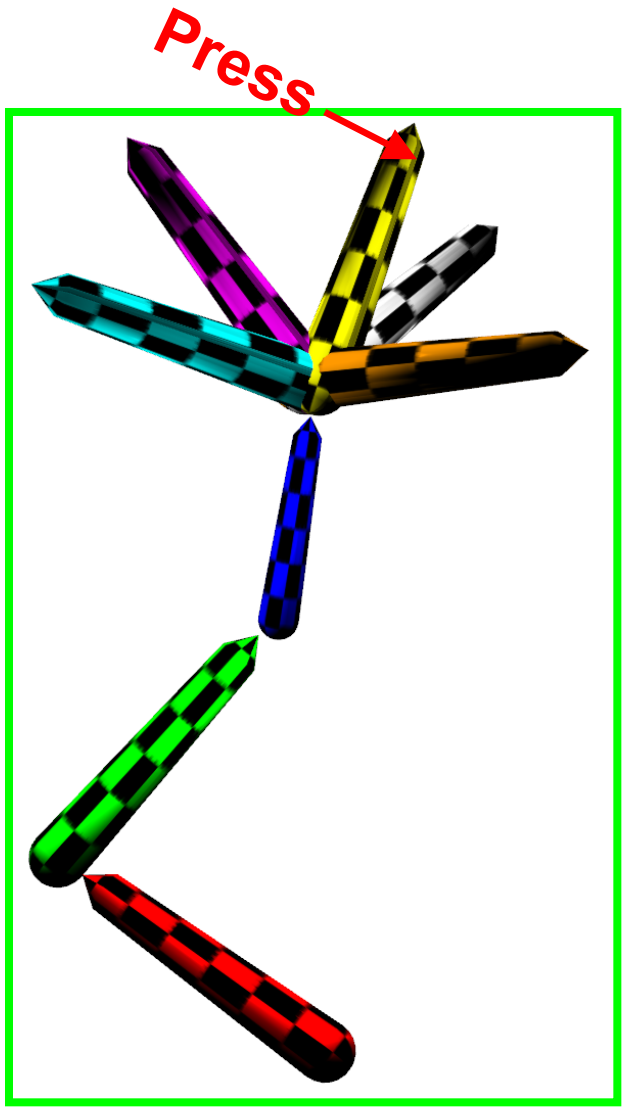


reference frames
(lights in groups of
four) in motion capture



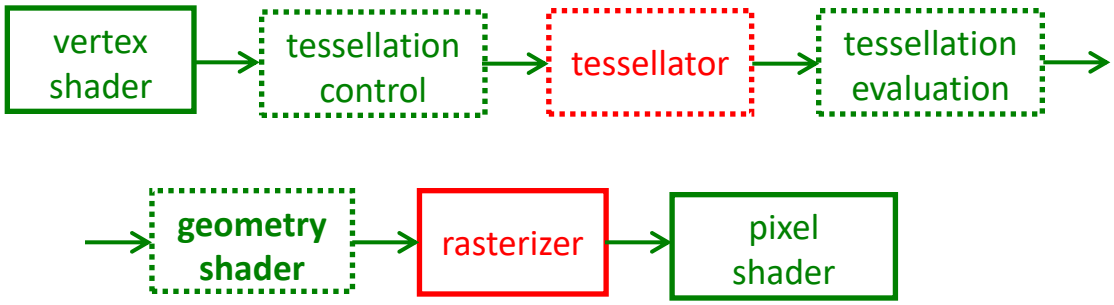
joints with different
degrees-of-freedom

Articulation, Implemented




```
void ApplyTransform(Mesh *m, mat4 xform) {
    m->toWorld = xform*m->toWorld;
    for (size_t i = 0; i < m->children.size(); i++)
        ApplyTransform(m->children[i], xform);
}

void MouseMove(float x, float y) {
  ...
  if (picked == &framer) {
    mat4 mDif = framer.Drag(x, y, camera.modelview, camera.persp);
      // sets pickedMesh->toWorld
    for (size_t i = 0; i < pickedMesh->children.size(); i++)
      ApplyTransform(pickedMesh>children[i], mDif);
  }
  ...
}
```

The Geometry Shader (2009)



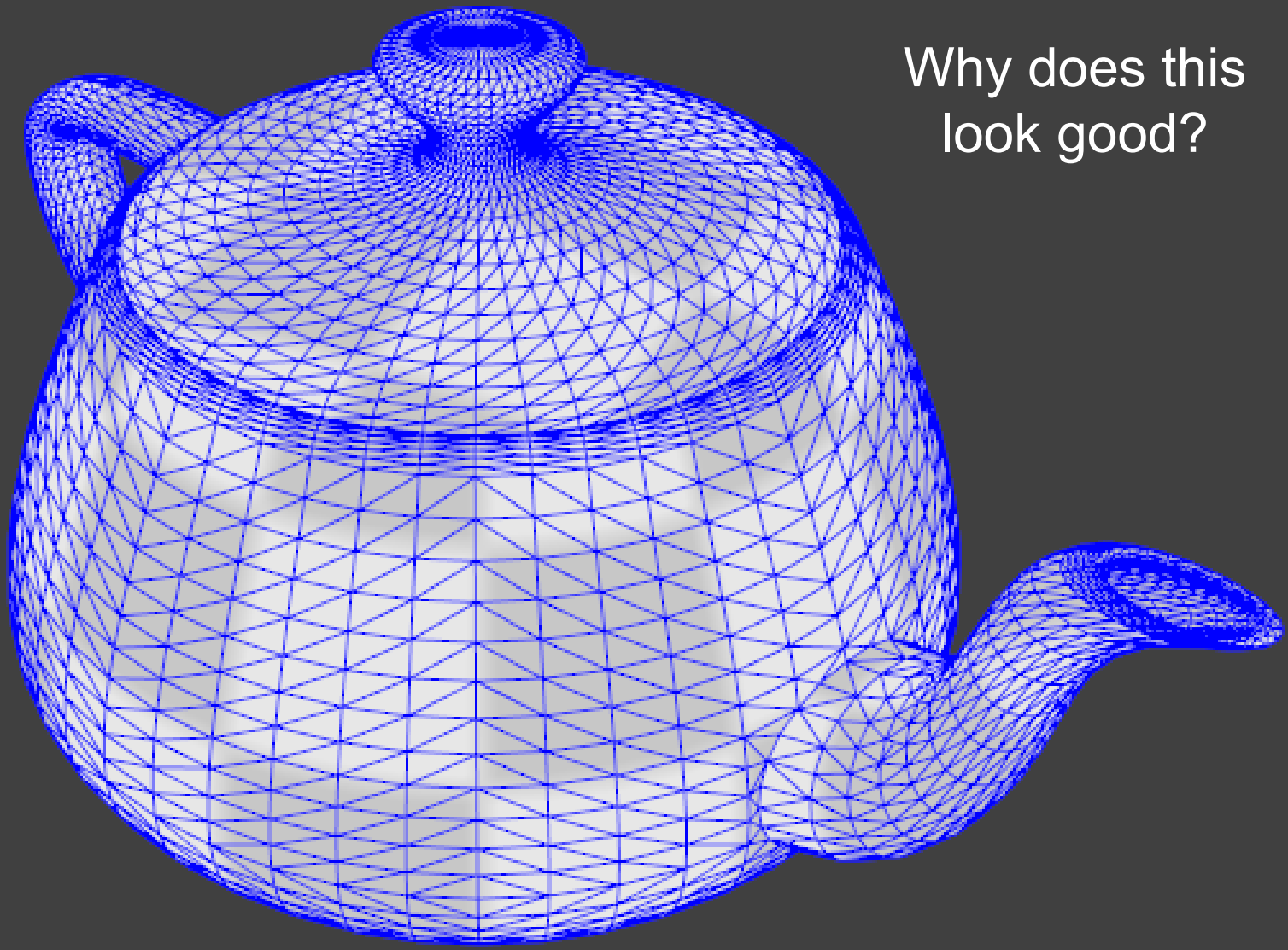
The geometry shader

- can create new vertices, lines, or triangles
- allows a primitive (triangle or quad) to pass additional vertex information to the rasterizer

Unlike the tessellation shader, the geometry shader does not create a tiling

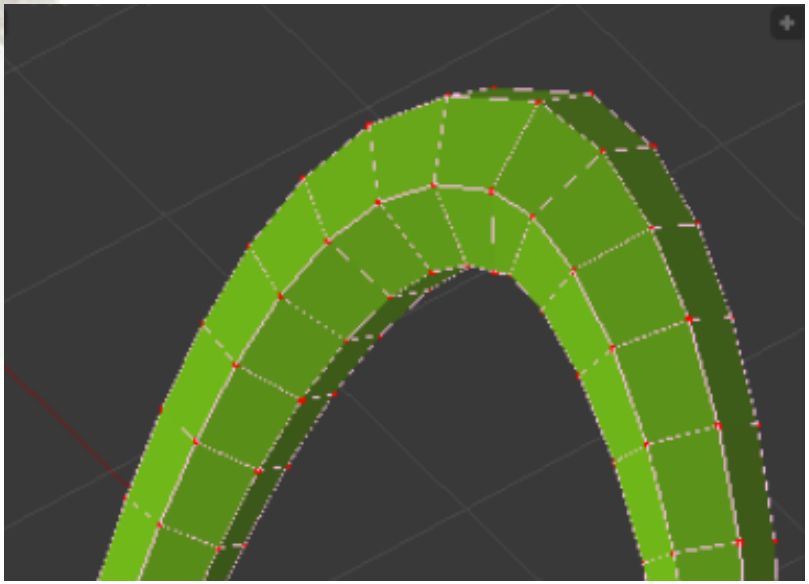
Three examples

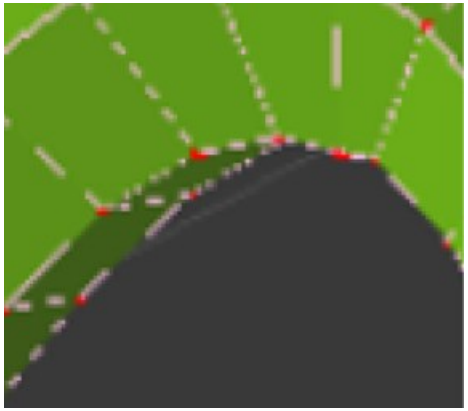
- improved line-on-polygon shading
- GPU debugging
- improved quadrilateral shading



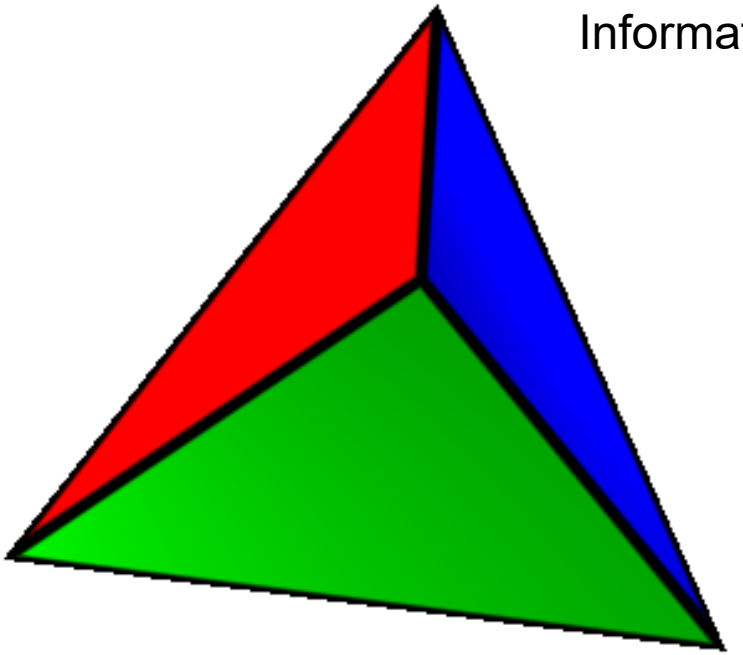
Why does this look good?

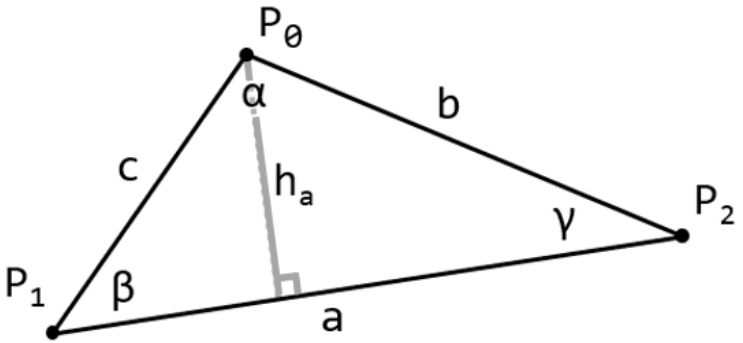
Because Z-Fighting has been replaced

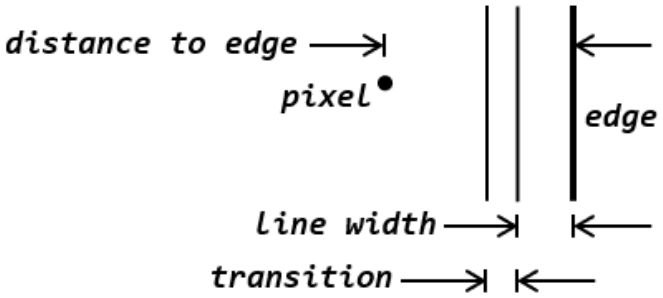


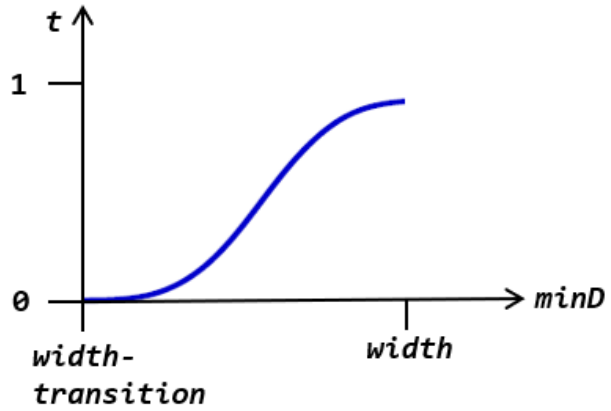


The Geometry Shader Provides Information to the Pixel Shader









When a single primitive (triangle or quad) can access its neighbors . . .

. . . cartoon shading and silhouette edge shading are possible

Quad Split Affects Shading

quad split

Demo

barycentric