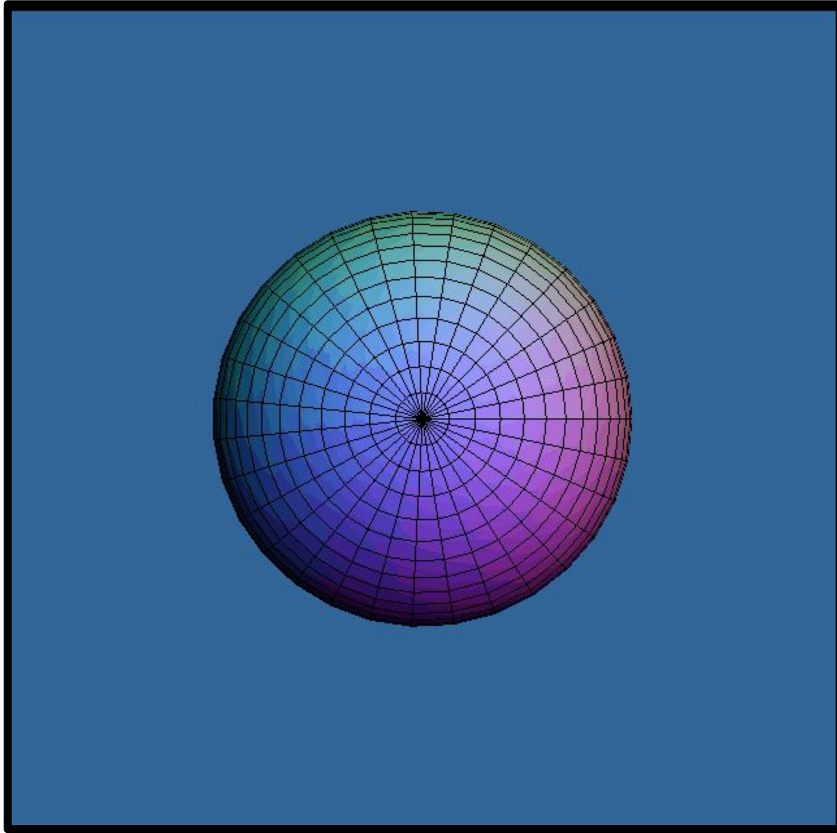
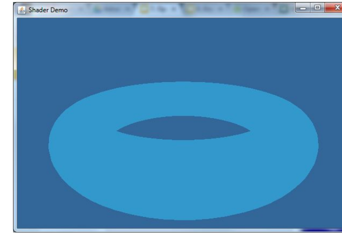
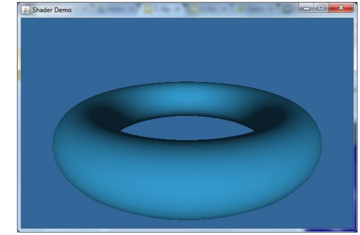
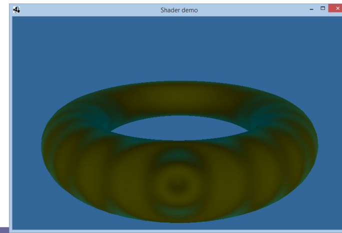
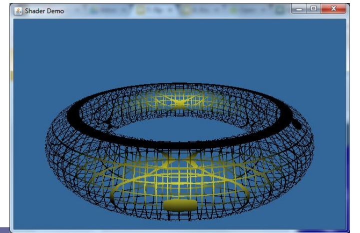




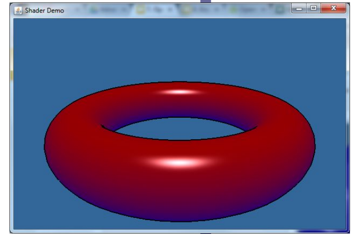


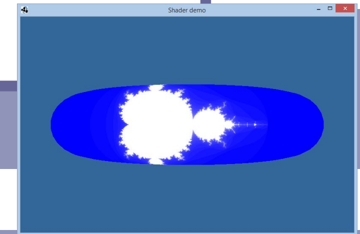
Further Graphics





Advanced Shader Techniques

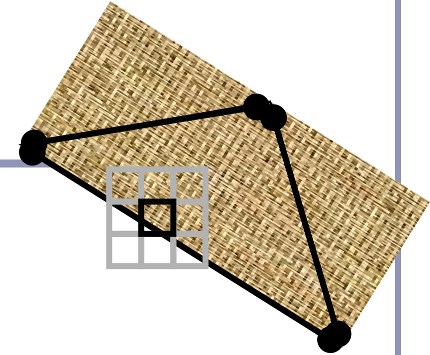




Alex Benton, University of Cambridge – alex@bentonian.com

Supported in part by Google UK, Ltd

Lighting and Shading (a quick refresher)



Recall the classic **lighting equation**:

- $I = k_A + k_D(N \bullet L) + k_S(E \bullet R)^n$

where…

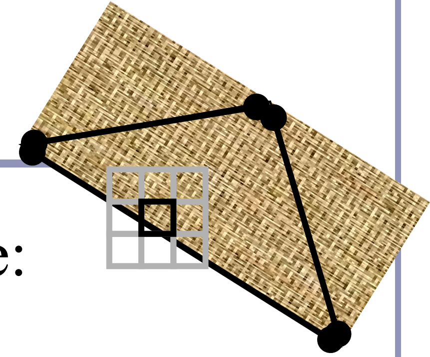
- k_A is the *ambient lighting coefficient* of the object or scene
- $k_D(N \bullet L)$ is the *diffuse component* of surface illumination ('matte')
- $k_S(E \bullet R)^n$ is the *specular component* of surface illumination ('shiny')
  where $R = L - 2(L \bullet N)N$

We compute color by vertex or by polygon fragment:

- Color at the vertex: **Gouraud shading**
- Color at the polygon fragment: **Phong shading**

Vertex shader outputs are interpolated across fragments, so code is clean whether we're interpolating colors or normals.

Shading with shaders



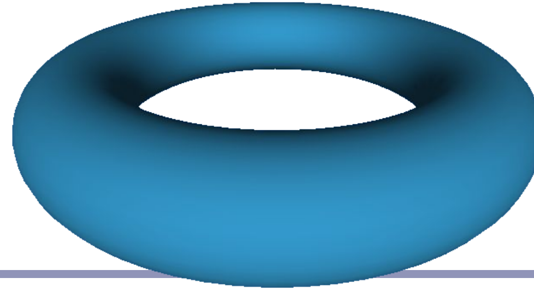
For each vertex our Java code will need to provide:

- Vertex position
- Vertex normal
- [Optional] Vertex color, k_A / k_D / k_S, reflectance, transparency…

We also need global state:

- Camera position and orientation, represented as a transform
- Object position and orientation, to modify the vertex positions above
- A list of light positions, ideally in world coordinates

Shader sample – Gouraud shading

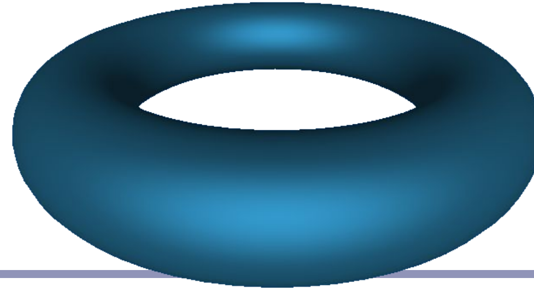


```
#version 330

uniform mat4 modelToScreen;
uniform mat4 modelToWorld;
uniform mat3 normalToWorld;
uniform vec3 lightPosition;

in vec4 v;
in vec3 n;

out vec4 color;

const vec3 purple = vec3(0.2, 0.6, 0.8);

void main() {
  vec3 p = (modelToWorld * v).xyz;
  vec3 n = normalize(normalToWorld * n);
  vec3 l = normalize(lightPosition - p);
  float ambient = 0.2;
  float diffuse = 0.8 * clamp(0, dot(n, l), 1);

  color = vec4(purple
      * (ambient + diffuse), 1.0);
  gl_Position = modelToScreen * v;
}
```

```
#version 330

in vec4 color;

out vec4 fragmentColor;

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

Diffuse lighting
$d = k_D(N \bullet L)$
expressed as a shader

Shader sample – Phong shading



```
#version 330

uniform mat4 modelToScreen;
uniform mat4 modelToWorld;
uniform mat3 normalToWorld;

in vec4 v;
in vec3 n;

out vec3 position;
out vec3 normal;

void main() {
  normal = normalize(
      normalToWorld * n);
  position =
      (modelToWorld * v).xyz;
  gl_Position =
      modelToScreen * v;
}
```

GLSL includes handy helper methods for illumination such as reflect()--perfect for specular highlights.

```
#version 330

uniform vec3 eyePosition;
uniform vec3 lightPosition;

in vec3 position;
in vec3 normal;

out vec4 fragmentColor;

const vec3 purple = vec3(0.2, 0.6, 0.8);

void main() {
  vec3 n = normalize(normal);
  vec3 l = normalize(lightPosition - position);
  vec3 e = normalize(position - eyePosition);
  vec3 r = reflect(l, n);

  float ambient = 0.2;
  float diffuse = 0.4 * clamp(0, dot(n, l), 1);
  float specular = 0.4 *
      pow(clamp(0, dot(e, r), 1), 2);

  fragmentColor = vec4(purple *
      (ambient + diffuse + specular), 1.0);
}
```

$a = k_A$
$d = k_D(N \bullet L)$
$s = k_S(\mathrm{E} \bullet \mathrm{R})^n$

Shader sample – Gooch shading

Gooch shading is an example of *non-realistic rendering*. It was designed by Amy and Bruce Gooch to replace photorealistic lighting with a lighting model that highlights structural and contextual data.

- They use the term of the conventional lighting equation to choose a map between ‘cool’ and ‘warm’ colors.
- This is in contrast to conventional illumination where lighting simply scales the underlying surface color.
- Combined with edge-highlighting through a second renderer pass, this creates 3D models which look like engineering schematics.

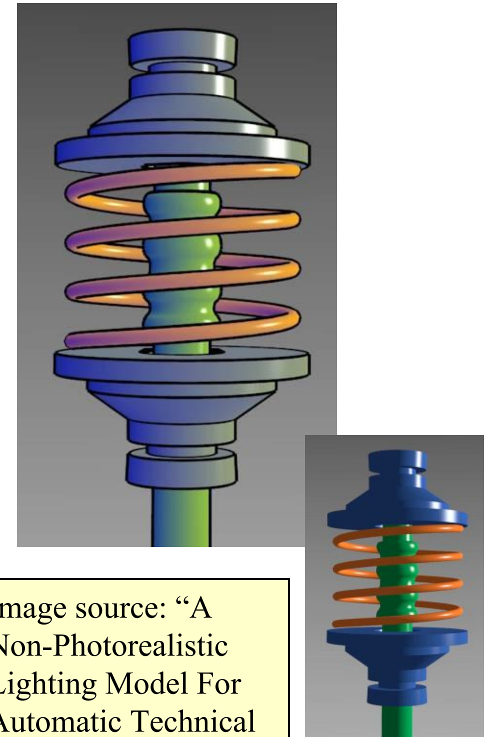


Image source: “A Non-Photorealistic Lighting Model For Automatic Technical Illustration”, Gooch, Gooch, Shirley and Cohen (1998). Compare the Gooch shader, above, to the Phong shader (right).

Shader sample – Gooch shading

```
#version 330

// Original author: Randi Rost
// Copyright (c) 2002-2005 3Dlabs Inc. Ltd.

uniform mat4 modelToCamera;
uniform mat4 modelToScreen;
uniform mat3 normalToCamera;

vec3 LightPosition = vec3(0, 10, 4);

in vec4 vPosition;
in vec3 vNormal;

out float NdotL;
out vec3  ReflectVec;
out vec3  ViewVec;

void main()
{
  vec3 ecPos      = vec3(modelToCamera * vPosition);
  vec3 tnorm      = normalize(normalToCamera * vNormal);
  vec3 lightVec   = normalize(LightPosition - ecPos);
  ReflectVec      = normalize(reflect(-lightVec, tnorm));
  ViewVec         = normalize(-ecPos);
  NdotL           = (dot(lightVec, tnorm) + 1.0) * 0.5;
  gl_Position     = modelToScreen * vPosition;
}
```

```
#version 330

// Original author: Randi Rost
// Copyright (c) 2002-2005 3Dlabs Inc. Ltd.

uniform vec3 vColor;

float DiffuseCool = 0.3;
float DiffuseWarm = 0.3;
vec3 Cool = vec3(0, 0, 0.6);
vec3 Warm = vec3(0.6, 0, 0);

in float NdotL;
in vec3  ReflectVec;
in vec3  ViewVec;

out vec4 result;

void main()
{
  vec3 kcool  = min(Cool + DiffuseCool * vColor, 1.0);
  vec3 kwarm  = min(Warm + DiffuseWarm * vColor, 1.0);
  vec3 kfinal = mix(kcool, kwarm, NdotL);

  vec3 nRefl  = normalize(ReflectVec);
  vec3 nview  = normalize(ViewVec);
  float spec  = pow(max(dot(nRefl, nview), 0.0), 32.0);

  if (gl_FrontFacing) {
    result = vec4(min(kfinal + spec, 1.0), 1.0);
  } else {
    result = vec4(0, 0, 0, 1);
  }
}
```

Shader sample – Gooch shading

In the vertex shader source, notice the use of the built-in ability to distinguish front faces from back faces:

```
if (gl_FrontFacing) {...
```

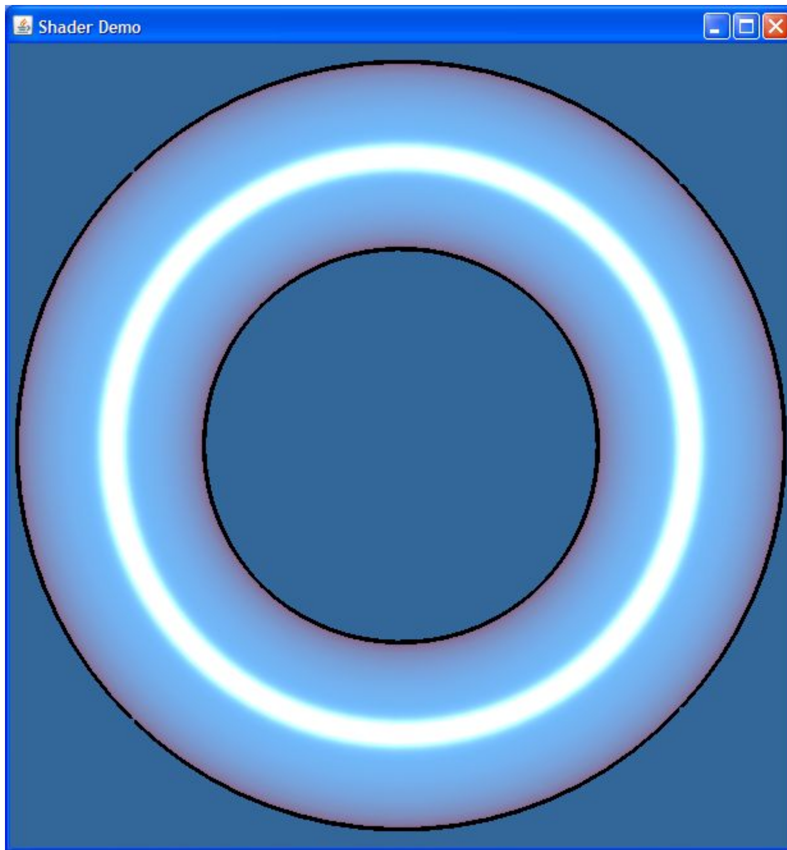
This supports distinguishing front faces (which should be shaded smoothly) from the edges of back faces (which will be drawn in heavy black.)

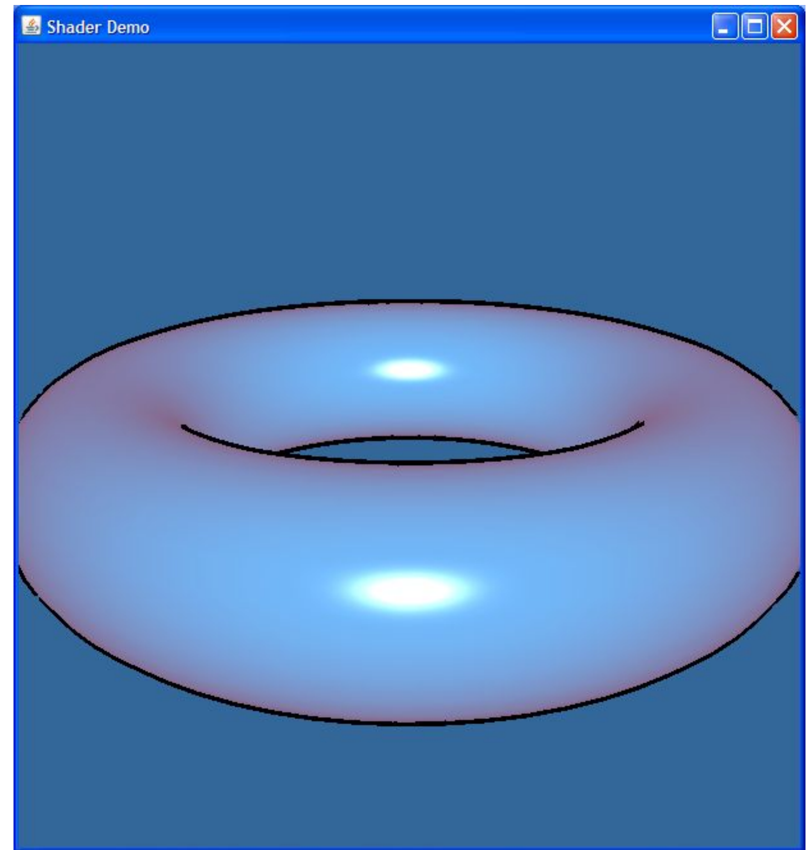
In the fragment shader source, this is used to choose the weighted color by clipping with the `a` component:

```
vec3 kfinal = mix(kcool, kwarm, NdotL);
```

Here `mix()` is a GLSL method which returns the linear interpolation between `kcool` and `kwarm`. The weighting factor is `NdotL`, the lighting value.

Shader sample – Gooch shading

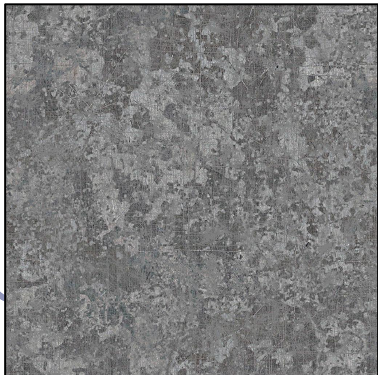


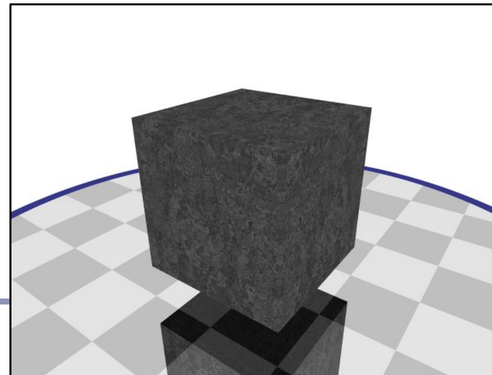
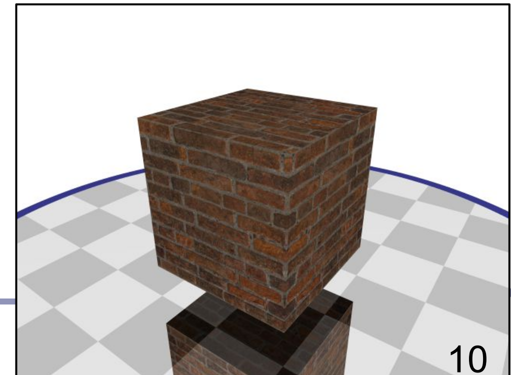


Texture mapping

Real-life objects rarely consist of perfectly smooth, uniformly colored surfaces.

Texture mapping is the art of applying an image to a surface, like a decal. Coordinates on the surface are mapped to coordinates in the texture.



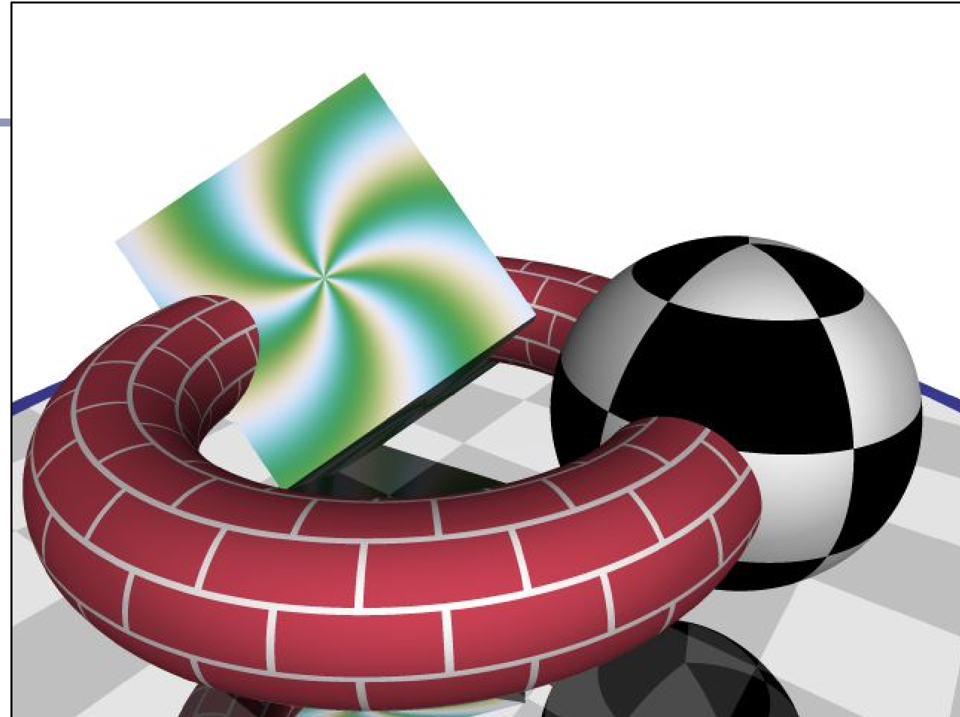



Procedural texture

Instead of relying on discrete pixels, you can get infinitely more precise results with procedurally generated textures.

Procedural textures compute the color directly from the U,V coordinate without an image lookup.

For example, here's the code for the torus' brick pattern (right):

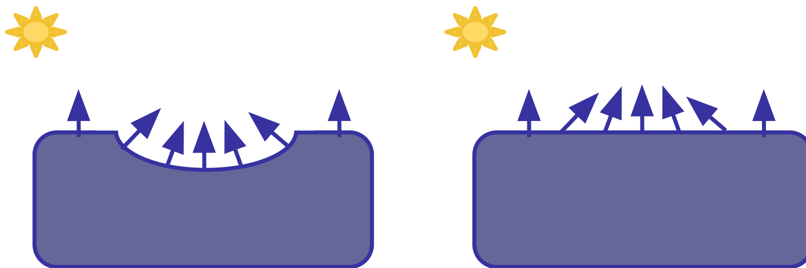


I've cheated slightly and multiplied the u coordinate by 4 to repeat the brick texture four times around the torus.

```
tx = (int) 10 * u
ty = (int) 10 * v
oddity = (tx & 0x01) == (ty & 0x01)
edge = ((10 * u - tx < 0.1) && oddity) || (10 * v - ty < 0.1)
return edge ? WHITE : RED
```

Non-color textures: normal mapping

Normal mapping applies the principles of texture mapping to the surface normal instead of surface color.

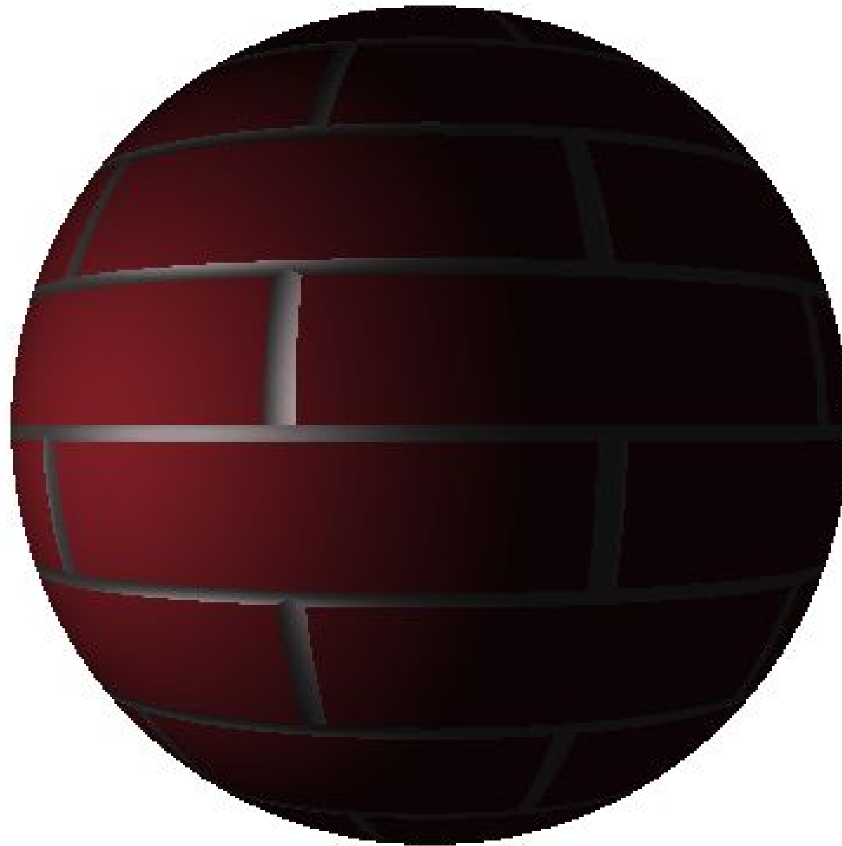


The specular and diffuse shading of the surface varies with the normals in a dent on the surface.

If we duplicate the normals, we don't have to duplicate the dent.

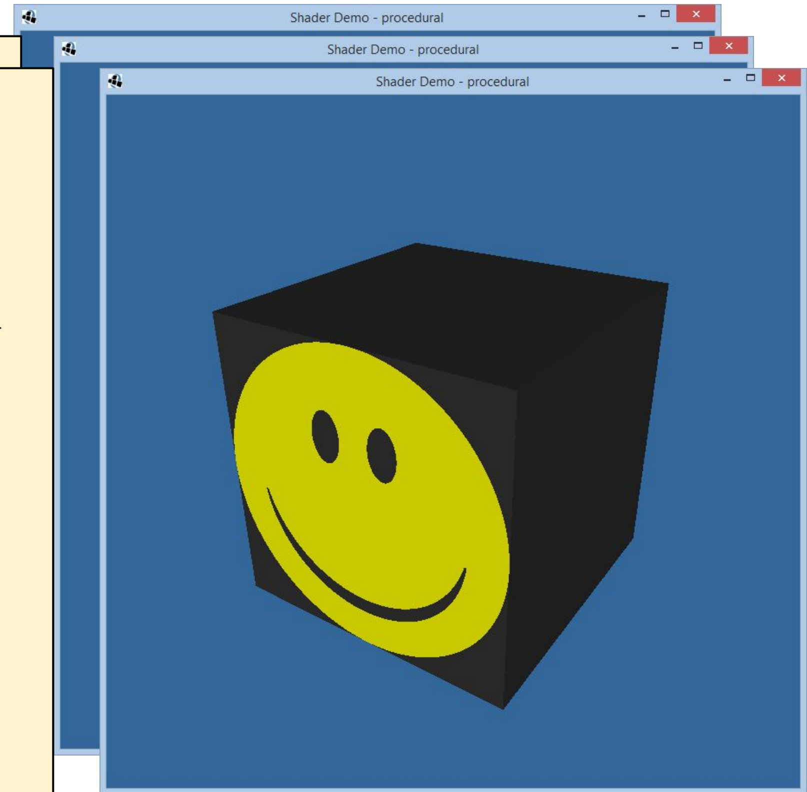
In a sense, the renderer computes a trompe-l'oeuil image on the fly and 'paints' the surface with more detail than is actually present in the geometry.

Non-color textures: normal mapping



Procedural texturing in the fragment shader

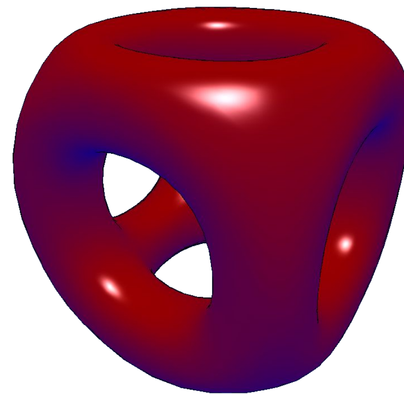
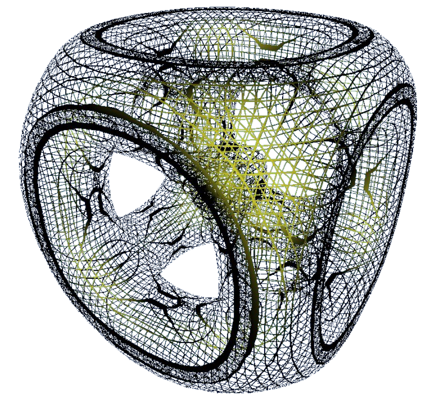
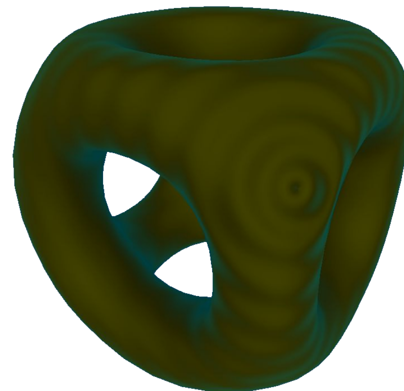
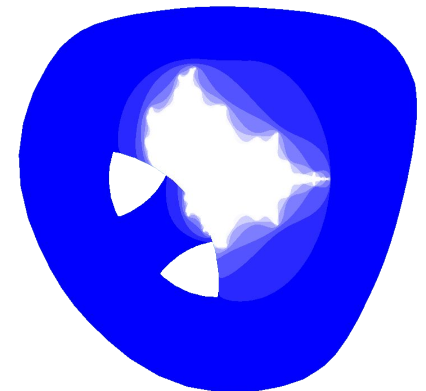
```
// ...
const vec3 CENTER = vec3(0, 0, 1);
const vec3 LEFT_EYE = vec3(-0.2, 0.25, 0);
const vec3 RIGHT_EYE = vec3(0.2, 0.25, 0);
// ...

void main() {
  bool isOutsideFace = (length(position - CENTER) >
1);
  bool isEye = (length(position - LEFT_EYE) < 0.1)
      || (length(position - RIGHT_EYE) < 0.1);
  bool isMouth = (length(position - CENTER) < 0.75)
      && (position.y <= -0.1);

  vec3 color = (isMouth || isEye || isOutsideFace)
      ? BLACK : YELLOW;
  fragmentColor = vec4(color, 1.0);
}
```



(Code truncated for brevity--again, check out the source on github for how I did the curved mouth and oval eyes.)

Advanced surface effects

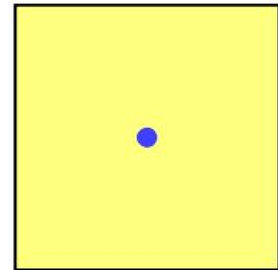
- Ray-tracing, ray-marching!
- Specular highlights
- Non-photorealistic illumination
- Volumetric textures
- Bump-mapping
- Interactive surface effects
- Ray-casting in the shader
- Higher-order math in the shader
- ...much, much more!

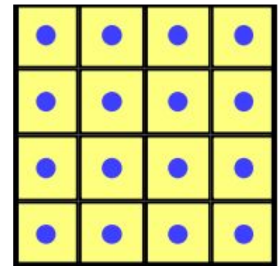





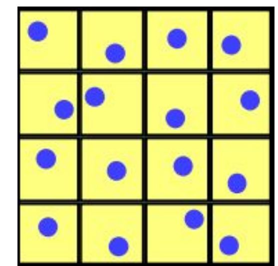
Antialiasing with OpenGL

Antialiasing remains a challenge with hardware-rendered graphics, but image quality can be significantly improved through GPU hardware.

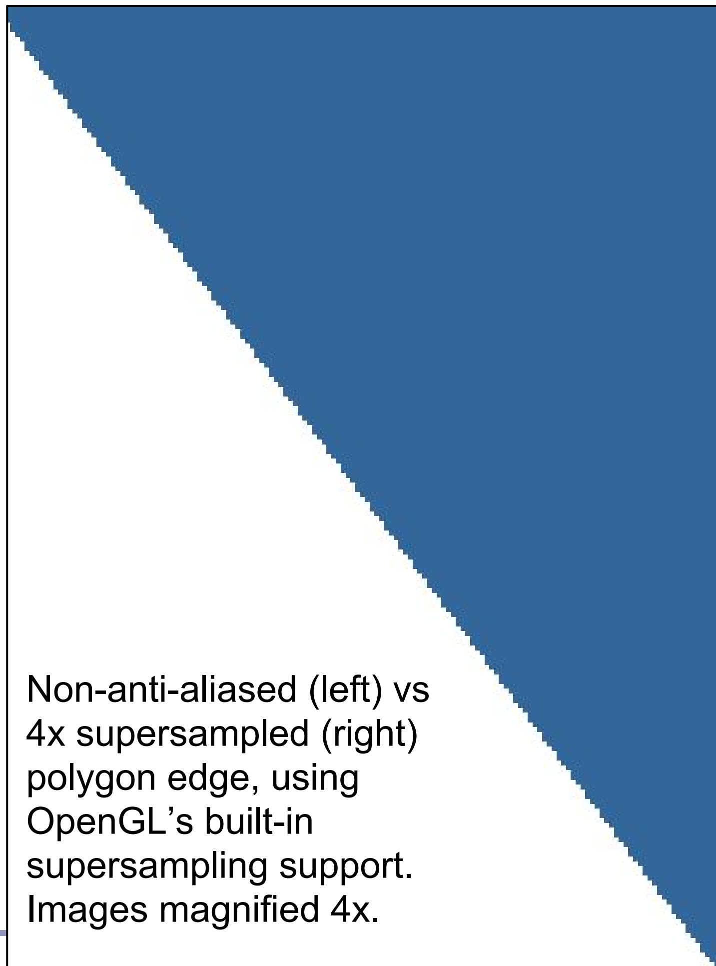
- The simplest form of hardware anti-aliasing is Multi-Sample Anti-Aliasing (*MSAA*).
- "Render everything at higher resolution, then down-sample the image to blur jaggies"
- Enable MSAA in OpenGL with `glfwWindowHint(GLFW_SAMPLES, 4);`



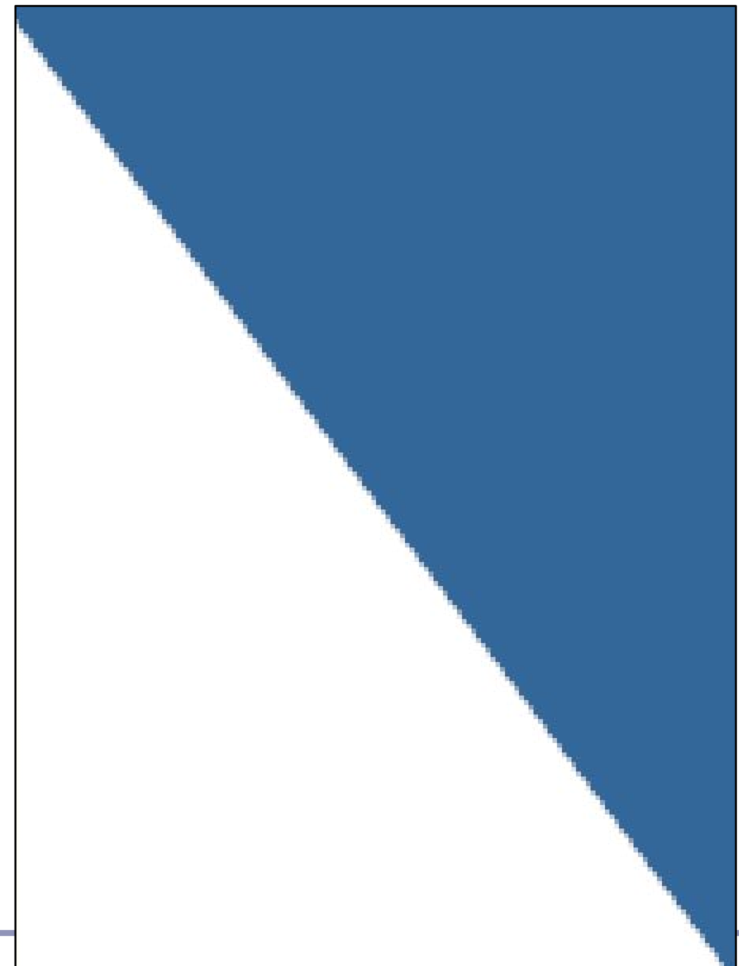




Antialiasing with OpenGL: MSAA



Non-anti-aliased (left) vs 4x supersampled (right) polygon edge, using OpenGL's built-in supersampling support. Images magnified 4x.



Antialiasing on the GPU

MSAA suffers from high memory constraints, and can be very limiting in high-resolution scenarios (high demand for time and texture access bandwidth.)

Eric Chan at MIT described an optimized hardware-based anti-aliasing method in 2004:

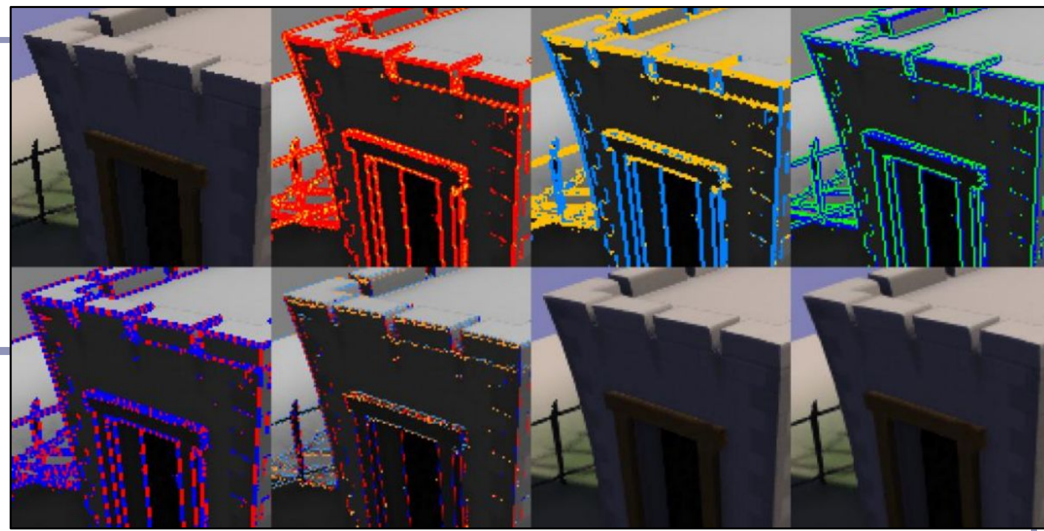
1. Draw the scene normally
2. Draw wide lines at the objects' silhouettes
   a. Use blurring filters and precomputed luminance tables to blur the lines' width
3. Composite the filtered lines into the framebuffer using alpha blending

This approach is great for polygonal models, tougher for effects-heavy visual scenes like video games

+

=

Antialiasing on the GPU



More recently, NVIDIA's *Fast Approximate Anti-Aliasing* ("FXAA") has become popular because it optimizes MSAA's limitations.

Abstract:

1. Use local contrast (pixel-vs-pixel) to find edges (red), especially those subject to aliasing.
2. Map these to horizontal (gold) or vertical (blue) edges.
3. Given edge orientation, the highest contrast pixel pair 90 degrees to the edge is selected (blue/green)
4. Identify edge ends (red/blue)
5. Re-sample at higher resolution along identified edges, using sub-pixel offsets of edge orientations
6. Apply a slight blurring filter based on amount of detected sub-pixel aliasing

Image from
https://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf

Preventing aliasing in texture reads

Antialiasing technique: *adaptive analytic prefiltering*.

- The precision with which an edge is rendered to the screen is dynamically refined based on the rate at which the function defining the edge is changing with respect to the surrounding pixels on the screen.

This is supported in GLSL by the methods `dFdx(F)` and `dFdy(F)`.

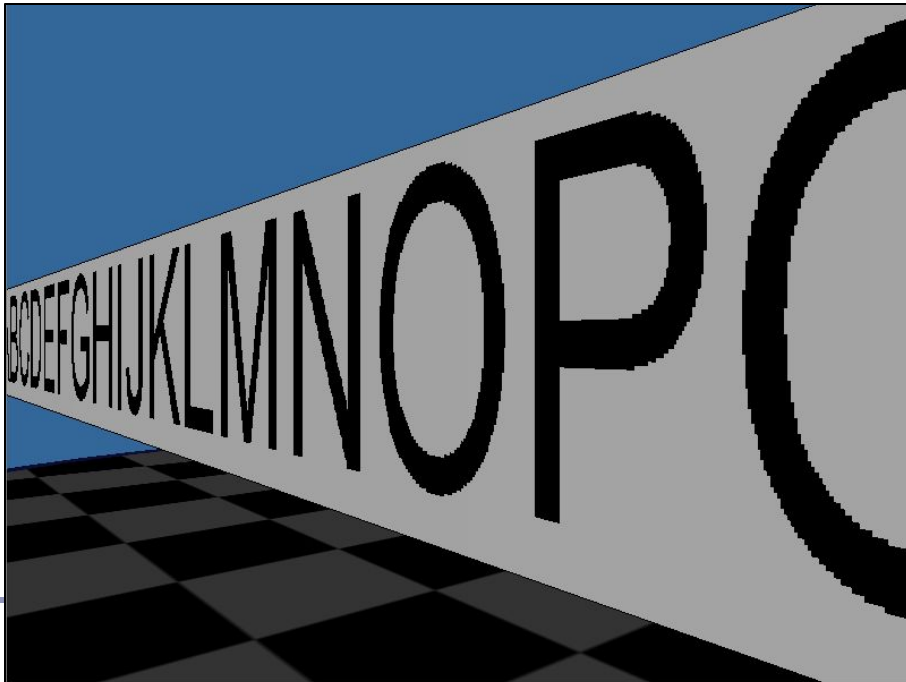
- These methods return the derivative with respect to X and Y, *in screen space,* of some variable F.
- These are commonly used in choosing the filter width for antialiasing procedural textures.

(A) Jagged lines visible in the box function of the procedural stripe texture
(B) Fixed-width averaging blends adjacent samples in texture space; aliasing still occurs at the top, where adjacency in texture space does not align with adjacency in pixel space.
(C) Adaptive analytic prefiltering smoothly samples both areas.
Image source: Figure 17.4, p. 440, *OpenGL Shading Language, Second Edition*, Randi Rost, Addison Wesley, 2006. Digital image scanned by Google Books.
Original image by Bert Freudenberg, University of Magdeburg, 2002.

Antialiasing texture reads with Signed Distance Fields

Conventional anti-aliasing in texture reads can only smooth pixels immediately adjacent to the source values.

Signed distance fields represent monochrome texture data as a distance map instead of as pixels. This allows per-pixel smoothing at multiple distances.

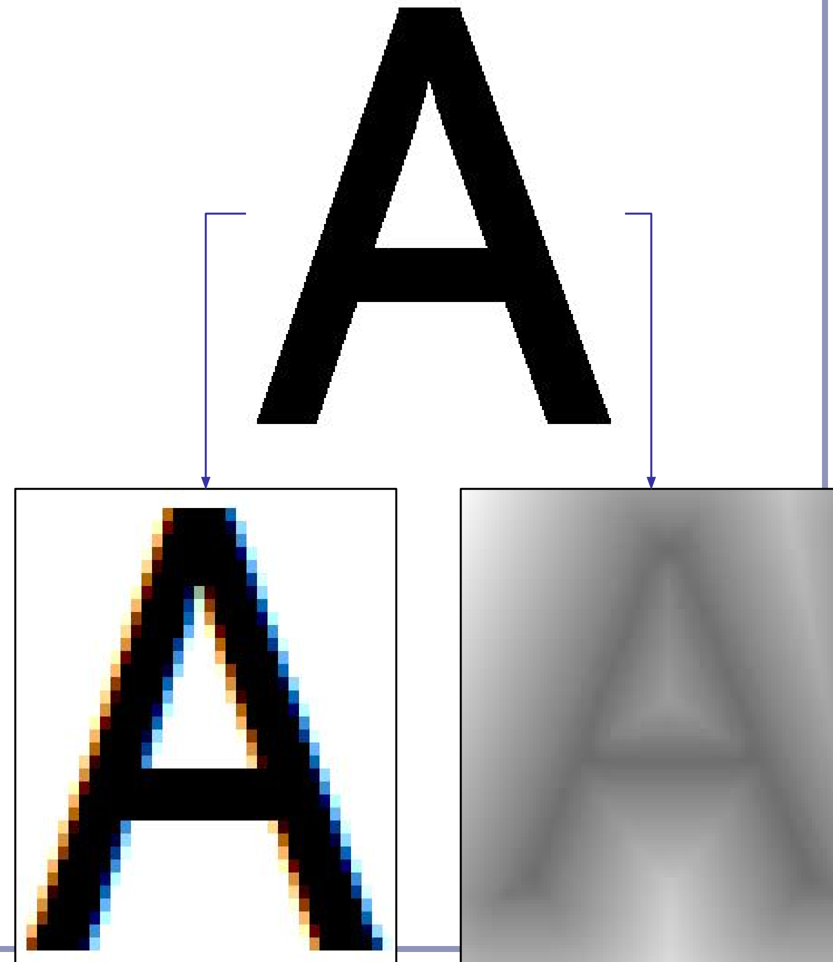


Antialiasing texture reads with Signed Distance Fields

The bitmap becomes a height map.

Each pixel stores the distance to the closest black pixel (if white) or white pixel (if black). Distance from white is negative.

3.6	2.8	2	1	-1
3.1	2.2	1.4	1	-1
2.8	2	1	-1	-1.4
2.2	1.4	1	-1	-2
2	1	-1	-1.4	-2.2
2	1	-1	-2	-2.8



Conventional antialiasing

Signed distance field

Antialiasing texture reads with Signed Distance Fields

Conventional bilinear filtering computes a weighted average of color, but an SDF computes a weighted average of distances.

This means that a small step away from the original values we find smoother, straighter lines where the slope of the isocline is perpendicular to the slope of the source data.

By smoothing the isocline of the distance threshold, we achieve smoother edges and nifty edge effects.

```
low = 0.02;    high = 0.035;
double dist =
  bilinearSample(tex coords);
double t =
  (dist - low) / (high - low);
return (dist < low) ? BLACK
  : (dist > high) ? WHITE
  : BLACK*(1 - t) + WHITE*(t);
```

Adding a second isocline enables colored borders.

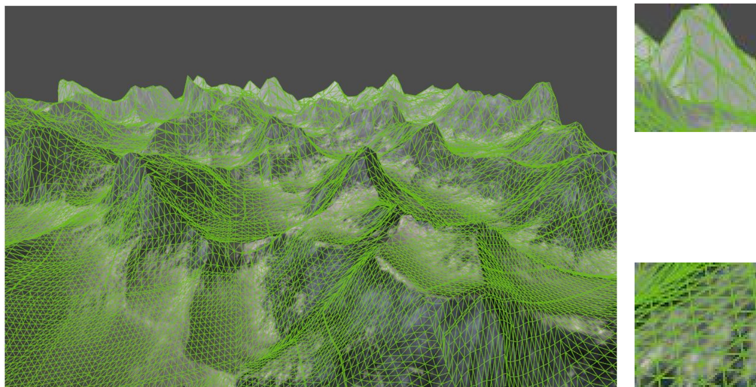
Tessellation shaders

Tesselation is a new shader type introduced in OpenGL 4.x. Tesselation shaders generate new vertices within *patches*, transforming a small number of vertices describing triangles or quads into a large number of vertices which can be positioned individually.

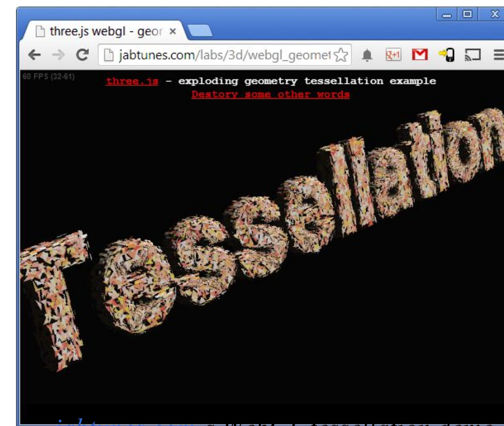
One use of tessellation is in rendering geometry such as game models or terrain with view-dependent *Levels of Detail* ("LOD").

Another is to do with geometry what ray-tracing did with bump-mapping: high-precision realtime geometric deformation.

Note how triangles are small and detailed close to the camera, but become very large and coarse in the distance.



Florian Boesch's LOD terrain demo
http://codeflow.org/entries/2010/nov/07/opengl-4-tessellation/

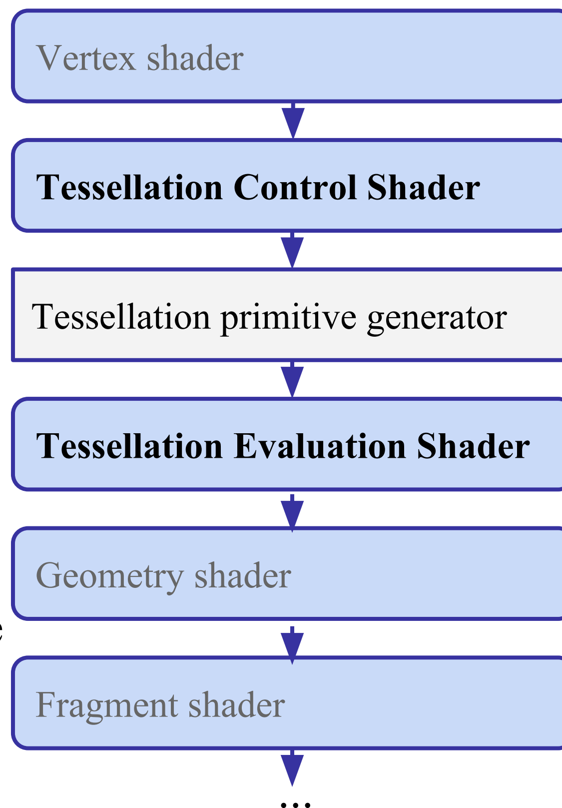


jabtunes.com's WebGL tessellation demo

Tessellation shaders

How it works:

- You tell OpenGL how many vertices a single *patch* will have:
  ```
  glPatchParameteri(GL_PATCH_VERTICES, 4);
  ```
- You tell OpenGL to render your patches:
  ```
  glDrawArrays(GL_PATCHES, first, numVerts);
  ```
- The *Tessellation Control Shader* specifies output parameters defining how a patch is split up: `gl_TessLevelOuter[]` and `gl_TessLevelInner[]`. These control the number of vertices per primitive edge and the number of nested inner levels, respectively.

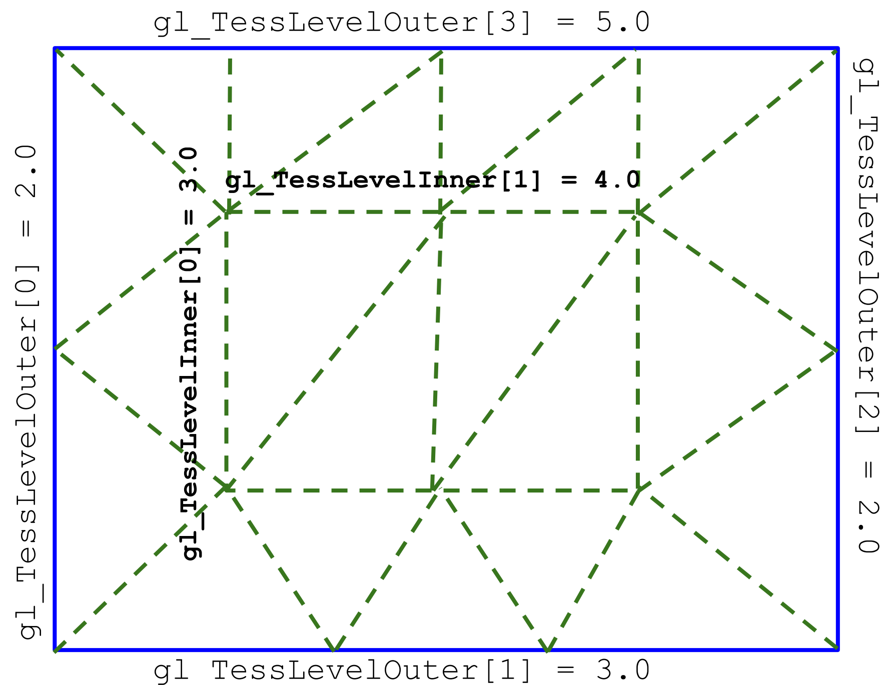


Tessellation shaders

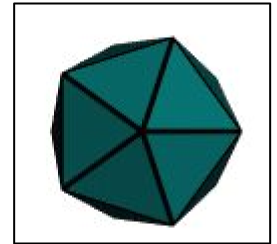
The *tessellation primitive generator* generates new vertices along the outer edge and inside the patch, as specified by `gl_TessLevelOuter[]` and `gl_TessLevelInner[]`.

Each field is an array. Within the array, each value sets the number of intervals to generate during subprimitive generation.

Triangles are indexed similarly, but only use the first three `Outer` and the first `Inner` array field.



Tessellation shaders



The generated vertices are then passed to the *Tesselation Evaluation Shader*, which can update vertex position, color, normal, and all other per-vertex data.

Ultimately the complete set of new vertices is passed to the geometry and fragment shaders.

Inner = 1
Inner = 2
Inner = 3
Inner = 4
Outer = 1
Outer = 2
Outer = 3
Outer = 4

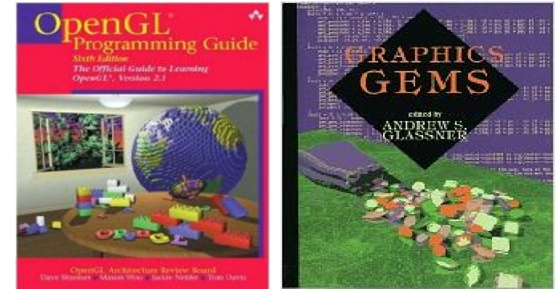
Image credit: Philip Rideout
http://prideout.net/blog/?p=48

CPU vs GPU – an object demonstration

Redux: http://www.youtube.com/watch?v=fKK933KK6Gg

"NVIDIA: Mythbusters - CPU vs GPU"
https://www.youtube.com/watch?v=-P28LKWTzrI

Recommended reading



Course source code on Github -- many demos
(https://github.com/AlexBenton/AdvancedGraphics)

The OpenGL Programming Guide (2013), by Shreiner, Sellers, Kessenich and Licea-Kane

Some also favor *The OpenGL Superbible* for code samples and demos

There's also an OpenGL-ES reference, same series

OpenGL Insights (2012), by Cozzi and Riccio

OpenGL Shading Language (2009), by Rost, Licea-Kane, Ginsburg et al

The *Graphics Gems* series from Glassner

Anti-Aliasing:

https://people.csail.mit.edu/ericchan/articles/prefilter/

https://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf

http://iryoku.com/aacourse/downloads/09-FXAA-3.11-in-15-Slides.pdf