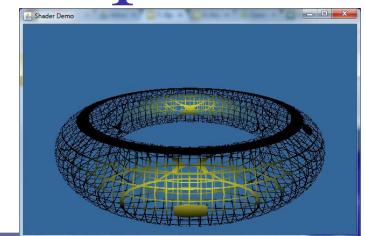
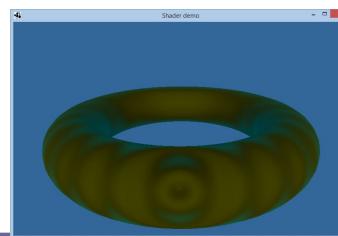
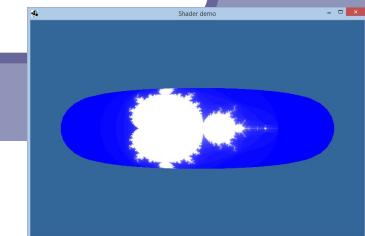
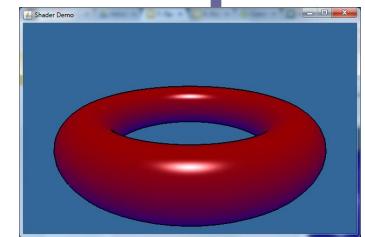


Advanced Graphics



OpenGL and Shaders I



3D technologies today

Java



- Common, re-usable language; well-designed
- Steadily increasing popularity in industry
- Weak but evolving 3D support

C++



- Long-established language
- Long history with OpenGL
- Long history with DirectX
- Losing popularity in some fields (finance, web) but still strong in others (games, medical)

JavaScript

- WebGL is surprisingly popular

OpenGL



- Open source with many implementations
- Well-designed, old, and still evolving
- Fairly cross-platform

DirectX/Direct3d (Microsoft)



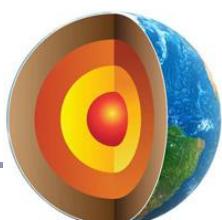
- Microsoft™ only
- Dependable updates

Mantle (AMD)

- Targeted at game developers
- AMD-specific

Higher-level commercial libraries

- RenderMan
- AutoDesks / SoftImage





OpenGL

OpenGL is...

- Hardware-independent
- Operating system independent
- Vendor neutral

On many platforms

- Great support on Windows, Mac, linux, etc
- Support for mobile devices with OpenGL ES
 - Android, iOS (but not Windows Phone)
 - Android Wear watches!
- Web support with WebGL

A state-based renderer

- many settings are configured before passing in data; rendering behavior is modified by existing state

Accelerates common 3D graphics operations

- Clipping (for primitives)
- Hidden-surface removal (Z-buffering)
- Texturing, alpha blending NURBS and other advanced primitives (GLUT)



Mobile GPUs

- OpenGL ES 1.0-3.2
 - A stripped-down version of OpenGL
 - Removes functionality that is not strictly necessary on mobile devices (like recursion!)
- Devices
 - iOS: iPad, iPhone, iPod Touch
 - Android phones
 - PlayStation 3, Nintendo 3DS, and more



OpenGL ES 2.0 rendering (iOS)



WebGL

- JavaScript library for 3D rendering in a web browser
 - Based on OpenGL ES 2.0
 - Many supporting JS libraries
 - Even gwt, angular, dart...
- Most modern browsers support WebGL, even mobile browsers
 - Enables in-browser 3D games
 - Enables realtime experimentation with glsl shader code



Samples from Shadertoy.com





Vulkan

Vulkan is the next generation of OpenGL: a cross-platform open standard aimed at pure performance on modern hardware

Compared to OpenGL, Vulkan--

- Reduces CPU load
- Has better support of multi-CPU core architectures
- Gives finer control of the GPU

--but--

- Drawing a few primitives can take 1000s of lines of code
- Intended for game engines and code that must be very well optimized



The Talos Principle running on Vulkan (via www.geforce.com)

OpenGL in Java - choices

JOGL: “Java bindings for OpenGL”

jogamp.org/jogl

JOGL apps can be deployed as applications or as *applets*, making it suitable for educational web demos and cross-platform applications.

- If the user has installed the latest Java, of course.
- And if you jump through Oracle’s authentication hoops.
- And... let’s be honest, 1998 called, it wants its applets back.

LWJGL: “Lightweight Java Games Library”

www.lwjgl.org

LWJGL is targeted at game developers, so it’s got a solid threading model and good support for new input methods like joysticks, gaming mice, and the Oculus Rift.



*JOGL shaders in action.
Image from Wikipedia*

OpenGL architecture

The CPU (your processor and friend) delivers data to the GPU (Graphical Processing Unit).

- The GPU takes in streams of vertices, colors, texture coordinates and other data; constructs polygons and other primitives; then uses *shaders* to draw the primitives to the screen pixel-by-pixel.
- The GPU processes the vertices according to the *state* set by the CPU; for example, “every trio of vertices describes a triangle”.

This process is called the *rendering pipeline*. Implementing the rendering pipeline is a joint effort between you and the GPU.

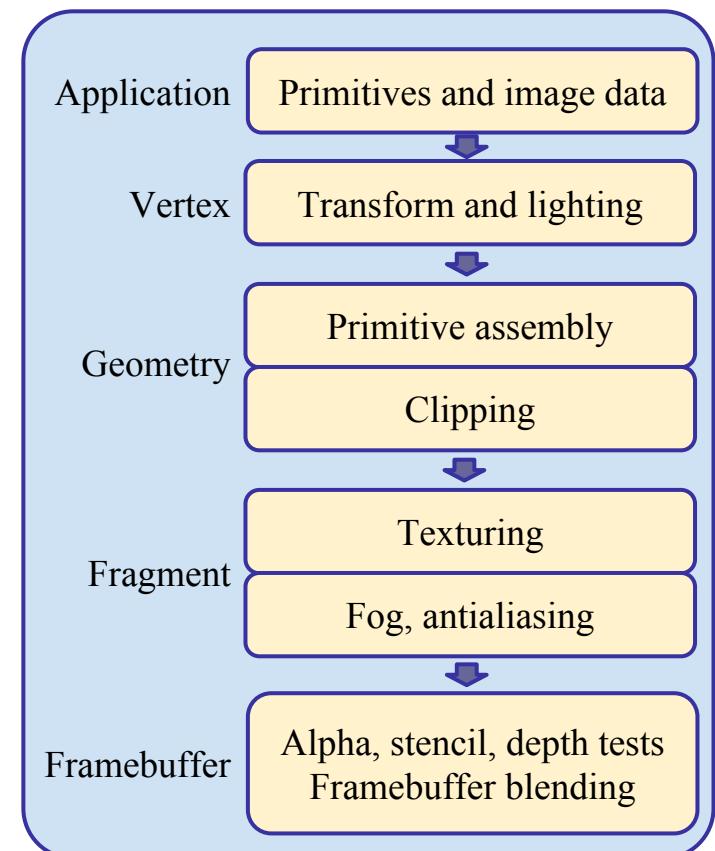
You’ll write shaders in the OpenGL shader language, GLSL.

You’ll write *vertex* and *fragment* shaders. (And maybe others.)

The OpenGL rendering pipeline

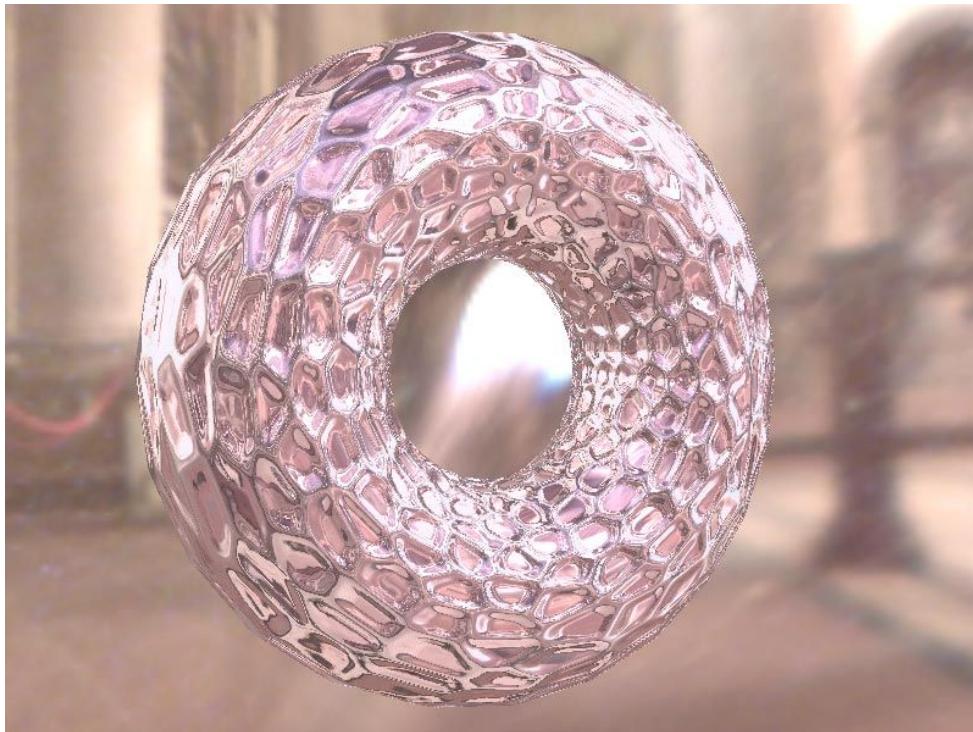
An OpenGL application assembles sets of *primitives*, *transforms* and *image data*, which it passes to OpenGL's GLSL shaders.

- *Vertex shaders* process every vertex in the primitives, computing info such as position of each one.
- *Fragment shaders* compute the color of every fragment of every pixel covered by every primitive.



The OpenGL rendering pipeline
(a massively simplified view)

Shader gallery I

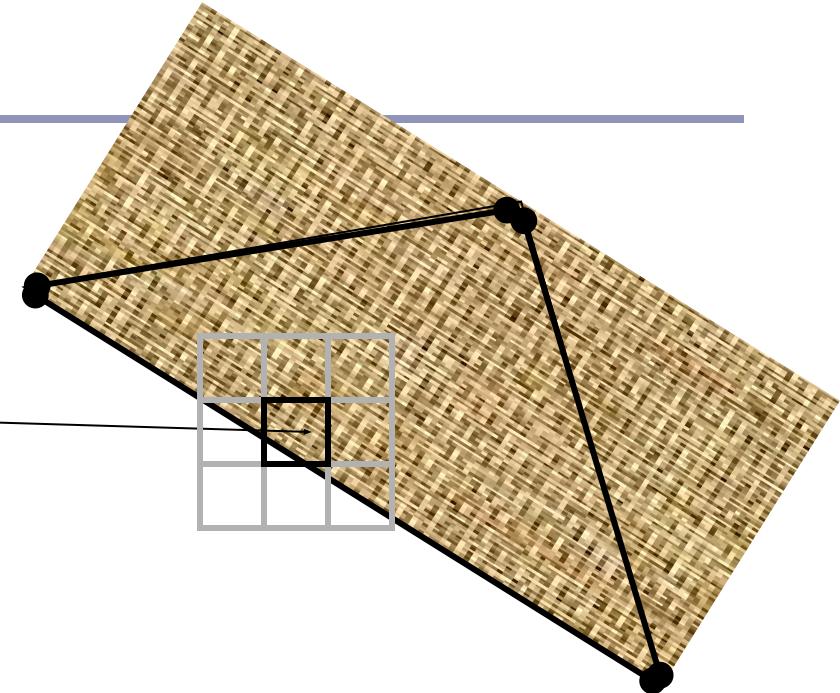


Above: Demo of Microsoft's XNA game platform
Right: Product demos by nvidia (top) and ATI (bottom)



OpenGL: Shaders

OpenGL shaders give the user control over each *vertex* and each *fragment* (each pixel or partial pixel) interpolated between vertices.



After vertices are processed, polygons are *rasterized*. During rasterization, values like position, color, depth, and others are interpolated across the polygon. The interpolated values are passed to each pixel fragment.

Think parallel

Shaders are compiled from within your code

- They used to be written in assembler
- Today they're written in high-level languages
- Vulkan's SPIR-V lets developers code in high-level GLSL but tune at the machine code level

GPUs typically have multiple processing units

That means that multiple shaders execute in parallel

- We're moving away from the purely-linear flow of early “C” programming models

Shader example one – ambient lighting

```
#version 330

uniform mat4 mvp;

in vec4 vPos;

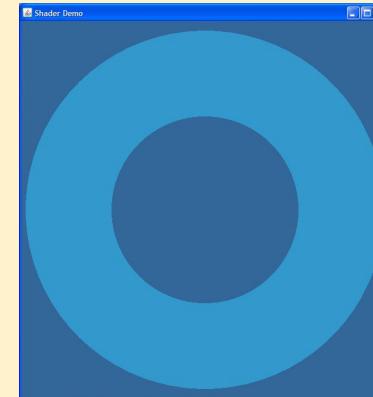
void main() {
    gl_Position = mvp * vPos;
}
```

// Vertex Shader

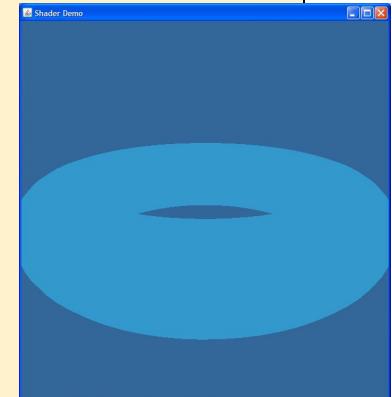
```
#version 330

out vec4 fragmentColor;

void main() {
    fragmentColor =
        vec4(0.2, 0.6, 0.8, 1);
}
```



// Fragment Shader



GLSL

Notice the C-style syntax

```
void main() { ... }
```

The vertex shader uses two inputs, one four-element `vec4` and one four-by-four `mat4` matrix; and one standard output, `gl_Position`.

The line

```
gl_Position = mvp * vPos;
```

applies our model-view-projection matrix to calculate the correct vertex position in perspective coordinates.

The fragment shader implements basic *ambient lighting* by setting its one output, `fragmentColor`, to a constant.

GLSL

The language design in GLSL is strongly based on ANSI C,
with some C++ added.

- There is a preprocessor--#define, etc
- Basic types: int, float, bool
- No double-precision float
- Vectors and matrices are standard: vec2, mat2 = 2x2; vec3, mat3 = 3x3; vec4, mat4 = 4x4
- Texture samplers: sampler1D, sampler2D, etc are used to sample multidimensional textures
- New instances are built with constructors, a la C++
- Functions can be declared before they are defined, and operator overloading is supported.

GLSL

Some differences from C/C++/Java:

- No pointers, strings, chars; no unions, enums; no bytes, shorts, longs; no unsigned. No switch() statements.
- There is no implicit casting (type promotion):

```
float foo = 1;
```

fails because you can't implicitly cast **int** to **float**.

- Explicit type casts are done by constructor:

```
vec3 foo = vec3(1.0, 2.0, 3.0);
```

```
vec2 bar = vec2(foo); // Drops foo.z
```

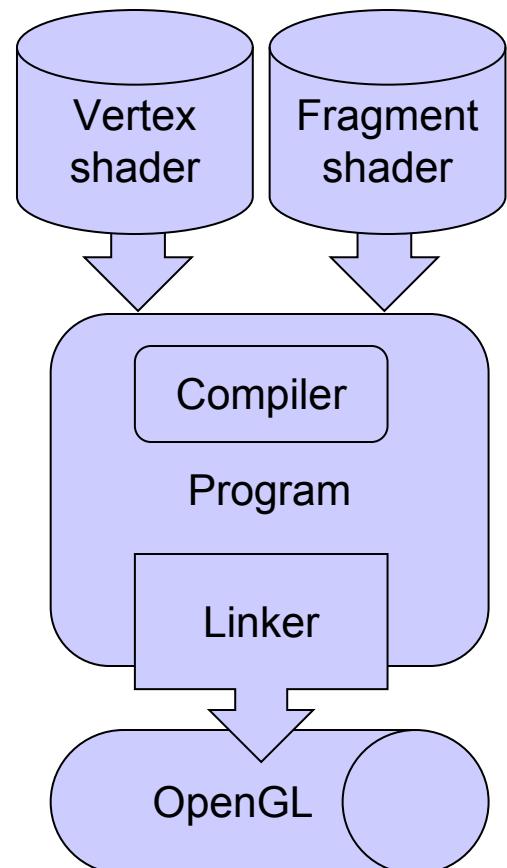
Function parameters are labeled as **in**, **out**, or **uniform**.

- Functions are called by *value-return*, meaning that values are copied into and out of parameters at the start and end of calls.

OpenGL / GLSL API - setup

To install and use a shader in OpenGL:

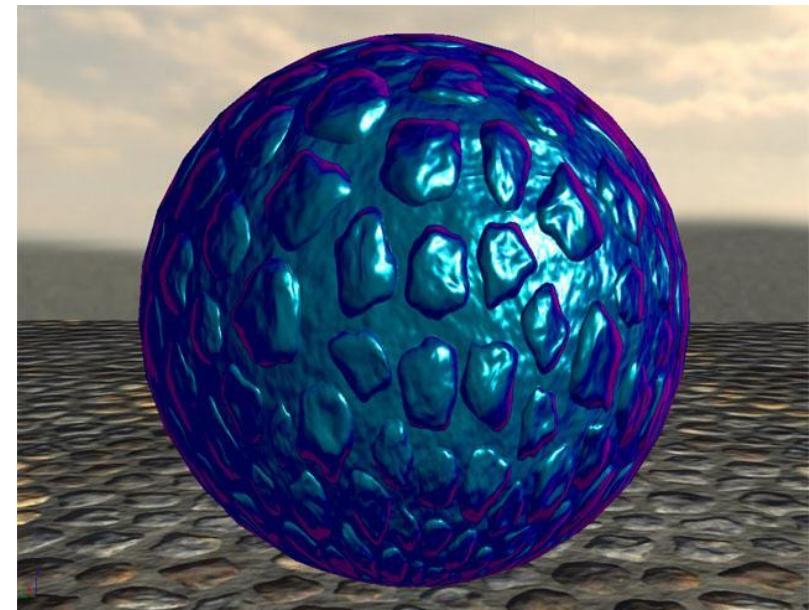
1. Create one or more empty *shader objects* with `glCreateShader`.
2. Load source code, in text, into the shader with `glShaderSource`.
3. Compile the shader with `glCompileShader`.
4. Create an empty *program object* with `glCreateProgram`.
5. Bind your shaders to the program with `glAttachShader`.
6. Link the program (ahh, the ghost of C!) with `glLinkProgram`.
7. Activate your program with `glUseProgram`.



Shader gallery II



Above: Kevin Boulanger (PhD thesis,
*“Real-Time Realistic Rendering of Nature
Scenes with Dynamic Lighting”*, 2005)

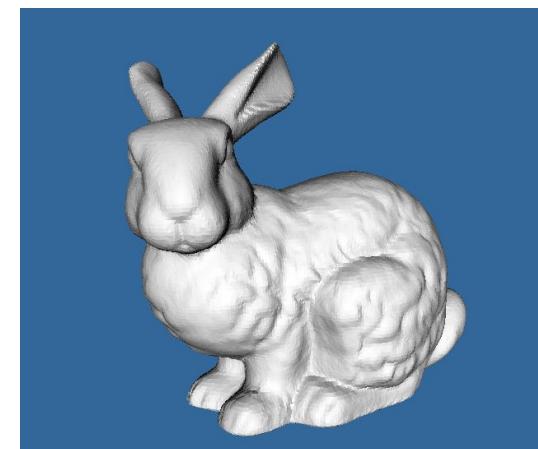
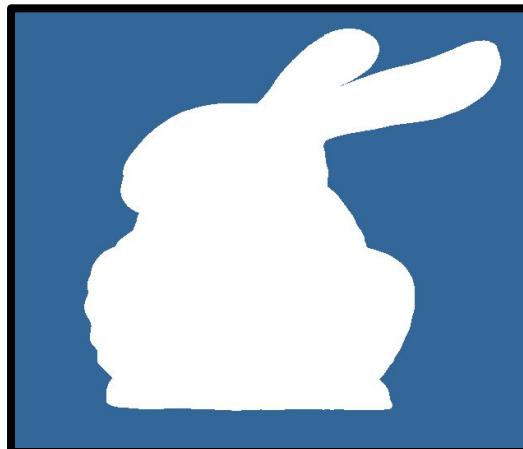


Above: Ben Cloward (“Car paint shader”)

What will you have to write?

It's up to you to implement perspective and lighting.

- 1. Pass geometry to the GPU**
2. Implement perspective on the GPU
3. Calculate lighting on the GPU





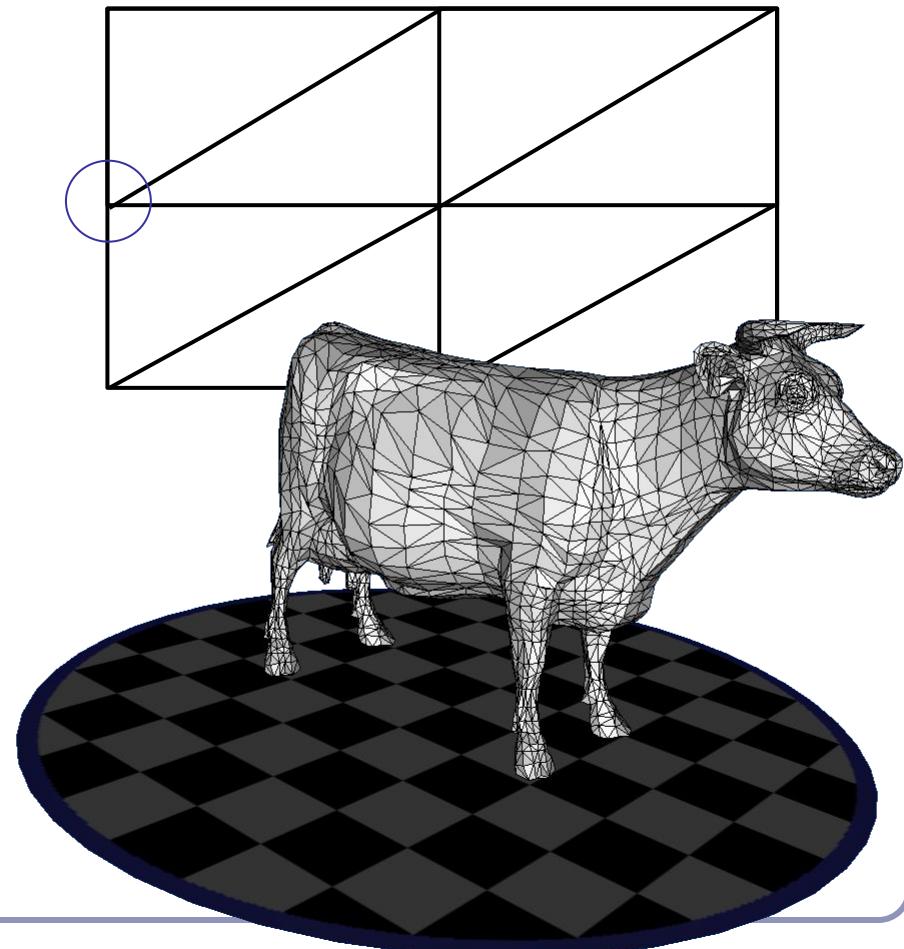
Geometry in OpenGL

The atomic datum of OpenGL is a **vertex**.

- 2d or 3d
- Specify arbitrary details

The fundamental primitives in OpenGL are the **line segment** and **triangle**.

- Very hard to get wrong
- $\{\text{vertices}\} + \{\text{ordering}\}$
= surface



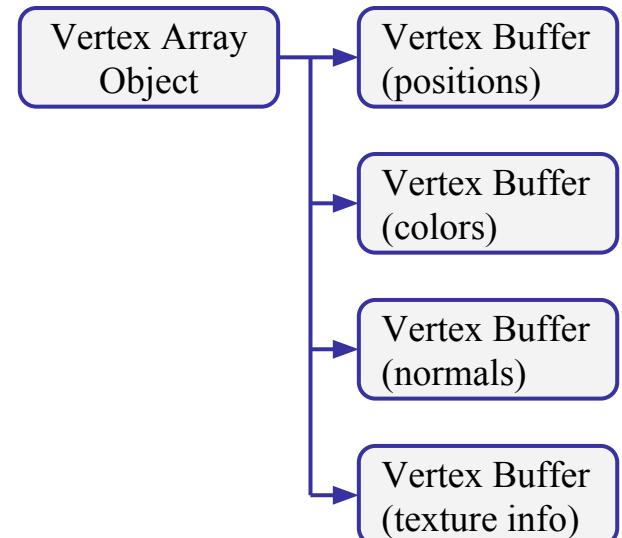


Geometry in OpenGL

Vertex buffer objects store arrays of vertex data--positional or descriptive. With a vertex buffer object (“VBO”) you can compute all vertices at once, pack them into a VBO, and pass them to OpenGL *en masse* to let the GPU processes all the vertices together.

To group different kinds of vertex data together, you can serialize your buffers into a single VBO, or you can bind and attach them to *Vertex Array Objects*. Each vertex array object (“VAO”) can contain multiple VBOs.

Although not required, VAOs help you to organize and isolate the data in your VBOs.





HelloGL.java [1/4]

```
///////////////////////////////
// Set up GLFW window

GLFWErrorCallback errorCallback = GLFWErrorCallback.createPrint(System.err);
GLFW.glfwSetErrorCallback(errorCallback);
GLFW.glfwInit();
GLFW.glfwWindowHint(GLFW.GLFW_CONTEXT_VERSION_MAJOR, 3);
GLFW.glfwWindowHint(GLFW.GLFW_CONTEXT_VERSION_MINOR, 3);
GLFW.glfwWindowHint(GLFW.GLFW_OPENGL_PROFILE, GLFW.GLFW_OPENGL_CORE_PROFILE);
long window = GLFW.glfwCreateWindow(
    800 /* width */, 600 /* height */, "HelloGL", 0, 0);
GLFW.glfwMakeContextCurrent(window);
GLFW.glfwSwapInterval(1);
GLFW.glfwShowWindow(window);

///////////////////////////////
// Set up OpenGL

GL.createCapabilities();
GL11.glClearColor(0.2f, 0.4f, 0.6f, 0.0f);
GL11.glClearDepth(1.0f);
```



HelloGL.java [2/4]

```
//////////  
// Set up minimal shader programs  
  
// Vertex shader source  
String[] vertex_shader = {  
    "#version 330\n",  
    "in vec3 v;",  
    "void main() {",  
    "    gl_Position = ",  
    "        vec4(v, 1.0);",  
    "}"  
};  
  
// Fragment shader source  
String[] fragment_shader = {  
    "#version 330\n",  
    "out vec4 frag_colour;",  
    "void main() {",  
    "    frag_colour = ",  
    "        vec4(1.0);",  
    "}"  
};
```

```
// Compile vertex shader  
int vs = GL20.glCreateShader(  
    GL20.GL_VERTEX_SHADER);  
GL20.glShaderSource(  
    vs, vertex_shader);  
GL20.glCompileShader(vs);  
  
// Compile fragment shader  
int fs = GL20.glCreateShader(  
    GL20.GL_FRAGMENT_SHADER);  
GL20.glShaderSource(  
    fs, fragment_shader);  
GL20.glCompileShader(fs);  
  
// Link vertex and fragment  
// shaders into active program  
int program =  
    GL20.glCreateProgram();  
GL20.glAttachShader(program, vs);  
GL20.glAttachShader(program, fs);  
GL20.glLinkProgram(program);  
GL20.glUseProgram(program);
```



HelloGL.java [3/4]

```
///////////////////////////////
// Set up data

// Fill a Java FloatBuffer object with memory-friendly floats
float[] coords = new float[] { -0.5f, -0.5f, 0, 0, 0.5f, 0, 0.5f, -0.5f, 0 };
FloatBuffer fbo = BufferUtils.createFloatBuffer(coords.length);
fbo.put(coords);                                // Copy the vertex coords into the
floatbuffer
fbo.flip();                                     // Mark the floatbuffer ready for reads

// Store the FloatBuffer's contents in a Vertex Buffer Object
int vbo = GL15 glGenBuffers();                  // Get an OGL name for the VBO
GL15 glBindBuffer(GL15.GL_ARRAY_BUFFER, vbo);   // Activate the VBO
GL15 glBufferData(GL15.GL_ARRAY_BUFFER, fbo, GL15.GL_STATIC_DRAW); // Send VBO data to GPU

// Bind the VBO in a Vertex Array Object
int vao = GL30 glGenVertexArrays();              // Get an OGL name for the VAO
GL30 glBindVertexArray(vao);                   // Activate the VAO
GL20 glEnableVertexAttribArray(0);             // Enable the VAO's first attribute (0)
GL20 glVertexAttribPointer(0, 3, GL11.GL_FLOAT, false, 0, 0); // Link VBO to VAO attrib 0
```



HelloGL.java [4/4]

```
///////////////////////////////
// Loop until window is closed

while (!GLFW.glfwWindowShouldClose(window)) {
    GLFW.glfwPollEvents();

    GL11.glClear(GL11.GL_COLOR_BUFFER_BIT | GL11.GL_DEPTH_BUFFER_BIT);
    GL30 glBindVertexArray(vao);
    GL11.glDrawArrays(GL11.GL_TRIANGLES, 0 /* start */, 3 /* num vertices */);

    GLFW.glfwSwapBuffers(window);
}

///////////////////////////////
// Clean up

GL15.glDeleteBuffers(vbo);
GL30.glDeleteVertexArrays(vao);
GLFW.glfwDestroyWindow(window);
GLFW.glfwTerminate();
GLFW.glfwSetErrorCallback(null).free();
```



Binding multiple buffers in a VAO

Need more info? We can pass more than just coordinate data--we can create as many buffer objects as we want for different types of per-vertex data. This lets us bind vertices with **normals**, **colors**, **texture coordinates**, etc...

Here we bind a vertex buffer object for position data and another for normals:

```
int vao = glGenVertexArrays();
glBindVertexArray(vao);
GL20.glEnableVertexAttribArray(0);
GL20.glEnableVertexAttribArray(1);
GL15 glBindBuffer(GL15.GL_ARRAY_BUFFER, vbo_0);
GL20 glVertexAttribPointer(0, 3, GL11.GL_FLOAT, false, 0, 0);
GL15 glBindBuffer(GL15.GL_ARRAY_BUFFER, vbo_1);
GL20 glVertexAttribPointer(1, 3, GL11.GL_FLOAT, false, 0, 0);
```

Later, to render, we work only with the vertex array:

```
glBindVertexArray(vao);
glDrawArrays(GL_LINE_STRIP, 0, data.length);
```

Caution--all VBOs in a VAO must describe the same number of vertices!



Accessing named GLSL attributes from Java

```
// Vertex shader  
// ...  
  
#version 330  
  
in vec3 v;  
void main() {  
    gl_Position =  
        vec4(v, 1.0);  
}  
  
// ...
```

```
// ...  
  
glEnableVertexAttribArray(0);  
glVertexAttribPointer(0,  
    3, GL_FLOAT, false, 0, 0);  
  
// ...
```

The HelloGL sample code hardcodes the assumption that the vertex shader input field ‘v’ is the zeroeth input (position 0).

That’s unstable: never rely on a fixed ordering.
Instead, fetch the attrib location:

```
int vLoc =  
    GL20.glGetAttribLocation(program, "v");  
GL20.glEnableVertexAttribArray(vLoc);  
GL20 glVertexAttribPointer(vLoc,  
    3, GL_FLOAT, false, 0, 0);
```

This enables greater flexibility and Java code that can adapt to dynamically-changing vertex and fragment shaders.



Improving data throughput

You configure how OpenGL interprets the vertex buffer. Vertices can be interpreted directly, or *indexed* with a separate integer indexing buffer. By re-using vertices and choosing ordering / indexing carefully, you can reduce the number of raw floats sent from the CPU to the GPU dramatically.

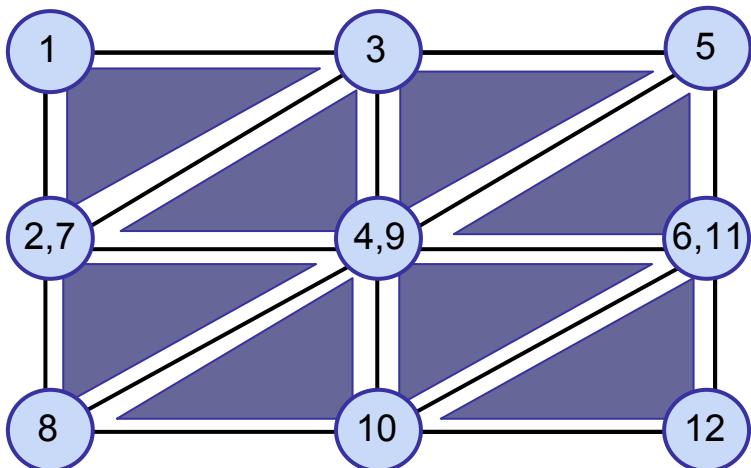
Options include line primitives--

- GL_LINES
- GL_LINE_STRIP
- GL_LINE_LOOP

--triangle primitives--

- GL_TRIANGLES
- GL_TRIANGLE_STRIP
- GL_TRIANGLE_FAN

--and more. OpenGL also offers *backface culling* and other optimizations.



*Triangle-strip vertex indexing
(counter-clockwise ordering)*



Memory management: Lifespan of an OpenGL object

Most objects in OpenGL are created and deleted explicitly. Because these entities live in the GPU, they're outside the scope of Java's garbage collection.

This means that **you must handle your own memory cleanup**.

```
// create and bind buffer object  
int name = glGenBuffers();  
glBindBuffer(GL_ARRAY_BUFFER, name);  
  
// work with your object  
// ...  
  
// delete buffer object, free memory  
glDeleteBuffers(name);
```





Emulating classic OpenGL 1.1 direct-mode rendering in modern GL

The original OpenGL API allowed you to use *direct mode* to send data for immediate output:

```
glBegin(GL_QUADS);
    glColor3f(0, 1, 0);
    glNormal3f(0, 0, 1);
    glVertex3f(1, -1, 0);
    glVertex3f(1, 1, 0);
    glVertex3f(-1, 1, 0);
    glVertex3f(-1, -1, 0);
glEnd();
```

Direct mode was very inefficient: the GPU was throttled by the CPU.

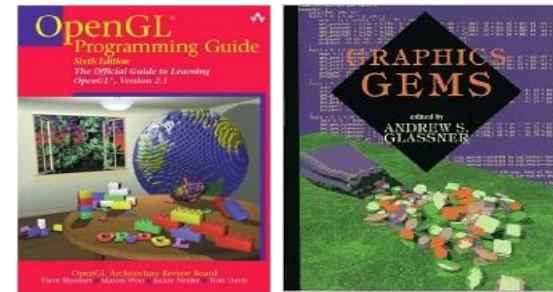
You can emulate the GL1.1 API:

```
class GLVertexData {
    void begin(mode) { ... }
    void color(color) { ... }
    void normal(normal) { ... }
    void vertex(vertex) { ... }
    ...
    void compile() { ... }
}
```

The method `compile()` can encapsulate all the vertex buffer logic, making each instance a self-contained buffer object.

Check out a working example in the `class framework.GLVertexData` on the course github repo.

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

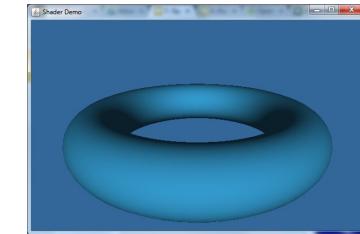
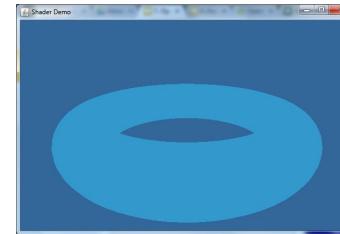
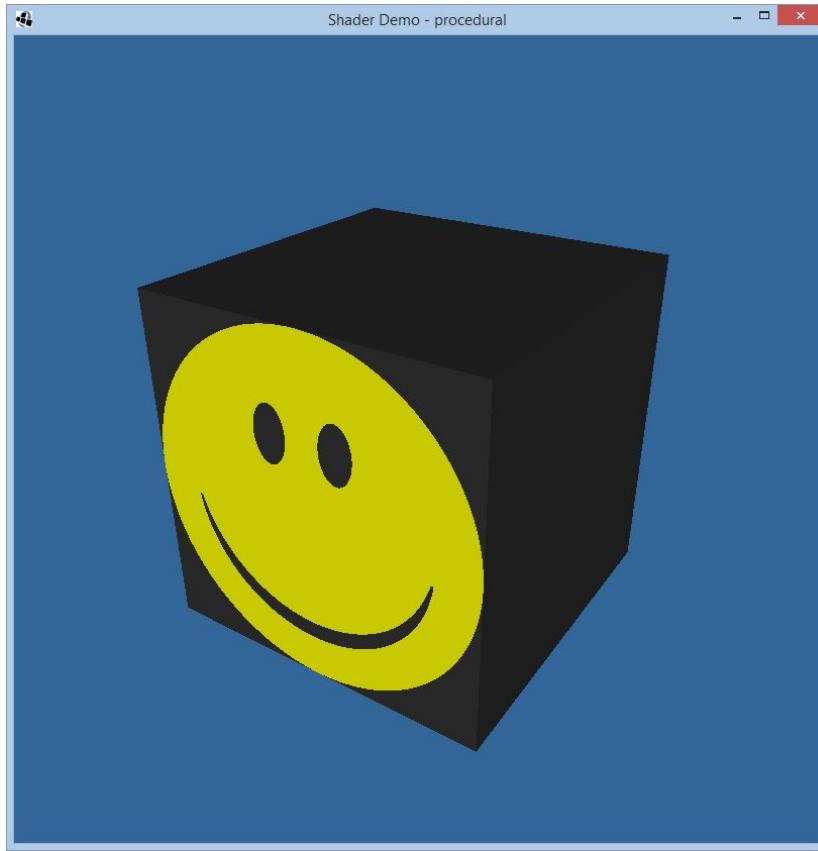
There's also an OpenGL-ES reference, same series

OpenGL Insights (2012), by Cozzi and Riccio

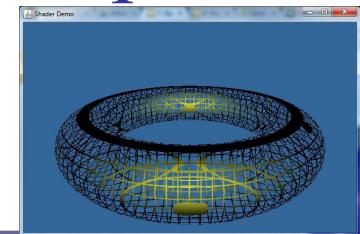
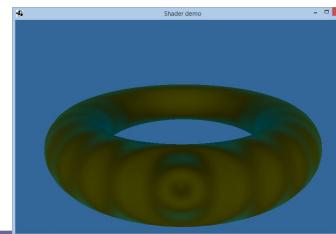
OpenGL Shading Language (2009), by Rost, Licea-Kane, Ginsburg et al

The *Graphics Gems* series from Glassner

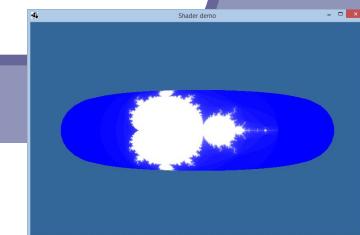
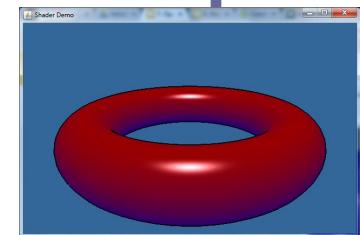
ShaderToy.com, a web site by Inigo Quilez (Pixar) dedicated to amazing shader tricks and raycast scenes



Advanced Graphics



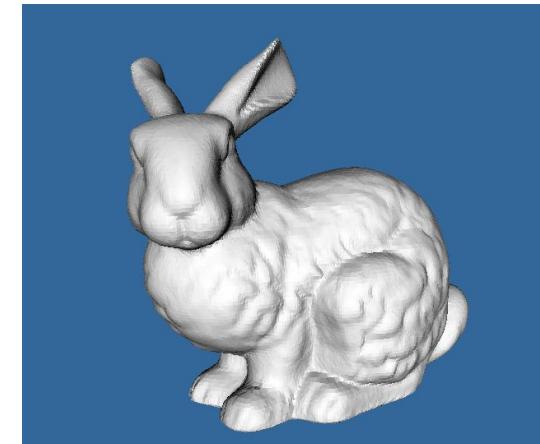
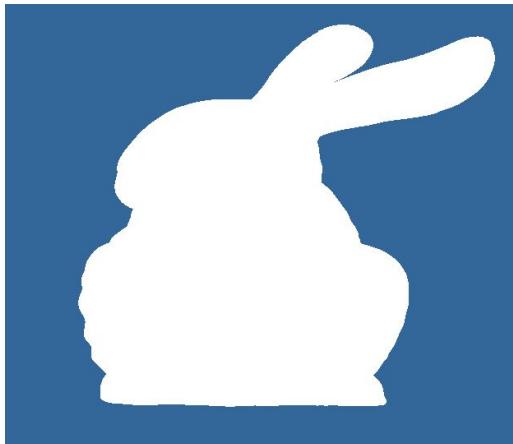
OpenGL and Shaders II



2.Perspective and Camera Control

It's up to you to implement perspective and lighting.

1. Pass geometry to the GPU
2. **Implement perspective on the GPU**
3. Calculate lighting on the GPU





Getting some perspective

To add *3D perspective* to our flat model, we face three challenges:

- Compute a 3D perspective matrix
- Pass it to OpenGL, and on to the GPU
- Apply it to each vertex

To do so we're going to need to apply our perspective matrix in the shader, which means we'll need to build our own 4x4 perspective transform.



4x4 perspective matrix transform

Every OpenGL package provides utilities to build a perspective matrix. You'll usually find a method named something like *glGetFrustum()* which will assemble a 4x4 grid of floats suitable for passing to OpenGL.

Or you can build your own:

$$P = \begin{pmatrix} \frac{1}{ar \cdot \tan(\frac{\alpha}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{\alpha}{2})} & 0 & 0 \\ 0 & 0 & \frac{-NearZ - FarZ}{NearZ - FarZ} & \frac{2 \cdot FarZ \cdot NearZ}{NearZ - FarZ} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

α : Field of view, typically 50°

ar : Aspect ratio of width over height

$NearZ$: Near clip plane

$FarZ$: Far clip plane



Writing uniform data from Java

Once you have your perspective matrix, the next step is to copy it out to the GPU as a **Mat4**, GLSL's 4x4 matrix type.

1. Convert your floats to a FloatBuffer:

```
float data[][] = /* your 4x4 matrix here */
FloatBuffer buffer = BufferUtils.createFloatBuffer(16);
for (int col = 0; col < 4; col++) {
    for (int row = 0; row < 4; row++) {
        buffer.put((float) (data[row][col]));
    }
}
buffer.flip();
```

2. Write the FloatBuffer to the named uniform:

```
int uniformLoc = GL20.glGetUniformLocation(
    program, "name");
if (uniformLoc != -1) {
    GL20.glUniformMatrix4fv(uniformLoc, false, buffer);
}
```



Reading uniform data in GLSL

The **FloatBuffer** output is received in the shader as a *uniform* input of type **Mat4**.

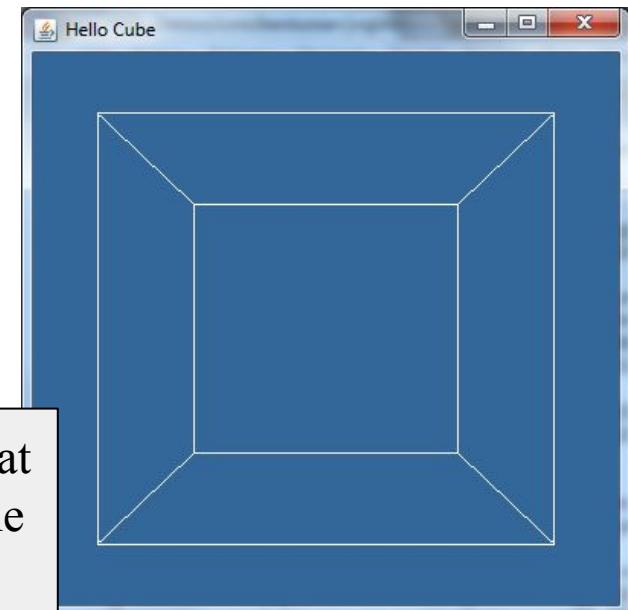
This shader takes a matrix and applies it to each vertex:

```
#version 330

uniform mat4 modelToScreen;

in vec4 vPosition;

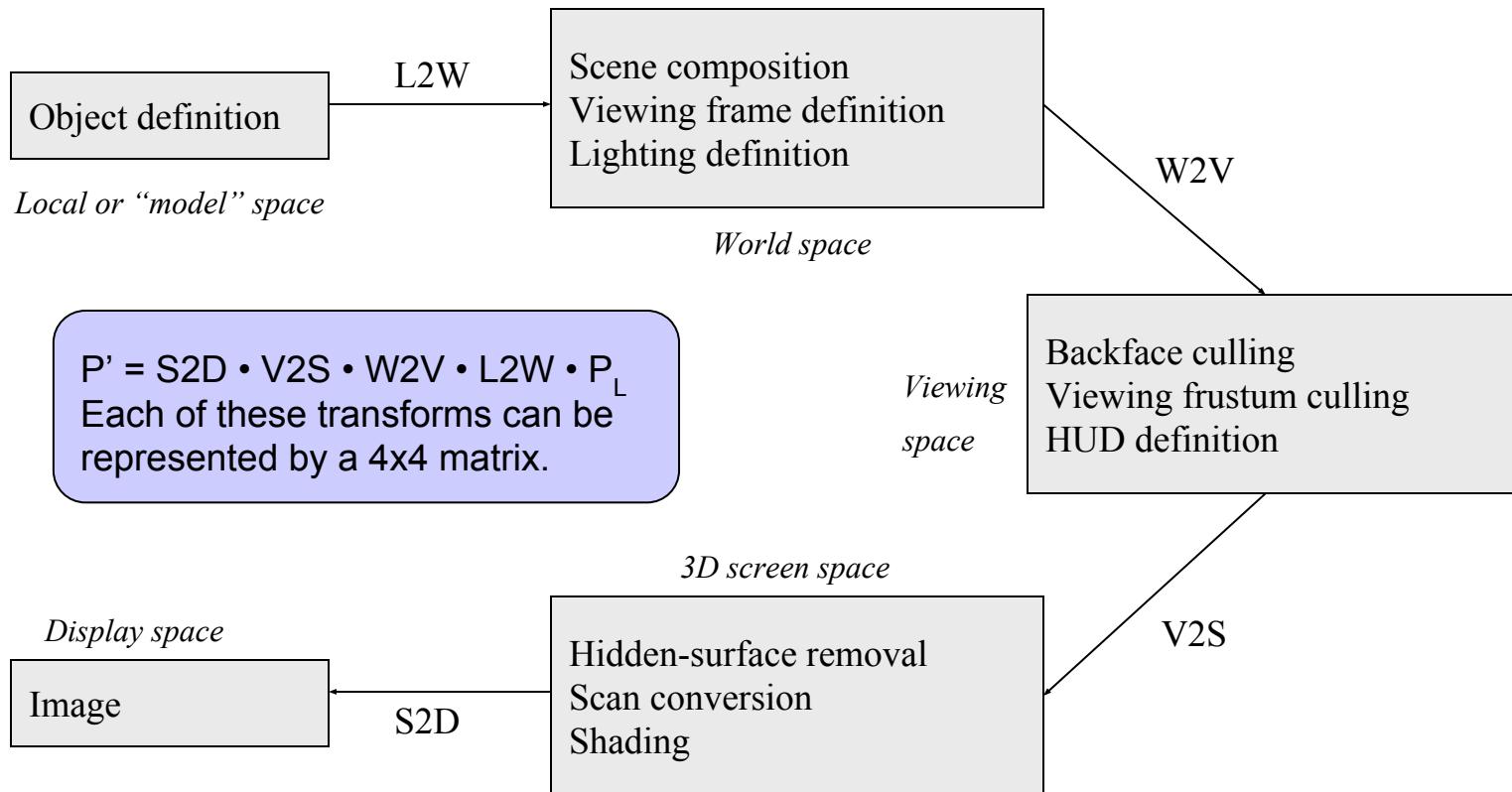
void main() {
    gl_Position = modelToScreen * vPosition;
}
```



Use uniforms for fields that are constant throughout the rendering pass, such as transform matrices and lighting coordinates.



Object position and camera position: a ‘pipeline’ model of matrix transforms





The pipeline model in OpenGL & GLSL

A flexible 3D graphics framework will track each transform:

- The object’s current transform
- The camera’s transform
- The viewing perspective transform

These matrices are all “constants” for the duration of a single frame of rendering. Each can be written to a 16-float buffer and sent to the GPU with `glUniformMatrix4fv`.

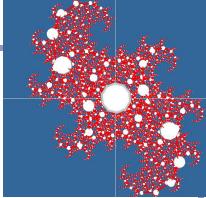
Remember to fetch uniform names with `glGetUniformLocation`, never assume ordering.

```
#version 330

uniform mat4 modelToWorld;
uniform mat4 worldToCamera;
uniform mat4 cameraToScreen;

in vec3 v;

void main() {
    gl_Position = cameraToScreen
        * worldToCamera
        * modelToWorld
        * vec4(v, 1.0);
}
```

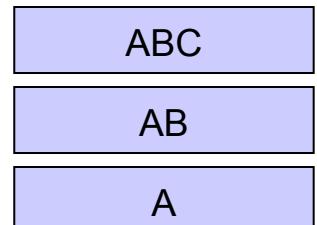


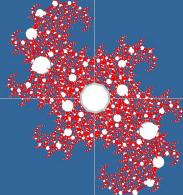
The pipeline model in software: The *matrix stack* design pattern

A common design pattern in 3D graphics, especially when objects can contain other objects, is to use *matrix stacks* to store stacks of matrices. The topmost matrix is the product of all matrices below.

- This allows you to build a local frame of reference—local space—and apply transforms within that space.
- Remember: matrix multiplication is associative but not commutative.
 - $ABC = A(BC) = (AB)C \neq ACB \neq BCA$

Pre-multiplying matrices that will be used more than once is faster than multiplying many matrices every time you render a primitive.

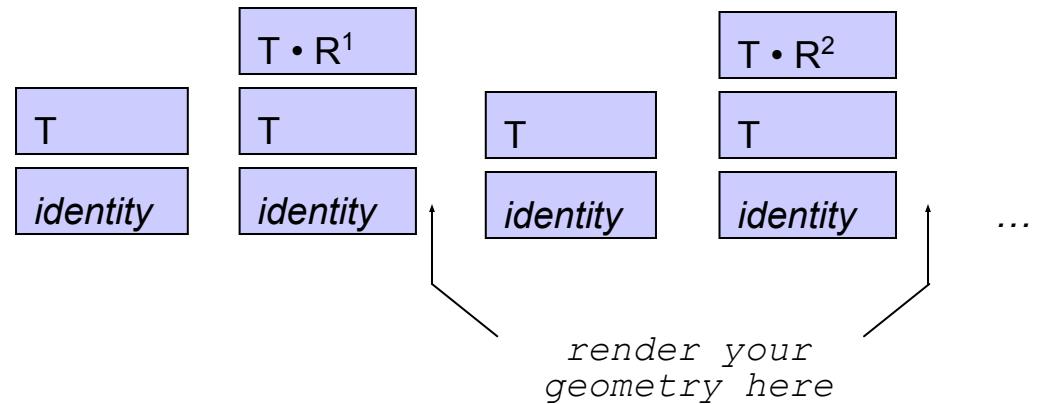


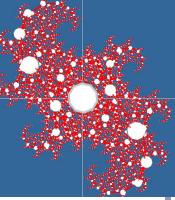


Matrix stacks

Matrix stacks are designed for nested relative transforms.

```
pushMatrix();
  translate(0,0,-5);
  pushMatrix();
    rotate(45,0,1,0);
    render();
  popMatrix();
  pushMatrix();
    rotate(-45,0,1,0);
    render();
  popMatrix();
popMatrix();
```

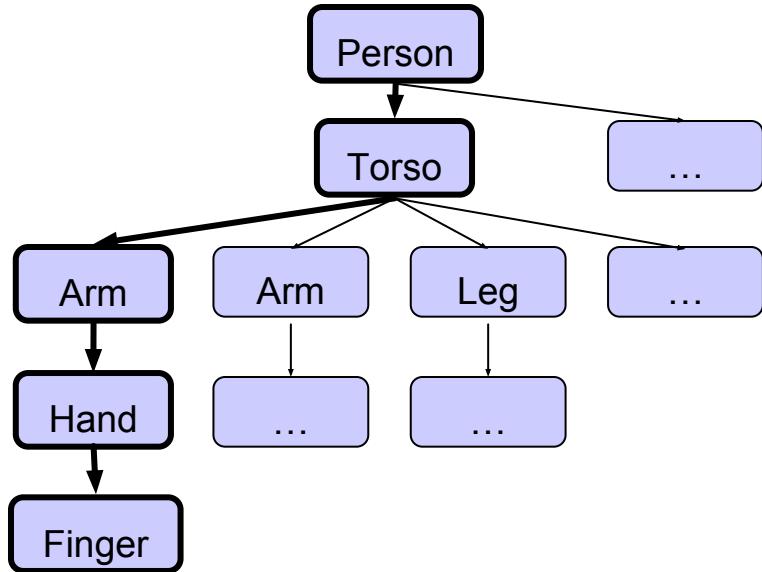




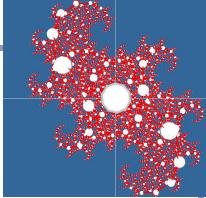
Scene graphs

A *scene graph* is a tree of scene elements where a child's transform is relative to its parent.

The final transform of the child is the ordered product of all of its ancestors in the tree.

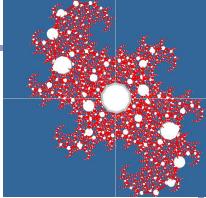


$$\begin{aligned} M_{\text{fingerToWorld}} &= \\ (M_{\text{person}} \cdot M_{\text{torso}} \cdot M_{\text{arm}} \cdot M_{\text{hand}} \cdot M_{\text{finger}}) \end{aligned}$$

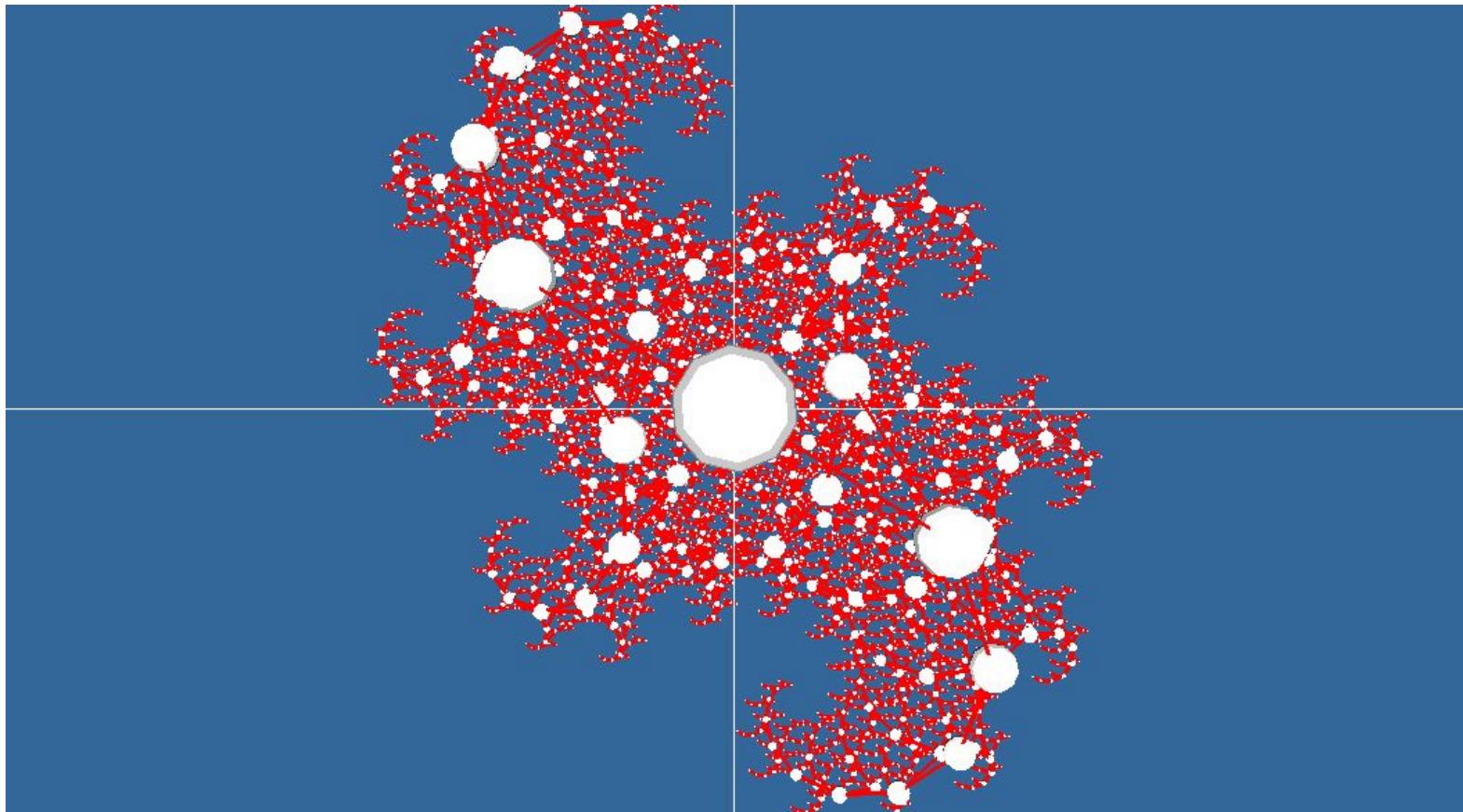


Hierarchical modeling in action

```
void renderLevel(GL gl, int level, float t) {  
    pushMatrix();  
    rotate(t, 0, 1, 0);  
    renderSphere(gl);  
    if (level > 0) {  
        scale(0.75f, 0.75f, 0.75f);  
        pushMatrix();  
        translate(1, -0.75f, 0);  
        renderLevel(gl, level-1, t);  
        popMatrix();  
        pushMatrix();  
        translate(-1, -0.75f, 0);  
        renderLevel(gl, level-1, t);  
        popMatrix();  
    }  
    popMatrix();  
}
```



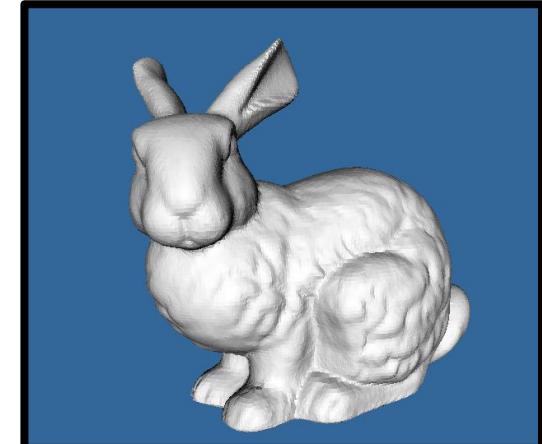
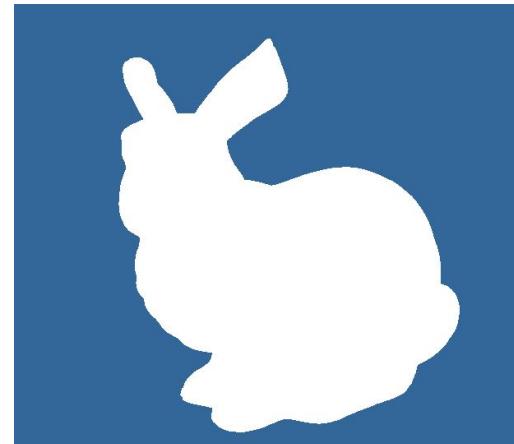
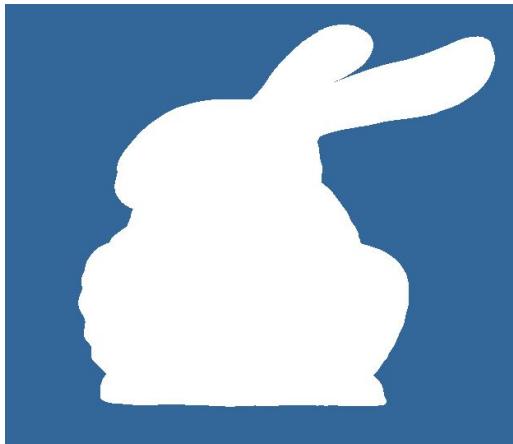
Hierarchical modeling in action



3. Lighting and Shading

It's up to you to implement perspective and lighting.

1. Pass geometry to the GPU
2. Implement perspective on the GPU
- 3. Calculate lighting on the GPU**





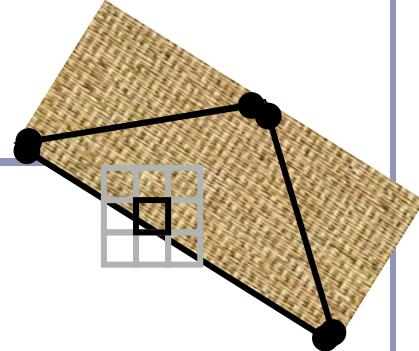
Lighting and Shading (a quick refresher)

Recall the classic **lighting equation**:

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

where...

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



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.



Lighting and Shading: required data

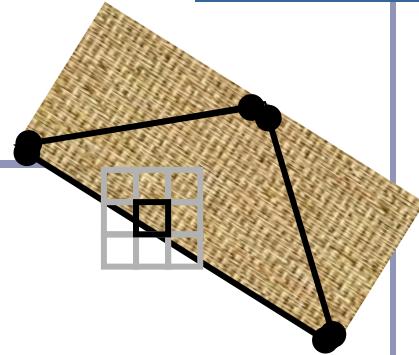
Shading means we need extra data about vertices.

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...

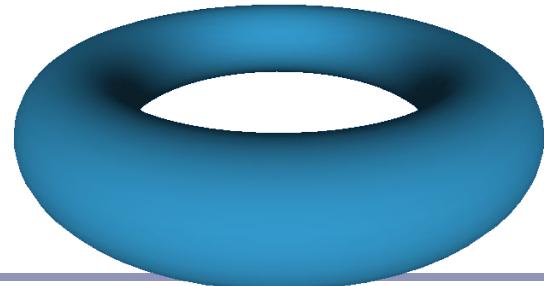
We also need global state:

- Camera perspective transform
- 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;

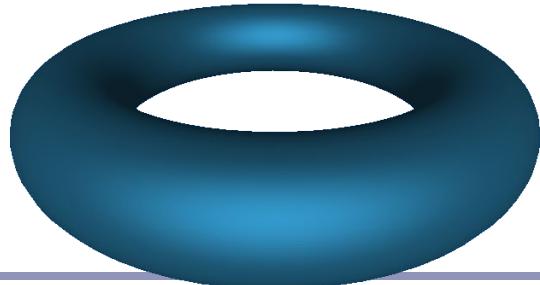
out vec4 fragmentColor;

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

Diffuse lighting
 $d = k_D(N \cdot L)$
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);
}
```

$$\begin{aligned}a &= k_A \\d &= k_D(N \cdot L) \\s &= k_S(E \cdot R)^n\end{aligned}$$



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.
- This, combined with edge-highlighting through a second renderer pass, creates models which look more like engineering schematic diagrams.

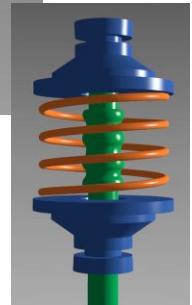
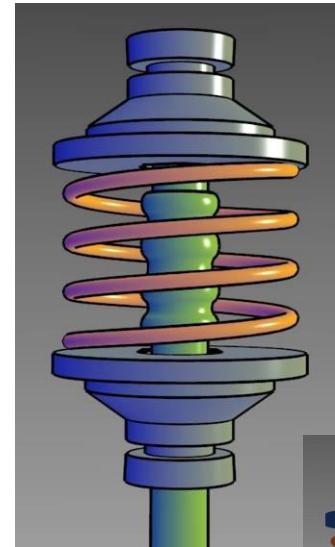


Image source: “A Non-Photorealistic Lighting Model For Automatic Technical Illustration”, Gooch, Gooch, Shirley and Cohen (1998). Compare the Gooch shader, above, to the Phong shader (right).



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.)

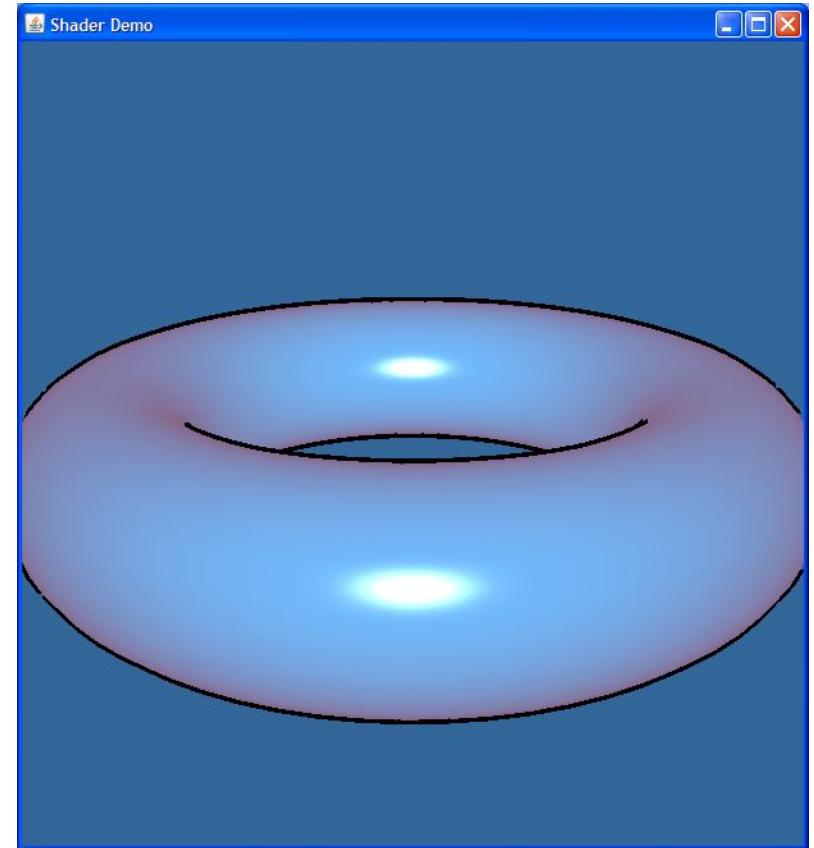
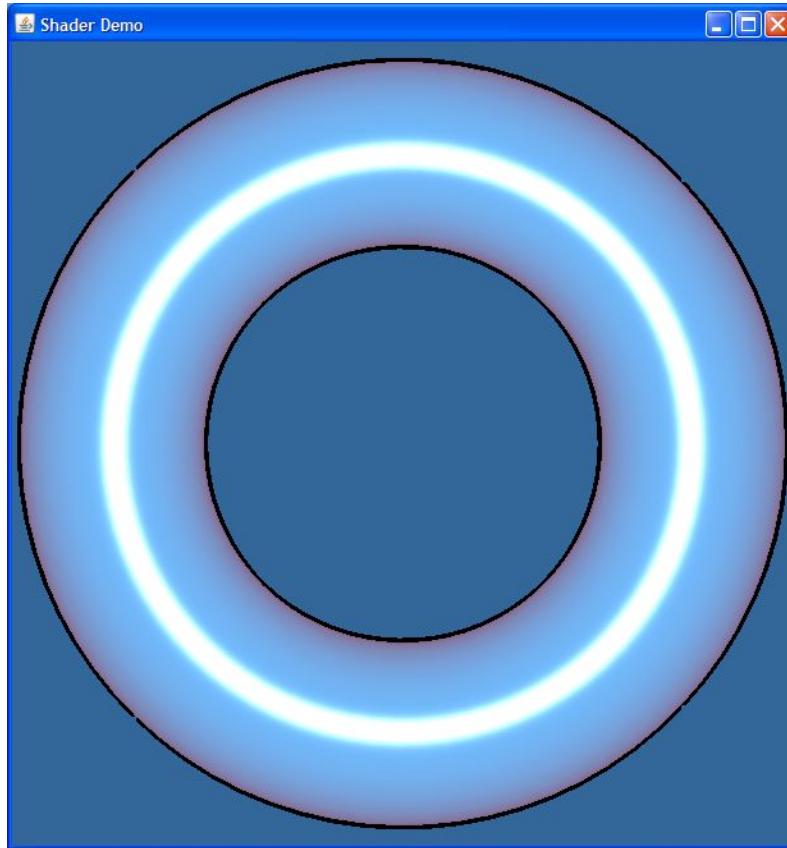
In the fragment shader source, this is used to choose the weighted color by clipping with the `a` component:

```
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

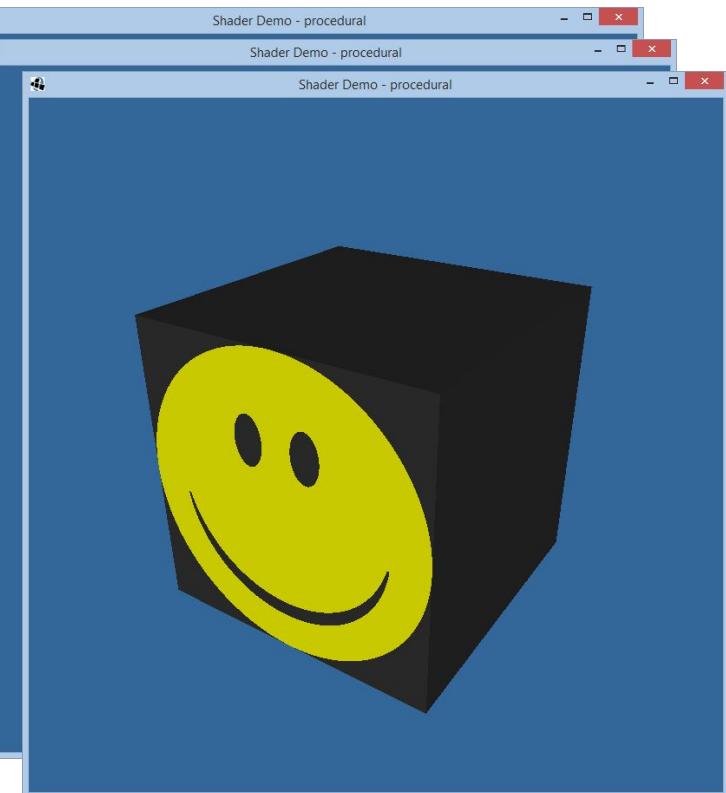


Procedural texturing in the fragment shader

```
// ...
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);

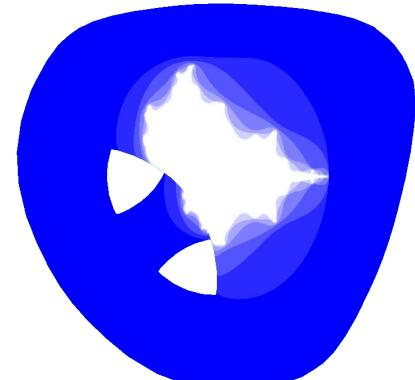
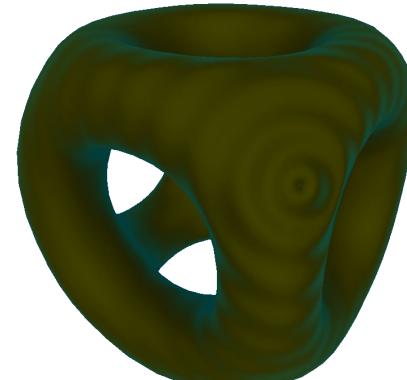
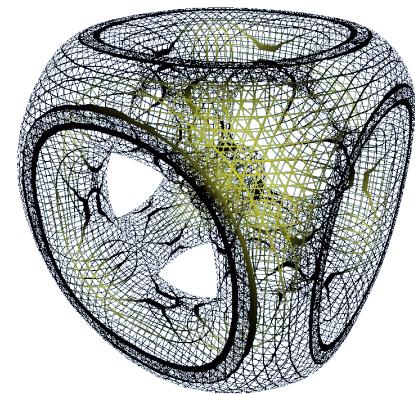
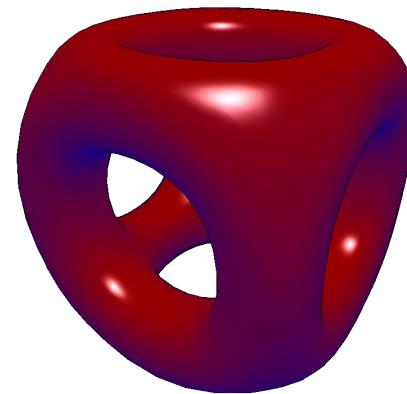
    vec3 color = (isMouth || isEye || isOutsideFace)
        ? BLACK : YELLOW;
    fragmentColor = vec4(color, 1.0);
}
```



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

Advanced surface effects

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



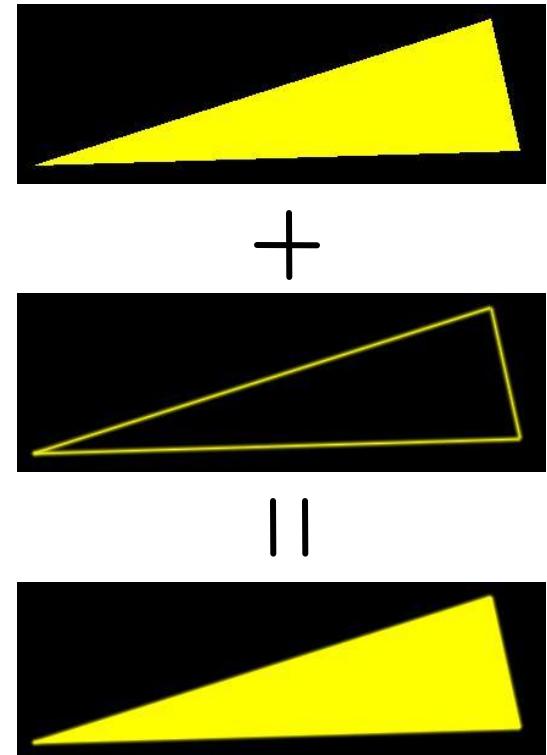
Antialiasing on the GPU

Hardware antialiasing can dramatically improve image quality.

- The naïve approach is to supersample the image
- This is easier in shaders than it is in standard software
- But it really just postpones the problem.

Several GPU-based antialiasing solutions have been found.

- Eric Chan published an elegant polygon-based antialiasing approach in 2004 which uses the GPU to prefilter the edges of a model and then blends the filtered edges into the original polygonal surface. (See figures at right.)



Antialiasing on the GPU

One clever form of antialiasing is *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 the shader language by the methods `dFdx(F)` and `dFdy(F)`.

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



(A)



(B)



(C)

(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.

Particle systems on the GPU

Shaders extend the use of *texture memory* dramatically. Shaders can write to texture memory, and textures are no longer limited to being two-dimensional planes of RGB(A).

- A particle system can be represented by storing a position and velocity for every particle.
- A fragment shader can render a particle system entirely in hardware by using texture memory to store and evolve particle data.

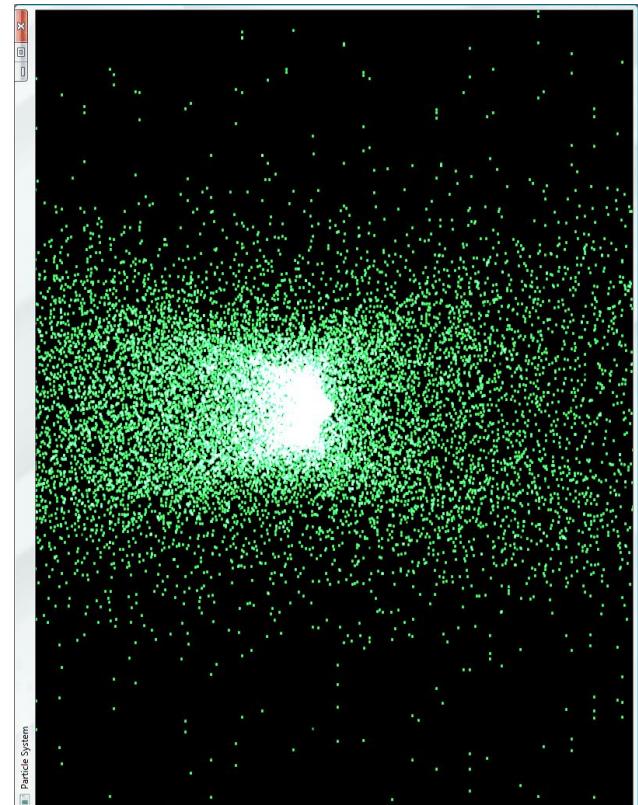
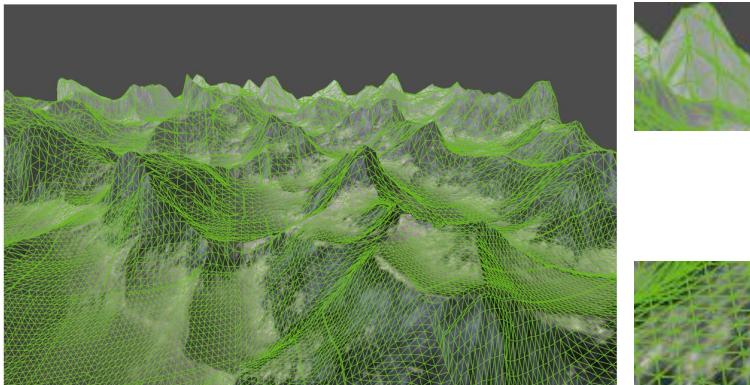


Image by Michael Short

Tessellation shaders

Tesselation is a new shader type introduced in OpenGL 4.x. Tessellation 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.

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

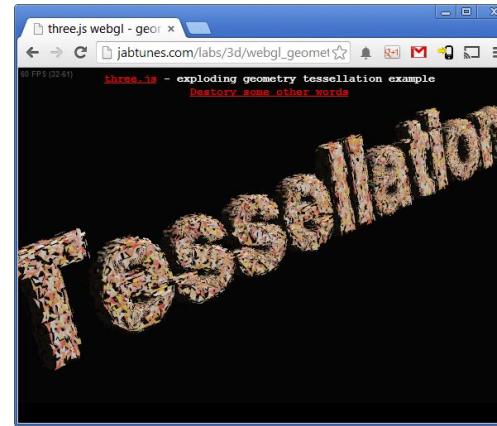


Florian Boesch's LOD terrain demo

<http://codeflow.org/entries/2010/nov/07/opengl-4-tessellation/>

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.



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);
```

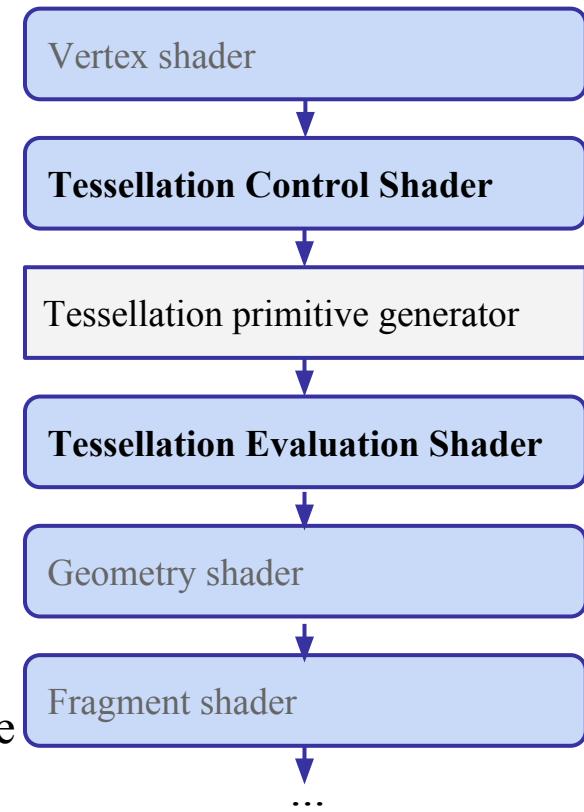
- You tell OpenGL to render your patches:

```
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.

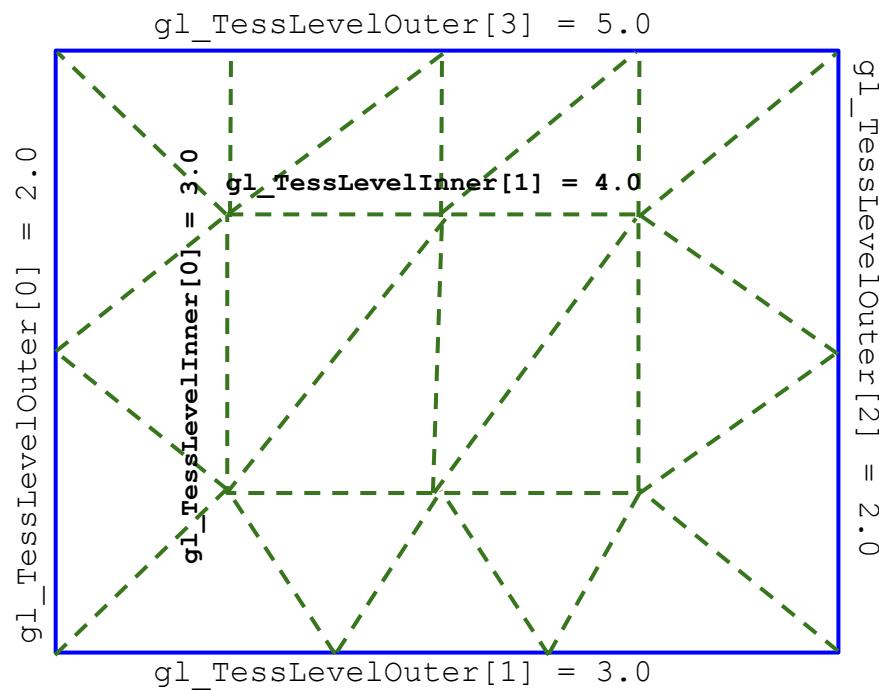


Tessellation shaders

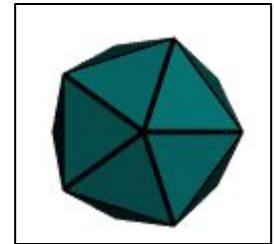
- The *tessellation primitive generator* generates new vertices along the outer edge and inside the patch, as specified by `gl_TessLevelOuter[]` and `gl_TessLevelInner[]`.

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

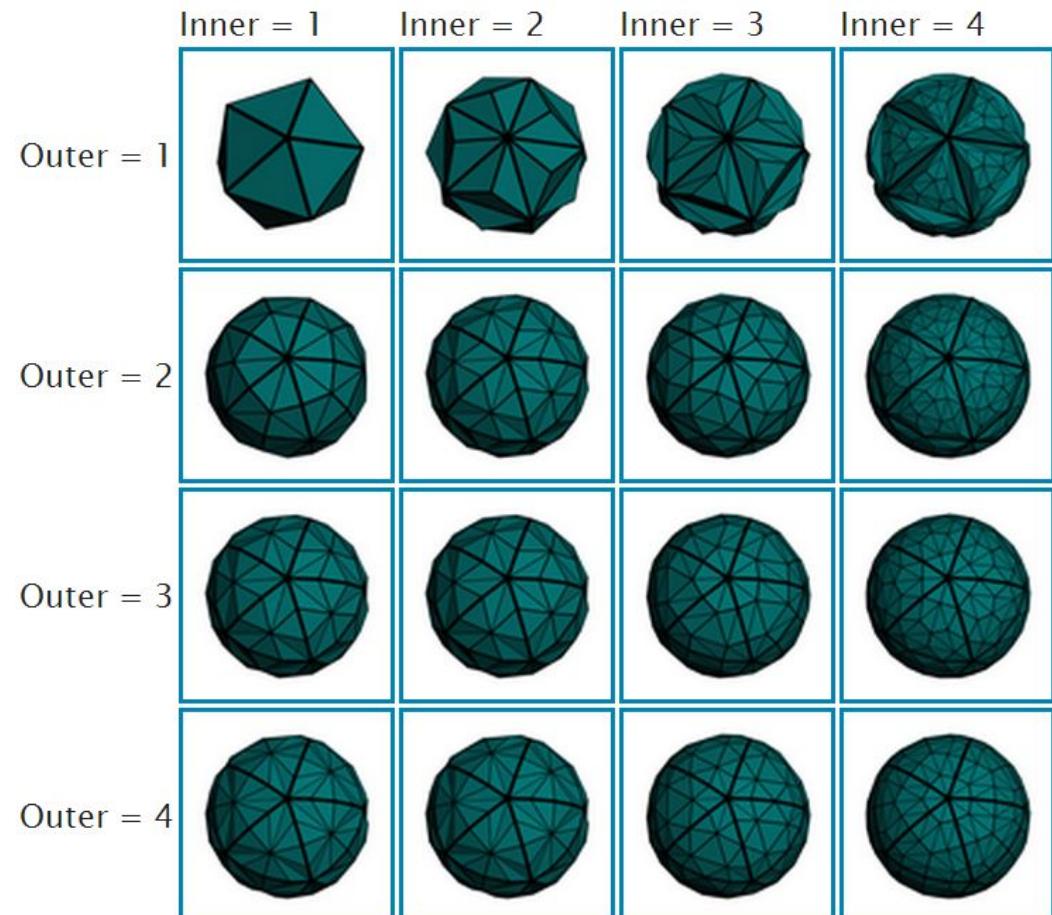
Triangles are indexed similarly, but only use the first three `Outer` and the first `Inner` array field.



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.



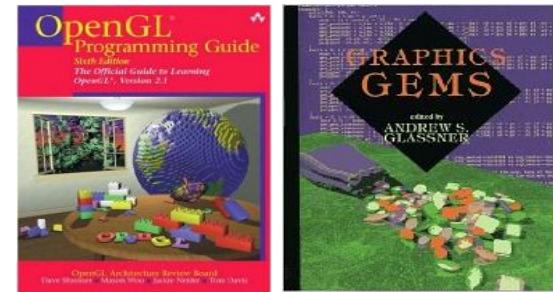
CPU vs GPU – an object demonstration



“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

ShaderToy.com, a web site by Inigo Quilez (Pixar) dedicated to amazing shader tricks and raycast scenes

Advanced Graphics

Ray Tracing

All the maths

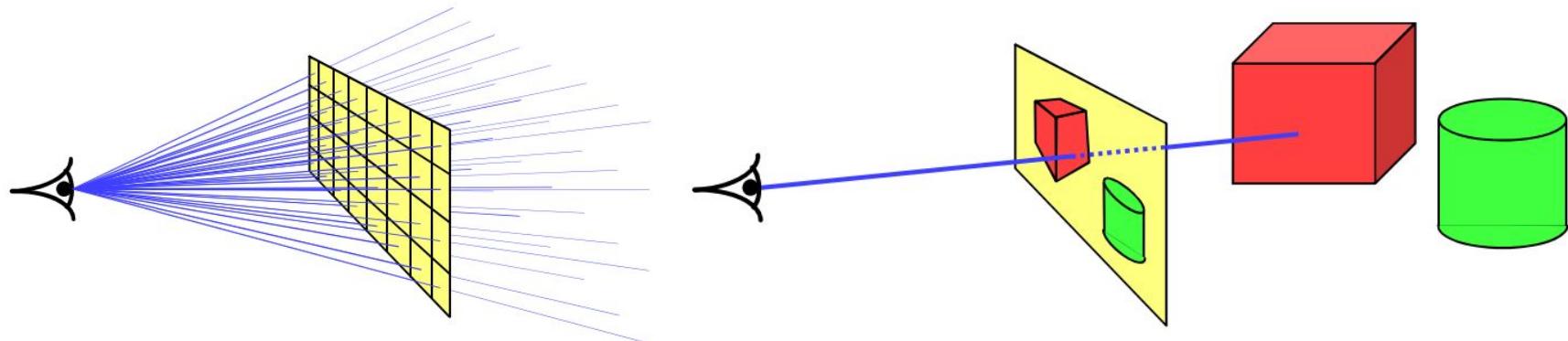
“Cornell Box” by Steven Parker, University of Utah.

A tera-ray monte-carlo rendering of the Cornell Box, generated in 2 CPU years on an Origin 2000. The full image contains 2048 x 2048 pixels with over 100,000 primary rays per pixel (317 x 317 jittered samples). Over one trillion rays were traced in the generation of this image.

Ray tracing

- A powerful alternative to polygon scan-conversion techniques
- An elegantly simple algorithm:

Given a set of 3D objects, shoot a ray from the eye through the center of every pixel and see what it hits.



The algorithm

Select an eye point and a screen plane.

for (every pixel in the screen plane):

Find the ray from the eye through the pixel's center.

for (each object in the scene):

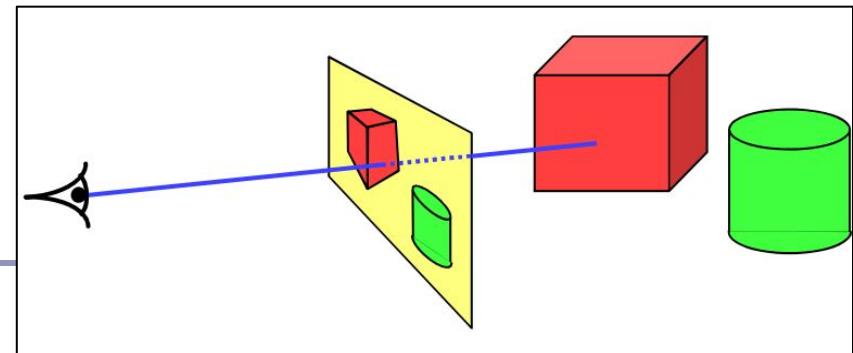
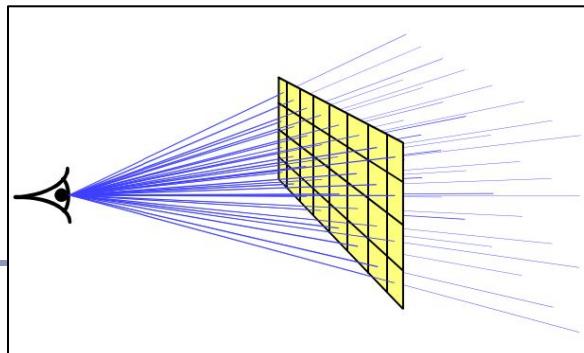
 if (the ray hits the object):

 if (the intersection is the nearest (so far) to the eye):

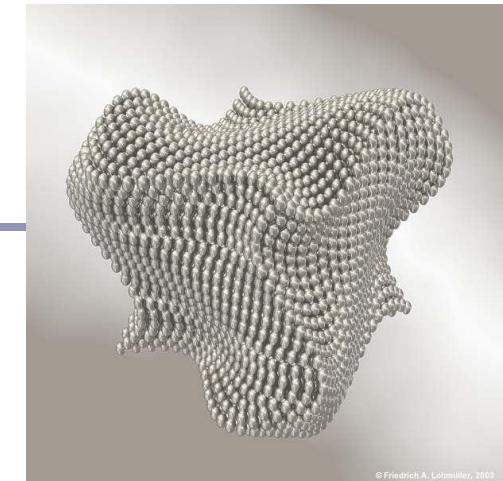
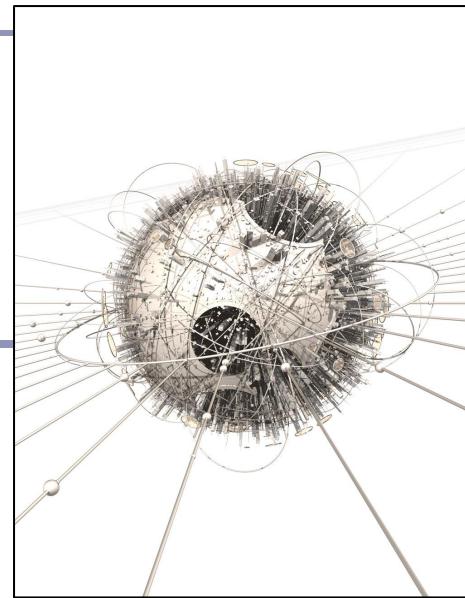
Record the intersection point.

Record the color of the object at that point.

Set the screen plane pixel to the nearest recorded color.



Examples



["Villarceau Circles"](#) by [Tor Olav Kristensen](#) (2004)



It doesn't take much code

The basic algorithm is straightforward, but there's much room for subtlety

- Refraction
- Reflection
- Shadows
- Anti-aliasing
- Blurred edges
- Depth-of-field effects
- ...



```
typedef struct{double x,y,z;}vec;vec U,black,amb={.02,.02,.02};
struct sphere{vec cen,color;double rad,kd,ks,kt,kl,ir;}*s,*best
,sph[]={0.,6.,.5,1.,1.,1.,.9,.05,.2,.85,0.,1.7,-1.,8.,-.5,1.,.5
,.2,1.,.7,.3,0.,.05,1.2,1.,8.,-.5,.1,.8,.8,1.,.3,.7,0.,0.,1.2,3
,,-6.,15.,1.,.8,1.,7.,0.,0.,0.,.6,1.5,-3.,-3.,12.,.8,1.,1.,5.,0
.,0.,0.,.5,1.5,};int yx;double u,b,tmin,sqrt(),tan();double
vdot(vec A,vec B){return A.x*B.x+A.y*B.y+A.z*B.z;}vec vcomb(
double a,vec A,vec B){B.x+=a*A.x;B.y+=a*A.y;B.z+=a*A.z;return
B;}vec vunit(vec A){return vcomb(1./sqrt(vdot(A,A)),A,black);}
struct sphere*intersect(vec P,vec D){best=0;tmin=10000;s=sph+5;
while(s-->sph)b=vdot(D,U=vcomb(-1.,P,s->cen)),u=b*b-vdot(U,U)+
s->rad*s->rad,u=u>0?sqrt(u):10000,u=b-u>0.000001?b-u:b+u,tmin=
u>0.00001&&u<tmin?best=s,u:tmin;return best;}vec trace(int
level,vec P,vec D){double d,eta,e;vec N,color;struct sphere*s,
*l;if(!level--)return black;if(s=intersect(P,D));else return
amb;color=amb;eta=s->ir;d=-vdot(D,N=vunit(vcomb(-1.,P=vcomb(
tmin,D,P),s->cen)));if(d<0)N=vcomb(-1.,N,black),eta=1/eta,d=
-d;l=sph+5;while(l-->sph)if((e=l->kl*vdot(N,U=vunit(vcomb(-1.,
P,l->cen))))>0&&intersect(P,U)==l)color=vcomb(e,l->color,color);
U=s->color;color.x*=U.x;color.y*=U.y;color.z*=U.z;e=1-eta*eta*(1-
d*d);return vcomb(s->kt,e>0?trace(level,P,vcomb(eta,D,vcomb(
eta*d-sqrt(e),N,black))):black,vcomb(s->ks,trace(level,P,vcomb(
2*d,N,D)),vcomb(s->kd,color,vcomb(s->kl,U,black))));}main(){int
d=512;printf("%d %d\n",d,d);while(yx<d*d){U.x=yx%d-d/2;U.z=d/2-
yx++/d;U.y=d/2/tan(25/114.5915590261);U=vcomb(255.,trace(3,
black,vunit(U)),black);printf("%0.f %0.f %0.f\n",U.x,U.y,U.z);}
/*minray!*/
```

Paul Heckbert's 'minray' ray tracer, which fit on the back of his business card. (circa 1983)

Running time

The ray tracing time for a scene is a function of
(num rays cast) x
(num lights) x
(num objects in scene) x
(num reflective surfaces) x
(num transparent surfaces) x
(num shadow rays) x
(ray reflection depth) x ...



Image by nVidia

Contrast this to polygon rasterization: time is a function of the number of elements in the scene times the number of lights.

Ray-traced illumination

Once you have the point P (the intersection of the ray with the nearest object) you'll compute how much each of the lights in the scene illuminates P.

$$\text{diffuse} = 0$$

$$\text{specular} = 0$$

for (each light L_i in the scene):

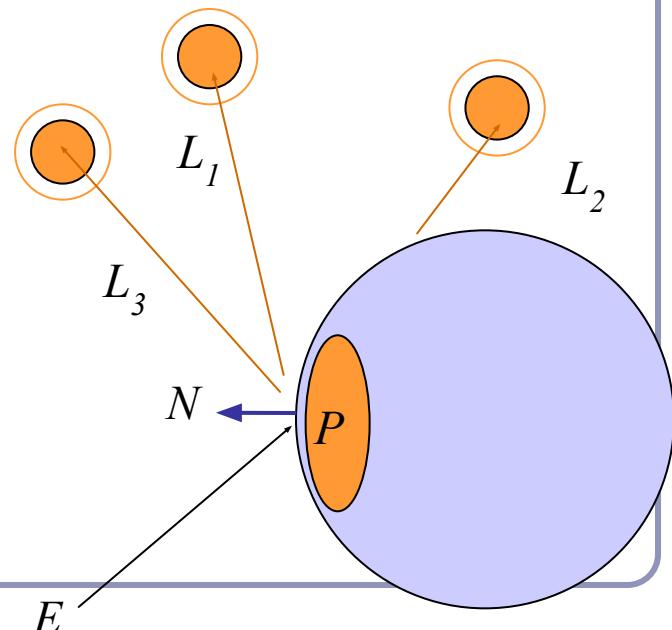
if $(N \bullet L) > 0$:

[Optionally: if (a ray from P to L_i can reach L_i):]

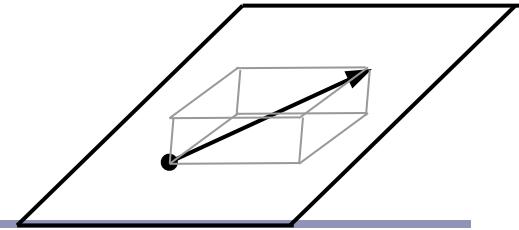
$$\text{diffuse} += k_D(N \bullet L)$$

$$\text{specular} += k_S(R \bullet E)^n$$

$$\text{intensity at } P = \text{ambient} + \text{diffuse} + \text{specular}$$



Hitting things with rays



A ray is defined parametrically as

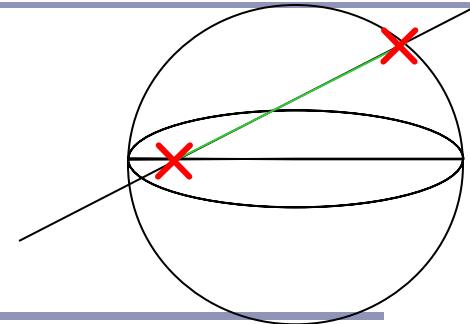
$$P(t) = E + tD, \quad t \geq 0 \quad (\alpha)$$

where E is the ray's origin (our eye position) and D is the ray's direction, a unit-length vector.

We expand this equation to three dimensions, x , y and z :

$$\left. \begin{array}{l} x(t) = x_E + tx_D \\ y(t) = y_E + ty_D \\ z(t) = z_E + tz_D \end{array} \right\} \quad t \geq 0 \quad (\beta)$$

Hitting things with rays: Sphere



The unit sphere, centered at the origin, has the implicit equation

$$x^2 + y^2 + z^2 = 1 \quad (\gamma)$$

Substituting equation (β) into (γ) gives

$$(x_E + tx_D)^2 + (y_E + ty_D)^2 + (z_E + tz_D)^2 = 1$$

which expands to

$$t^2(x_D^2 + y_D^2 + z_D^2) + t(2x_Ex_D + 2y_Ey_D + 2z_Ez_D) + (x_E^2 + y_E^2 + z_E^2 - 1) = 0$$

which is of the form

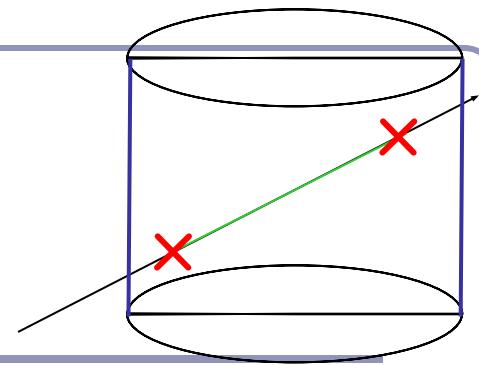
$$at^2 + bt + c = 0$$

which can be solved for t :

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

...giving us two points of intersection.

Hitting things with rays: Cylinder



The infinite unit cylinder, centered at the origin, has the implicit equation

$$x^2 + y^2 = 1 \quad (\delta)$$

Substituting equation (β) into (δ) gives

$$(x_E + tx_D)^2 + (y_E + ty_D)^2 = 1$$

which expands to

$$t^2(x_D^2 + y_D^2) + t(2x_Ex_D + 2y_Ey_D) + (x_E^2 + y_E^2 - 1) = 0$$

which is of the form

$$at^2 + bt + c = 0$$

which can be solved for t as before, giving us two points of intersection.

The cylinder is infinite; there is no z term.

Hitting things with rays: Planes and polygons

A planar polygon P can be defined as

$$\text{Polygon } P = \{v^1, \dots, v^n\}$$

which gives us the normal to P as

$$N = (v^n - v^1) \times (v^2 - v^1)$$

The equation for the plane of P is

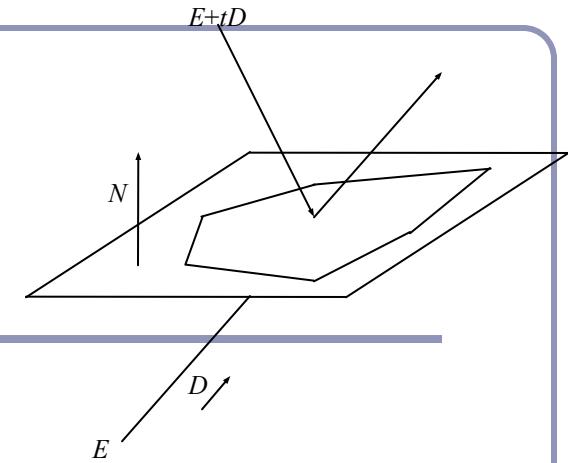
$$N \cdot (p - v^1) = 0 \quad (\zeta)$$

Substituting equation (α) into (ζ) for p yields

$$N \cdot (E + tD - v^1) = 0$$

$$x_N(x_E + tx_D - x_{v^1}) + y_N(y_E + ty_D - y_{v^1}) + z_N(z_E + tz_D - z_{v^1}) = 0$$

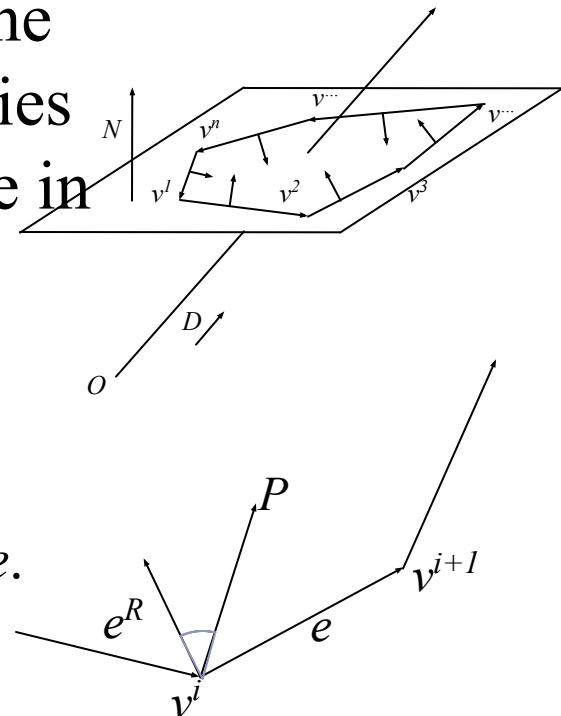
$$t = \frac{(N \cdot v^1) - (N \cdot E)}{N \cdot D}$$



Point in convex polygon

Half-planes method

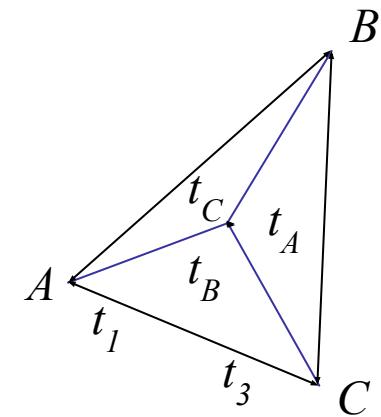
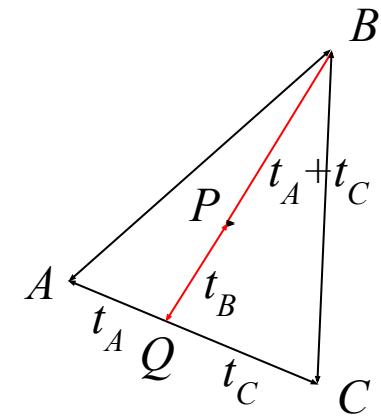
- Each edge defines an infinite half-plane covering the polygon. If the point P lies in all of the half-planes then it must be in the polygon.
- For each edge $e=v^i \rightarrow v^{i+1}$:
 - Rotate e by 90° CCW around N .
 - Do this quickly by crossing N with e .
 - If $e^R \cdot (P - v^i) < 0$ then the point is outside e .
- Fastest known method.



Barycentric coordinates

Barycentric coordinates (t_A, t_B, t_C) are a coordinate system for describing the location of a point P inside a triangle (A, B, C) .

- You can think of (t_A, t_B, t_C) as ‘masses’ placed at (A, B, C) respectively so that the center of gravity of the triangle lies at P .
- (t_A, t_B, t_C) are also proportional to the subtriangle areas.
 - The area of a triangle is $\frac{1}{2}$ the length of the cross product of two of its sides.



Point in nonconvex polygon

Winding number

- The *winding number* of a point P in a curve C is the number of times that the curve wraps around the point.
- For a simple closed curve (as any well-behaved polygon should be) this will be zero if the point is outside the curve, non-zero if it's inside.
- The winding number is the sum of the angles from v^i to P to v^{i+1} .
 - Caveat: This method is elegant but slow.

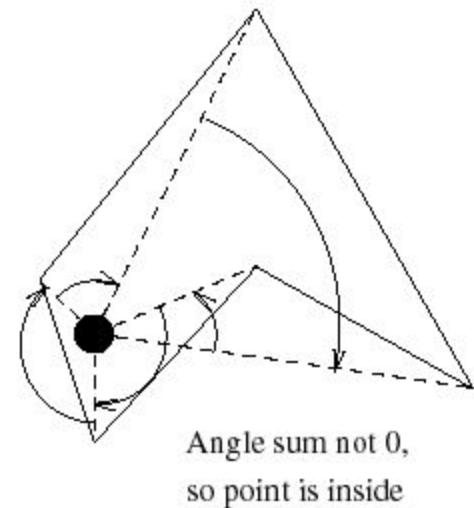
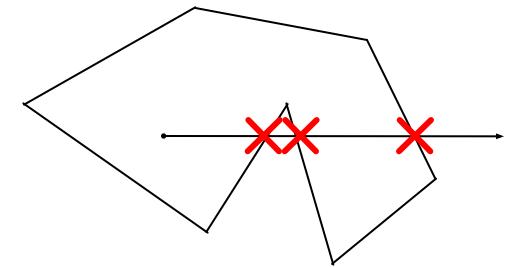


Figure from Eric Haines'
“Point in Polygon Strategies”,
Graphics Gems IV, 1994

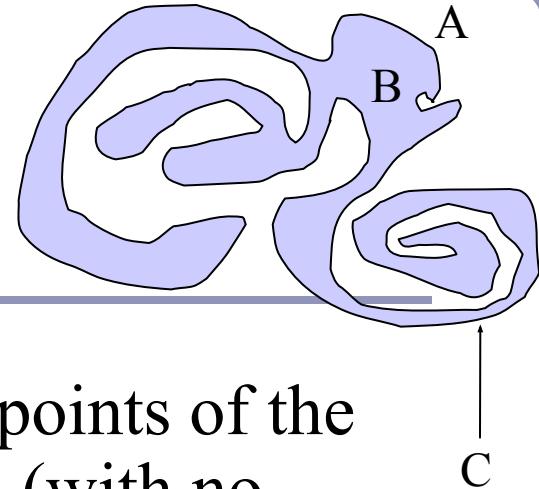
Point in nonconvex polygon

Ray casting (1974)

- Odd number of crossings = inside
- Issues:
 - How to find a point that you *know* is inside?
 - What if the ray hits a vertex?
 - Best accelerated by working in 2D
 - You could transform all vertices such that the coordinate system of the polygon has normal = Z axis...
 - Or, you could observe that crossings are invariant under scaling transforms and just project along any axis by ignoring (for example) the Z component.
- Validity proved by the *Jordan curve* theorem



The *Jordan curve theorem*



“Any simple closed curve C divides the points of the plane not on C into two distinct domains (with no points in common) of which C is the common boundary.”

- First stated (but proved incorrectly) by Camille Jordan (1838 -1922) in his *Cours d'Analyse*.

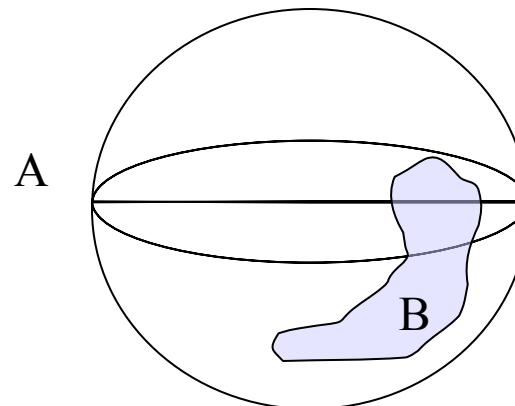
Sketch of proof : (For full proof see Courant & Robbins, 1941.)

- Show that any point in A can be joined to any other point in A by a path which does not cross C, and likewise for B.
- Show that any path connecting a point in A to a point in B *must* cross C.

The Jordan curve theorem on a sphere

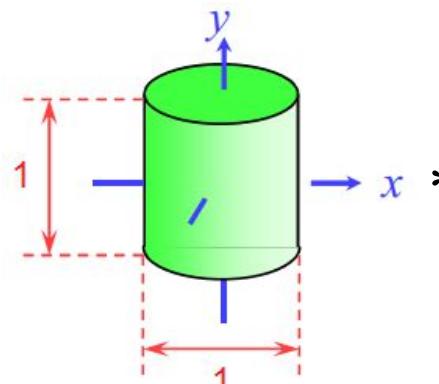
Note that the Jordan curve theorem can be extended to a curve on a sphere, or anything which is topologically equivalent to a sphere.

“Any simple closed curve on a sphere separates the surface of the sphere into two distinct regions.”



Local coordinates, world coordinates

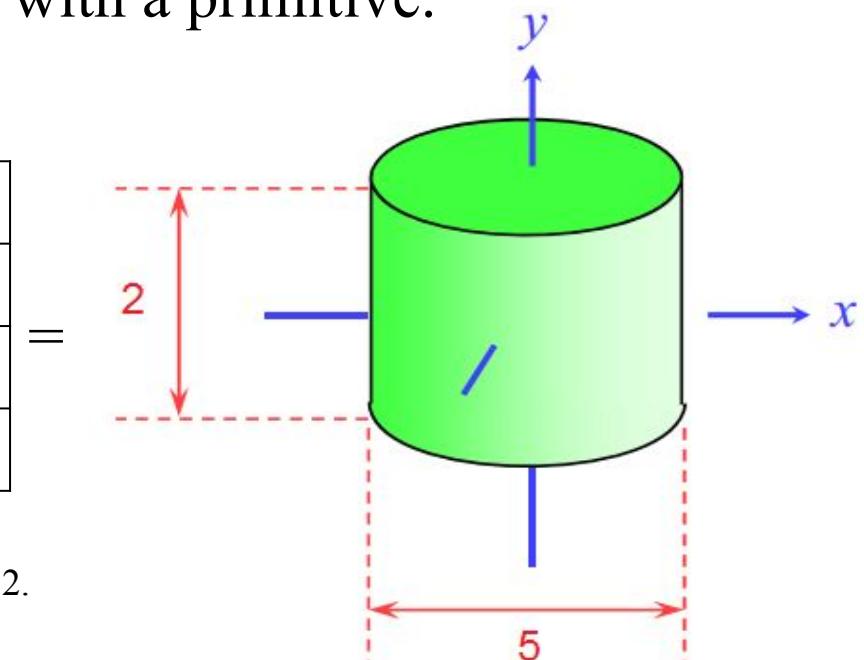
A very common technique in graphics is to associate a *local-to-world transform*, T , with a primitive.



The cylinder “as it sees itself”, in local coordinates

5	0	0	0
0	2	0	0
0	0	5	0
0	0	0	1

A 4×4 scale matrix, which multiplies x and z by 5, y by 2.



The cylinder “as the world sees it”, in world coordinates

Local coordinates, world coordinates: Transforming the ray

In order to test whether a ray hits a transformed object, we need to describe the ray in the object's *local coordinates*. We transform the ray by the *inverse of the local to world matrix*, T^{-1} .

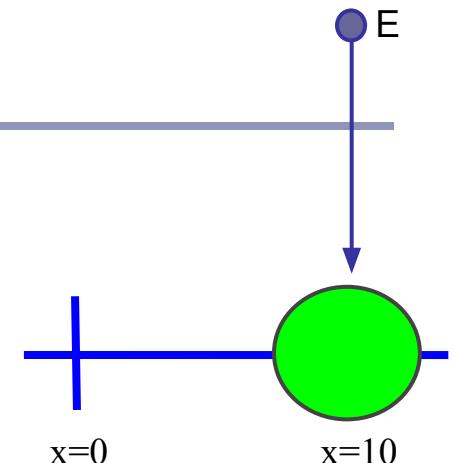
If the ray is defined by

$$P(t) = E + tD$$

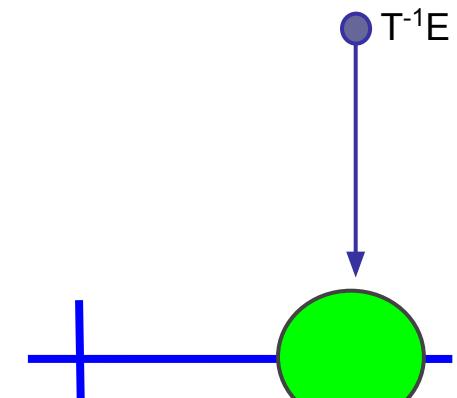
then the ray in local coordinates is defined by

$$T^{-1}(P(t)) = T^{-1}(E) + t(T^{-1}_{3 \times 3} D)$$

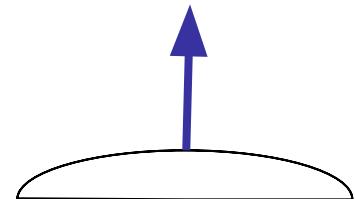
where $T^{-1}_{3 \times 3}$ is the top left 3×3 submatrix of T^{-1} .



World coordinates



Local coordinates



Finding the normal

We often need to know N , the *normal to the surface* at the point where a ray hits a primitive.

- If the ray R hits the primitive P at point X then N is...

<u>Primitive type</u>	<u>Equation for N</u>
Unit Sphere centered at the origin	$N = X$
Infinite Unit Cylinder centered at the origin	$N = [x_X \ y_X \ 0]$
Infinite Double Cone centered at the origin	$N = X \times (X \times [0, 0, z_X])$
Plane with normal n	$N = n$

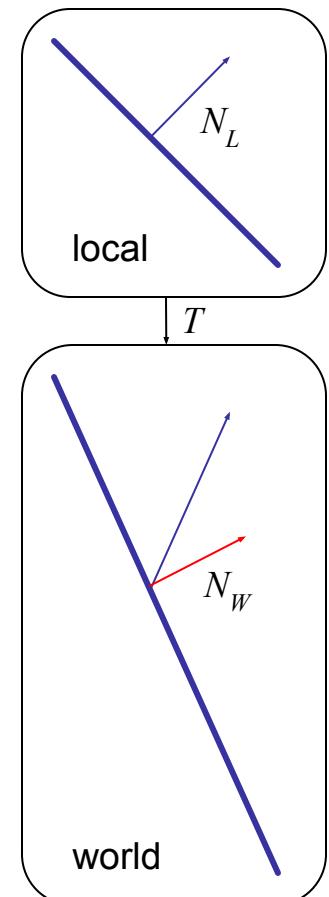
We use the normal for color, reflection, refraction, shadow rays...

Converting the normal from local to world coordinates

To find the world-coordinates normal N from the local-coordinates N_L , multiply N_L by the transpose of the inverse of the top left-hand 3×3 submatrix of T :

$$N = ((T_{3 \times 3})^{-1})^T N_L$$

- We want the top left 3×3 to discard translations
- For any rotation Q , $(Q^{-1})^T = Q$
- Scaling is unaffected by transpose, and a scale of (a, b, c) becomes $(1/a, 1/b, 1/c)$ when inverted



Local coordinates, world coordinates

Summary

To compute the intersection of a ray $R = E + tD$ with an object transformed by local-to-world transform T :

1. Compute R' , the ray R in local coordinates, as
$$P'(t) = T^{-1}(P(t)) = T^{-1}(E) + t(T^{-1}_{3\times 3}(D))$$
2. Perform your hit test in local coordinates.
3. Convert all hit points from local coordinates back to world coordinates by multiplying them by T .
4. Convert all hit normals from local coordinates back to world coordinates by multiplying them by $((T^{3\times 3})^{-1})^T$.

This will allow you to efficiently and quickly fire rays at arbitrarily-transformed primitive objects.

Your scene graph and you

Many 2D GUIs today favor an event model in which events ‘bubble up’ from child windows to parents. This is sometimes mirrored in a scene graph.

- Ex: a child changes size, changing the size of the parent’s bounding box
- Ex: the user drags a movable control in the scene, triggering an update event

If you do choose this approach, consider using the *Model View Controller* or *Model View Presenter* design pattern. 3D geometry objects are good for displaying data but they are not the proper place for control logic.

- For example, the class that stores the geometry of the rocket should not be the same class that stores the logic that moves the rocket.
- Always separate logic from representation.

Your scene graph and you

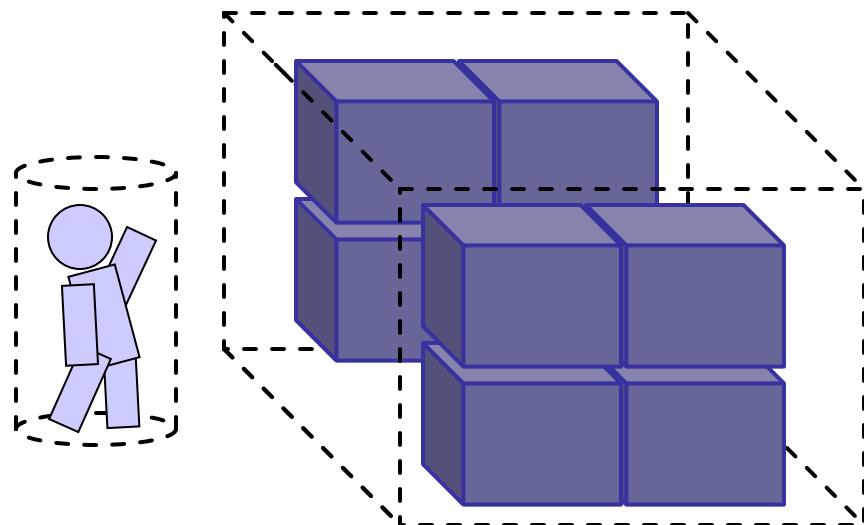
A common optimization derived from the scene graph is the propagation of *bounding volumes*.

Nested bounding volumes allow the rapid culling of large portions of geometry

- Test against the bounding volume of the top of the scene graph and then work down.

Great for...

- Collision detection between scene elements
- Culling before rendering
- Accelerating ray-tracing



Speed up ray-tracing with *bounding volumes*

Bounding volumes help to quickly accelerate volumetric tests, such as “does the ray hit the cow?”

- choose fast hit testing over accuracy
- ‘bboxes’ don’t have to be tight

Axis-aligned bounding boxes

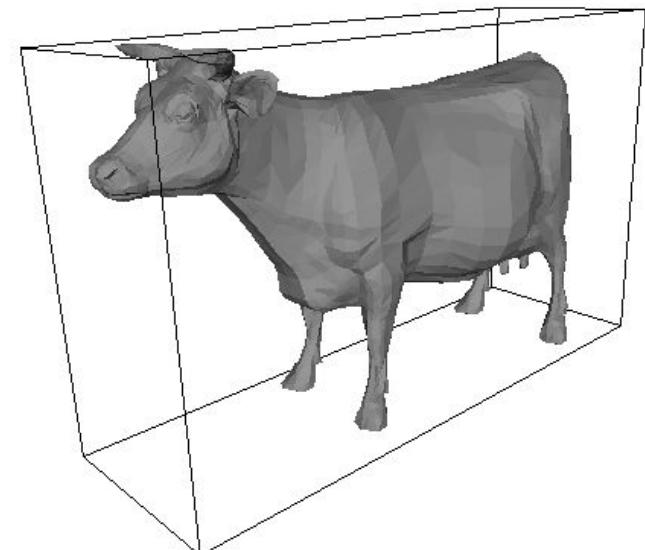
- max and min of x/y/z.

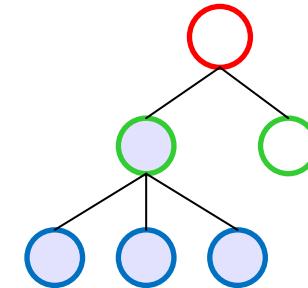
Bounding spheres

- max of radius from some rough center

Bounding cylinders

- common in early FPS games

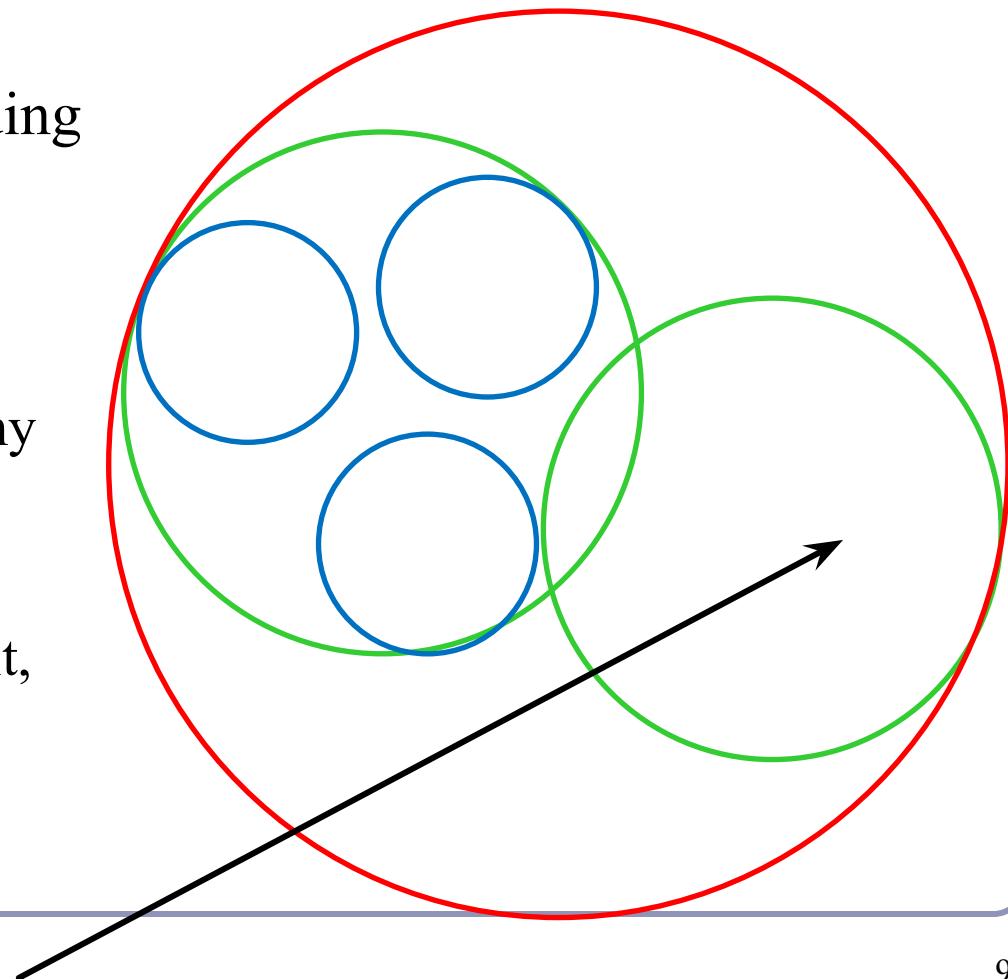




Bounding volumes in hierarchy

Hierarchies of bounding volumes allow early discarding of rays that won't hit large parts of the scene.

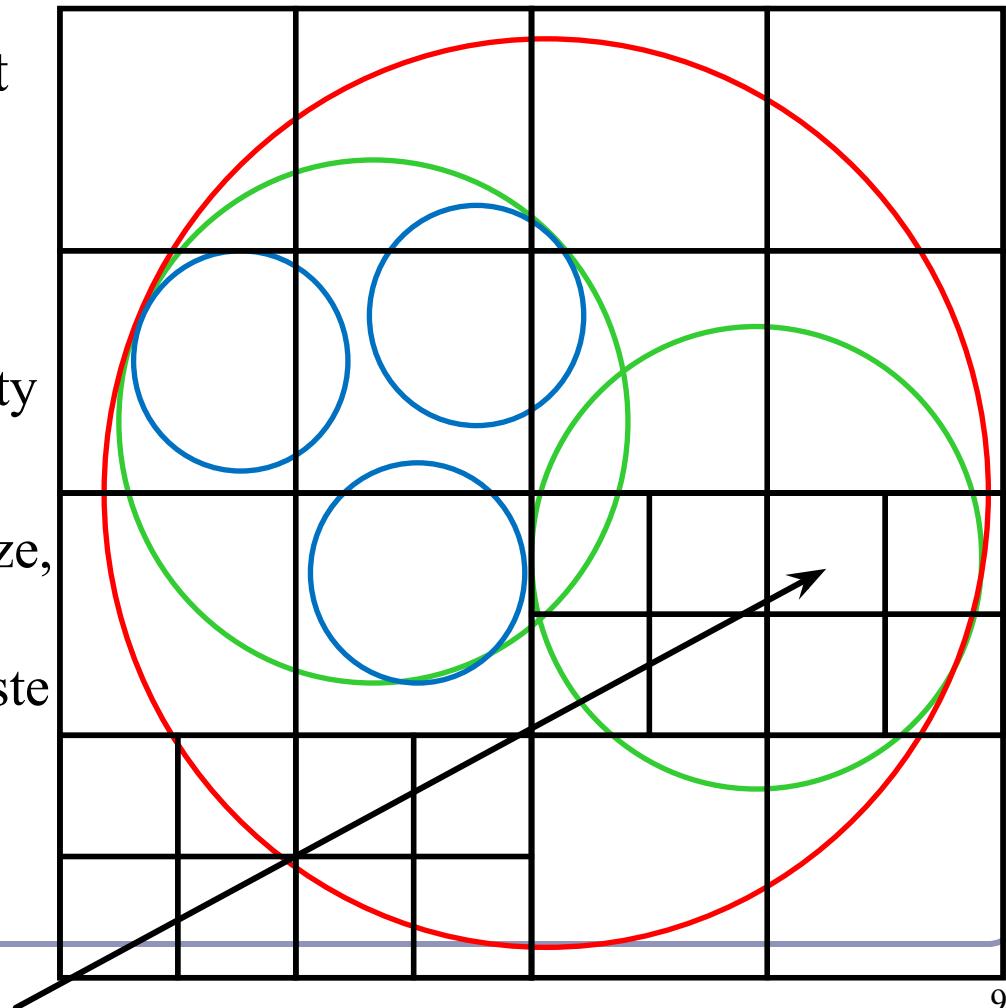
- Pro: Rays can skip subsections of the hierarchy
- Con: Without spatial coherence ordering the objects in a volume you hit, you'll still have to hit-test every object



Subdivision of space

Split space into cells and list in each cell every object in the scene that overlaps that cell.

- Pro: The ray can skip empty cells
- Con: Depending on cell size, objects may overlap many filled cells or you may waste memory on many empty cells



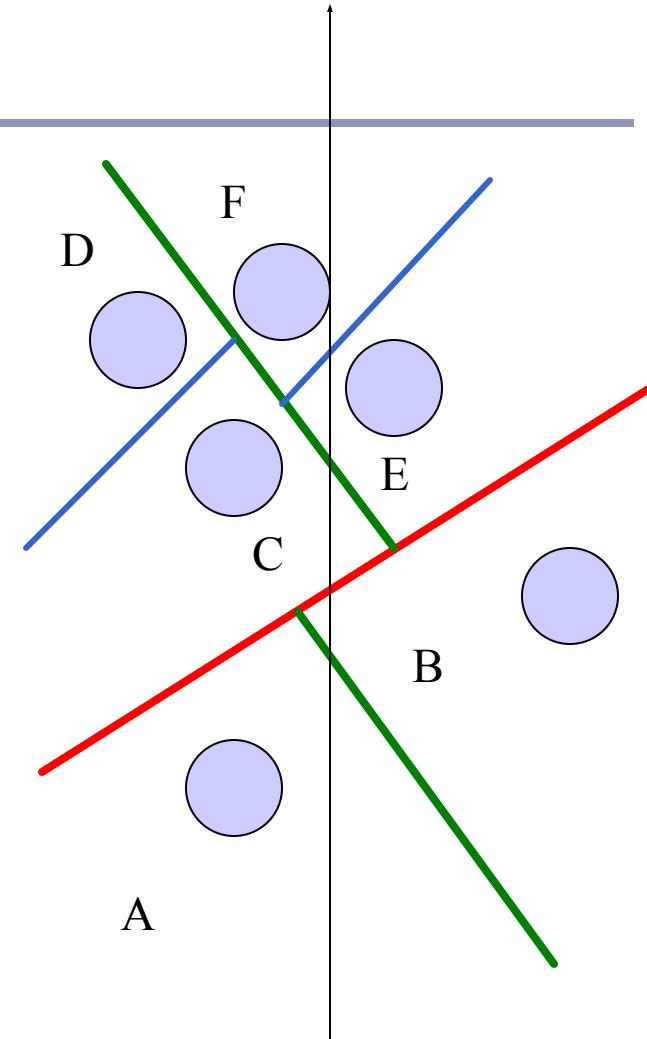
Popular acceleration structures: BSP Trees

The *BSP tree* partitions the scene into objects in front of, on, and behind a tree of planes.

- When you fire a ray into the scene, you test all near-side objects before testing far-side objects.

Problems:

- choice of planes is not obvious
- computation is slow
- plane intersection tests are heavy on floating-point math.



Popular acceleration structures: *kd-trees*

The *kd-tree* is a simplification of the BSP Tree data structure

- Space is recursively subdivided by axis-aligned planes and points on either side of each plane are separated in the tree.
- The *kd-tree* has $O(n \log n)$ insertion time (but this is very optimizable by domain knowledge) and $O(n^{2/3})$ search time.
- *kd-trees* don't suffer from the mathematical slowdowns of BSPs because their planes are always axis-aligned.

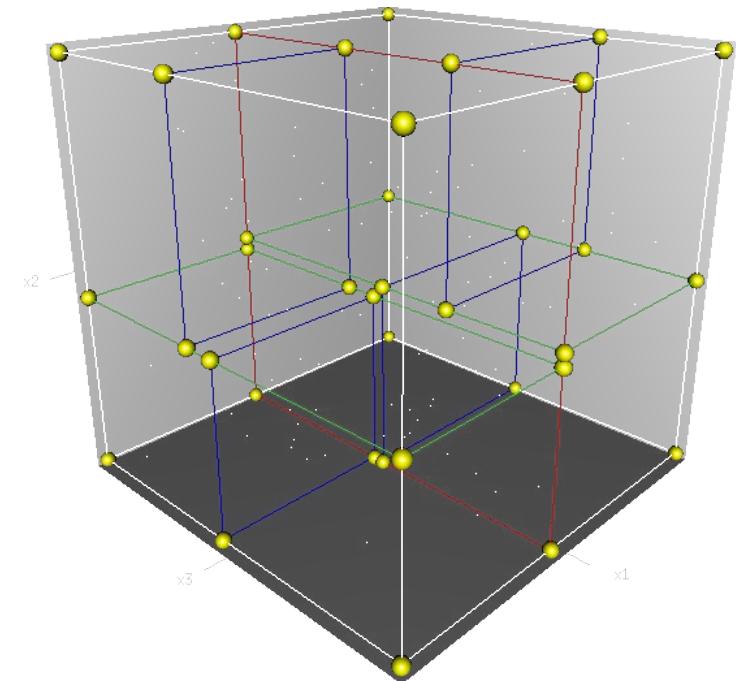


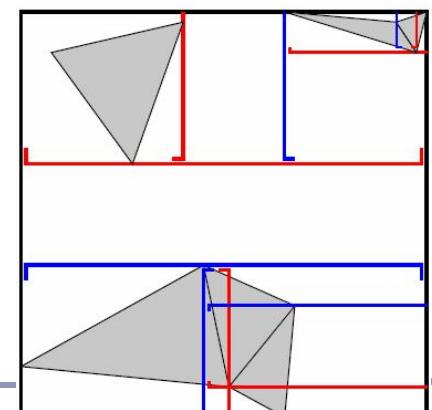
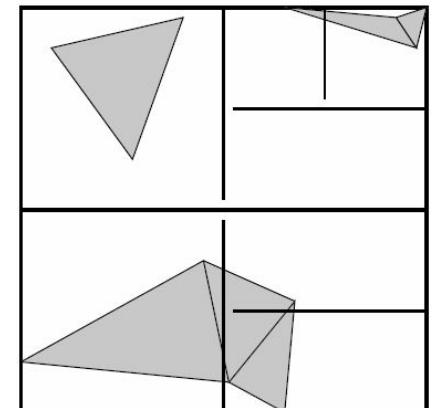
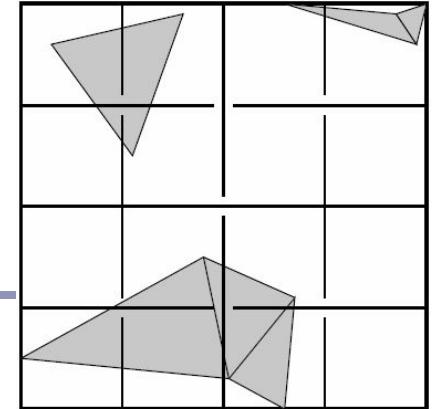
Image from Wikipedia, bless their hearts.

Popular acceleration structures: *Bounding Interval Hierarchies*

The *Bounding Interval Hierarchy* subdivides space around the volumes of objects and shrinks each volume to remove unused space.

- Think of this as a “best-fit” kd -tree
- Can be built dynamically as each ray is fired into the scene

Image from Wächter and Keller’s paper,
Instant Ray Tracing: The Bounding Interval Hierarchy, Eurographics (2006)



References

Jordan curves

R. Courant, H. Robbins, *What is Mathematics?*, Oxford University Press, 1941

<http://cgm.cs.mcgill.ca/~godfried/teaching/cg-projects/97/Octavian/compgeom.html>

Intersection testing

<http://www.realtimerendering.com/intersections.html>

<http://tog.acm.org/editors/erich/ptinpoly>

<http://mathworld.wolfram.com/BarycentricCoordinates.html>

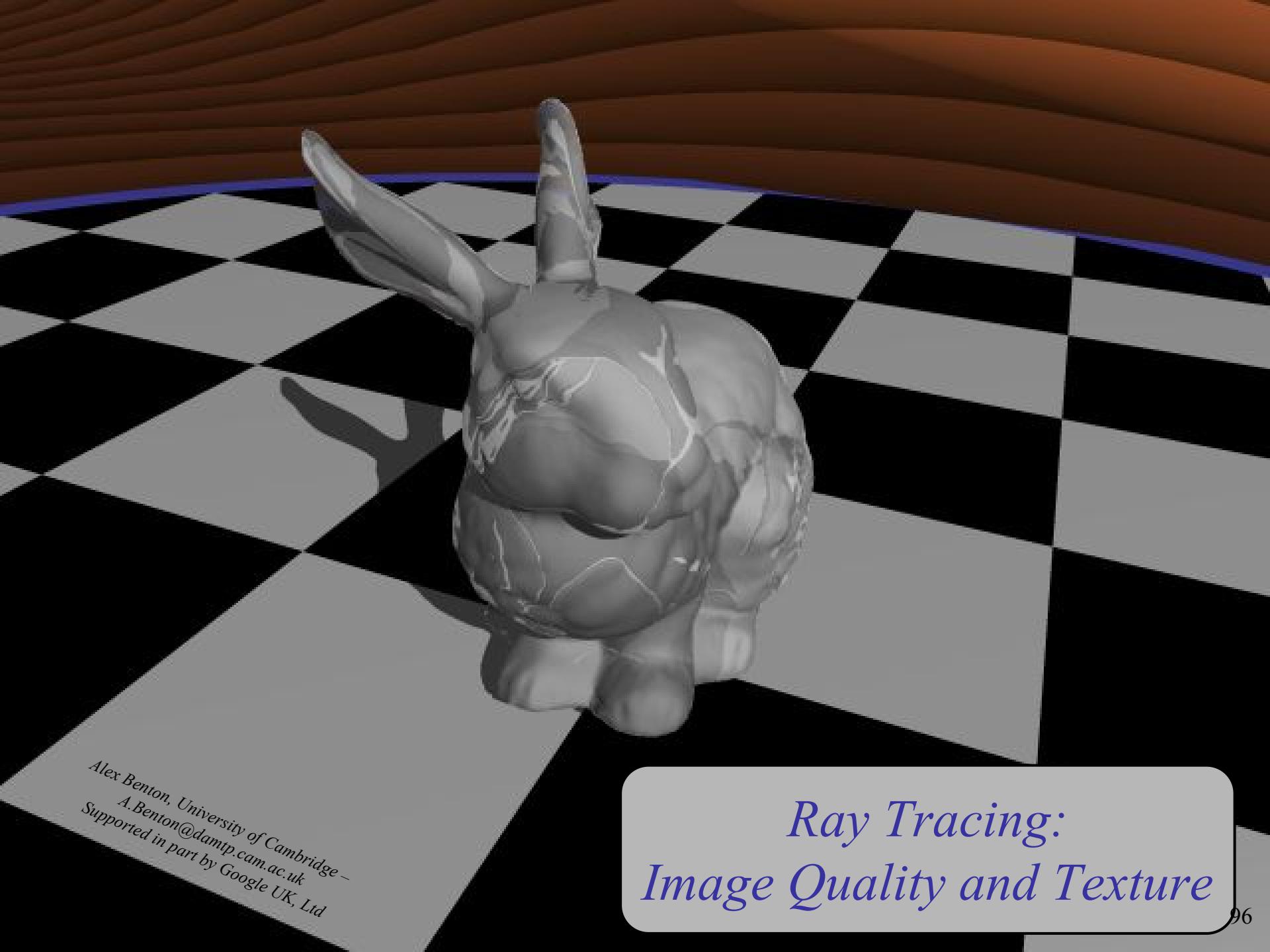
Ray tracing

Foley & van Dam, *Computer Graphics* (1995)

Jon Genetti and Dan Gordon, *Ray Tracing With Adaptive Supersampling in Object Space*,
<http://www.cs.uaf.edu/~genetti/Research/Papers/GI93/GI.html> (1993)

Zack Waters, “Realistic Raytracing”,

http://web.cs.wpi.edu/~emmanuel/courses/cs563/write_ups/zackw/realistic_raytracing.html



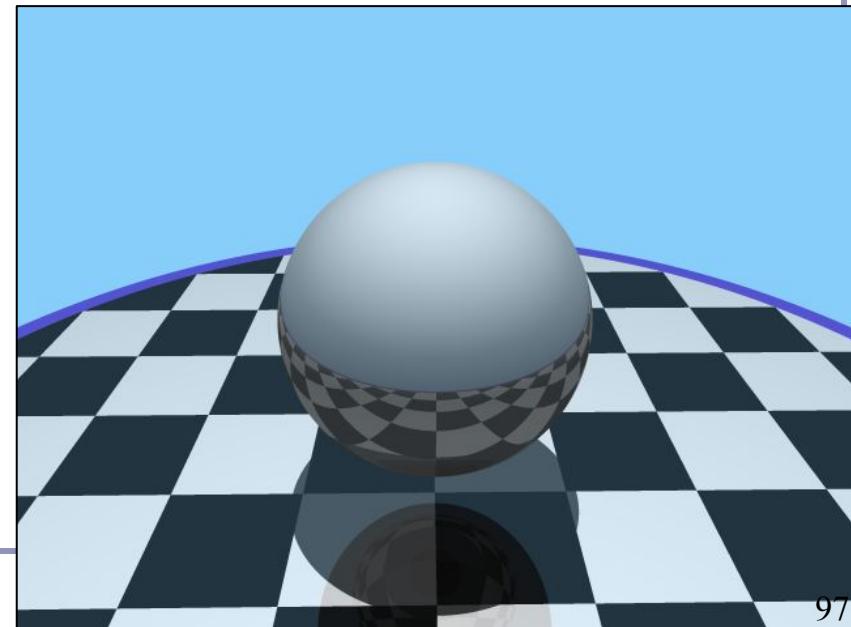
Alex Benton, University of Cambridge –
A.Benton@damtp.cam.ac.uk
Supported in part by Google UK, Ltd

*Ray Tracing:
Image Quality and Texture*

Shadows

To simulate shadows in ray tracing, fire a ray from P towards each light L_i . If the ray hits another object before the light, then discard L_i in the sum.

- This is a boolean removal, so it will give hard-edged shadows.
- Hard-edged shadows suggest a pinpoint light source.



Softer shadows

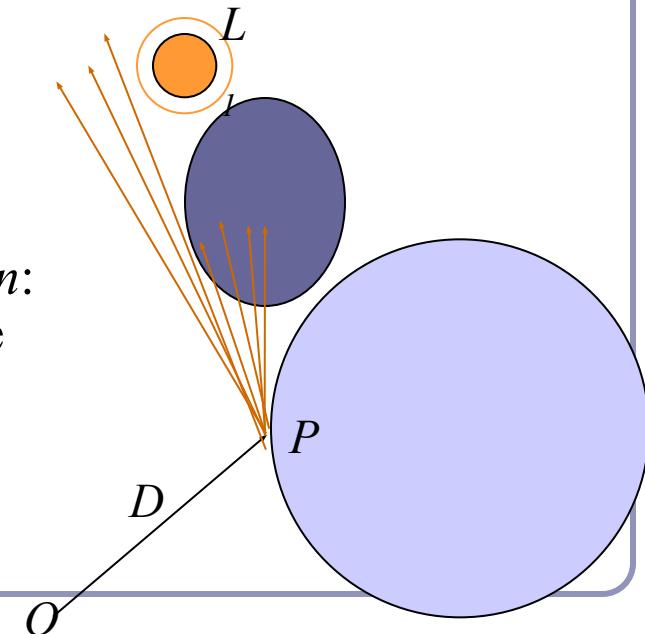
Shadows in nature are not sharp because light sources are not infinitely small.

- Also because light scatters, etc.

For lights with volume, fire many rays, covering the cross-section of your illuminated space.

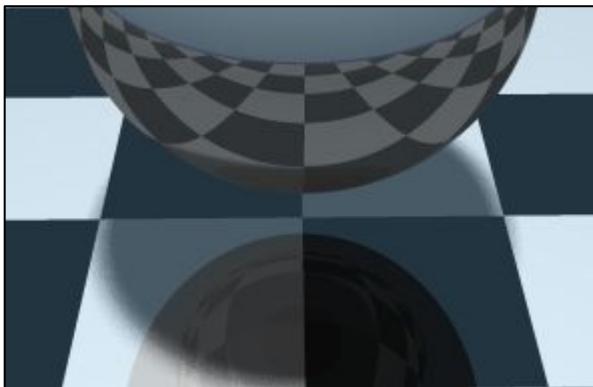
Illumination is scaled by (the total number of rays that aren't blocked) divided by (the total number of rays fired).

- This is an example of *Monte-Carlo integration*: a coarse simulation of an integral over a space by randomly sampling it with many rays.
- The more rays fired, the smoother the result.

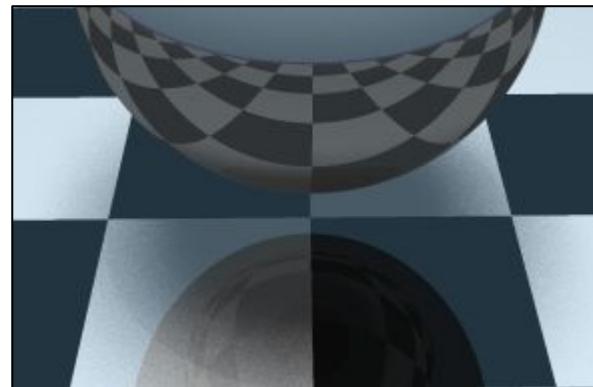


Softer shadows

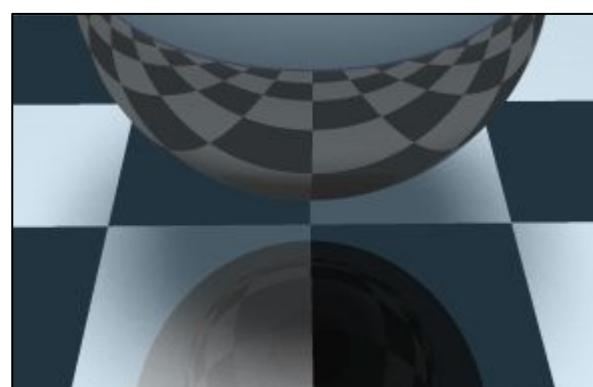
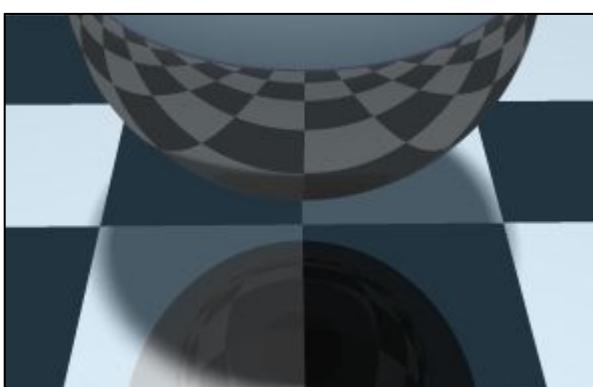
Rays per shadow test: 20



Light radius: 5



Rays per shadow test: 100

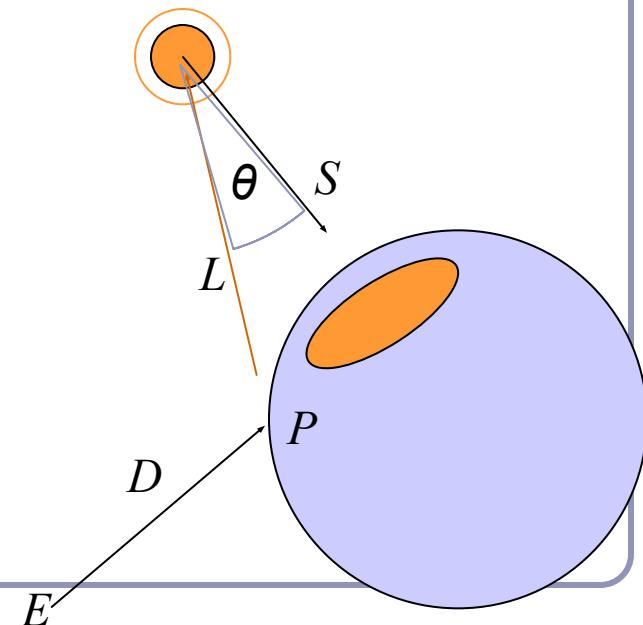


All images anti-aliased with 4x supersampling
Distance to light in all images: 20 units

Raytraced spotlights

To create a spotlight shining along axis S , you can multiply the (diffuse+specular) term by $(\max(L \cdot S, 0))^m$.

- Raising m will tighten the spotlight, but leave the edges soft.
- If you'd prefer a hard-edged spotlight of uniform internal intensity, you can use a conditional, e.g. $((L \cdot S > \cos(15^\circ)) ? 1 : 0)$.

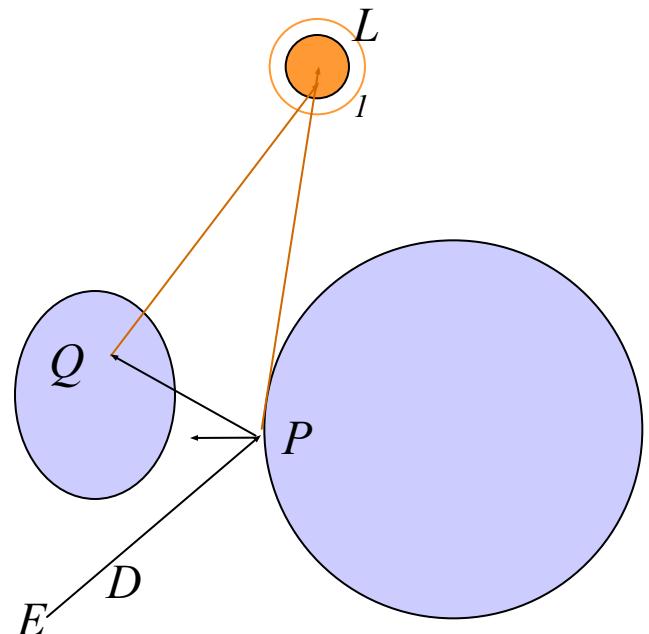


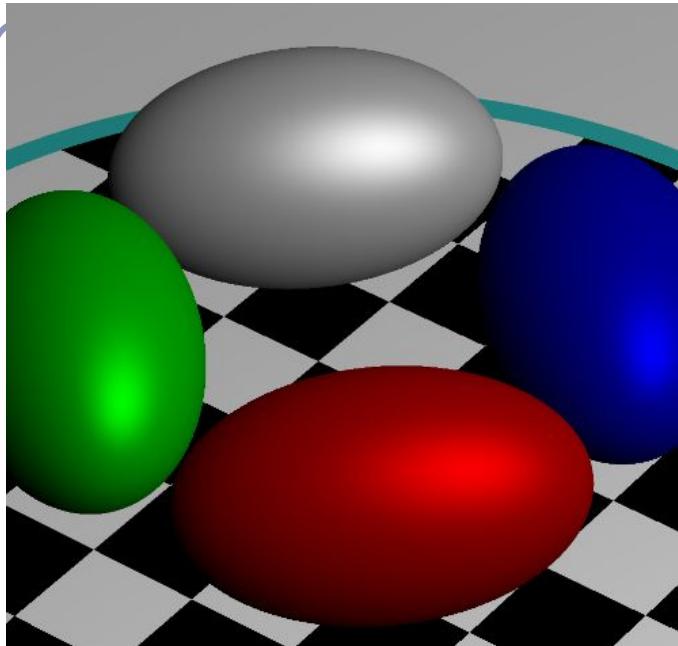
Reflection

Reflection rays are calculated as:

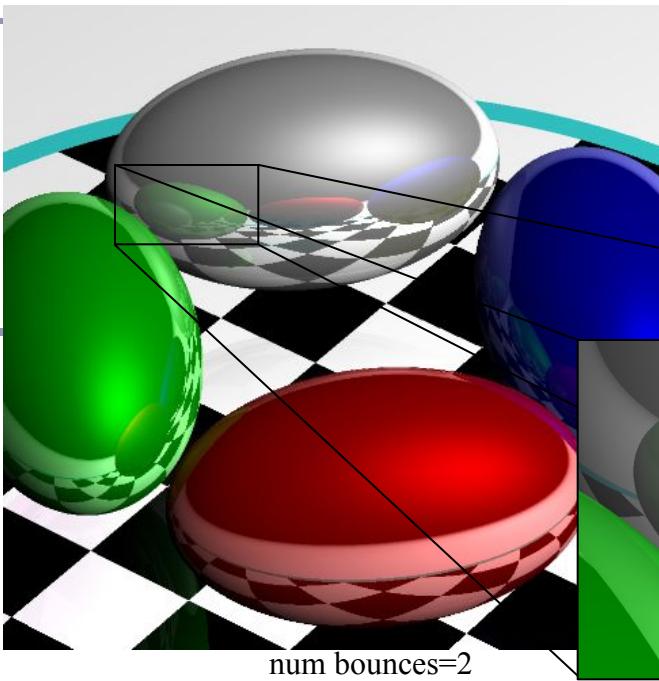
$$R = 2(-D \bullet N)N + D$$

- Finding the reflected color is a recursive raycast.
- Reflection has *scene-dependant* performance impact.
- If you're using the GPU, GLSL supports `reflect()` as a built-in function.

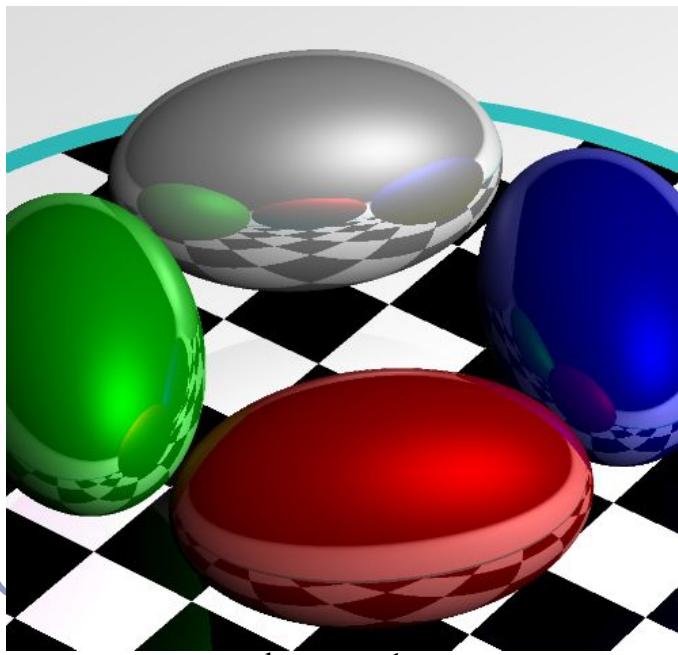




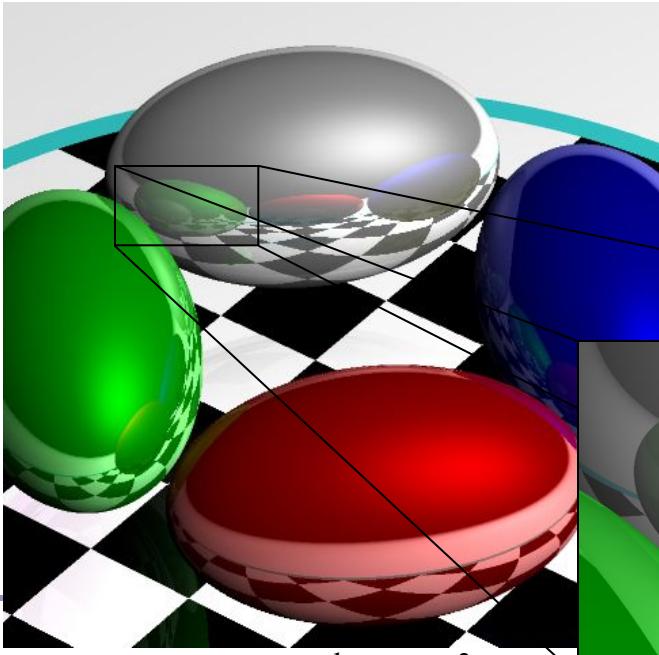
num bounces=0



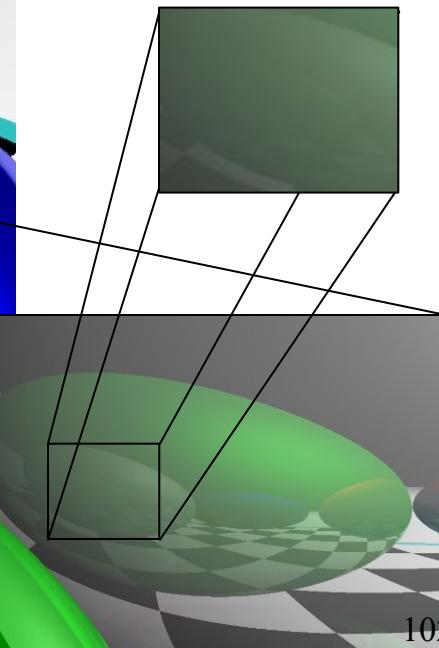
num bounces=2



num bounces=1



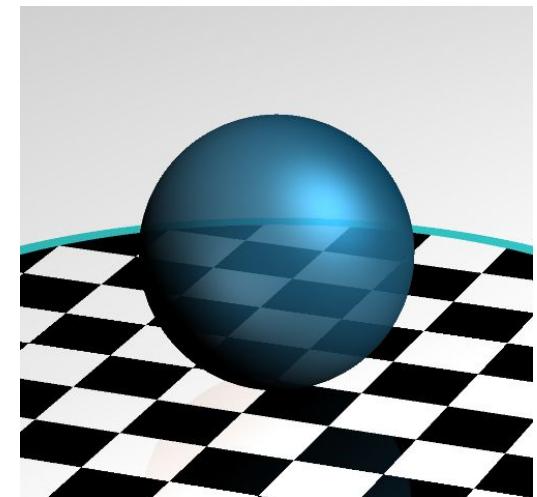
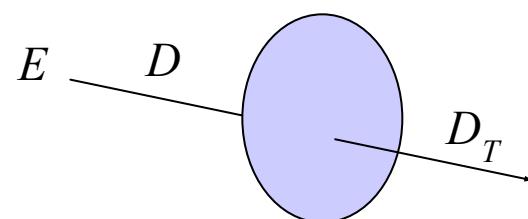
num bounces=3



Transparency

To add transparency, generate and trace a new *transparency ray* with $E_T = P$, $D_T = D$.

To support this in software, make color a 1×4 vector where the fourth component, ‘alpha’, determines the weight of the recursed transparency ray.



Refraction

The *angle of incidence* of a ray of light where it strikes a surface is the acute angle between the ray and the surface normal.

The *refractive index* of a material is a measure of how much the speed of light¹ is reduced inside the material.

- The refractive index of air is about 1.003.
- The refractive index of water is about 1.33.

¹ Or sound waves or other waves¹⁰⁴

Refraction

Snell's Law:

$$\frac{\sin \theta_1}{\sin \theta_2} = \frac{n_2}{n_1} = \frac{v_1}{v_2}$$

“The ratio of the sines of the *angles of incidence* of a ray of light at the interface between two materials is equal to the inverse ratio of the *refractive indices* of the materials is equal to the ratio of the speeds of light in the materials.”

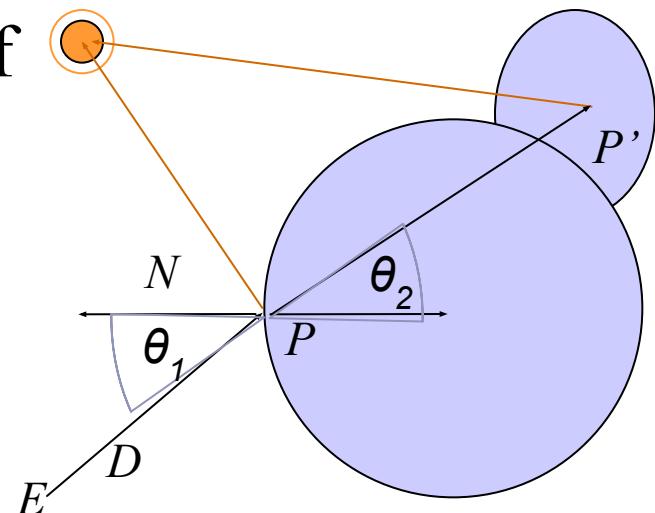
Historical note: this formula has been attributed to Willebrord Snell (1591-1626) and René Descartes (1596-1650) but first discovery goes to Ibn Sahl (940-1000) of Baghdad.

Refraction in ray tracing

$$\theta_1 = \cos^{-1}(N \bullet D)$$

$$\frac{\sin \theta_1}{\sin \theta_2} = \frac{n_2}{n_1} \rightarrow \theta_2 = \sin^{-1}\left(\frac{n_1}{n_2} \sin \theta_1\right)$$

Using Snell's Law and the angle of incidence of the incoming ray, we can calculate the angle from the negative normal to the outbound ray.



Refraction in ray tracing

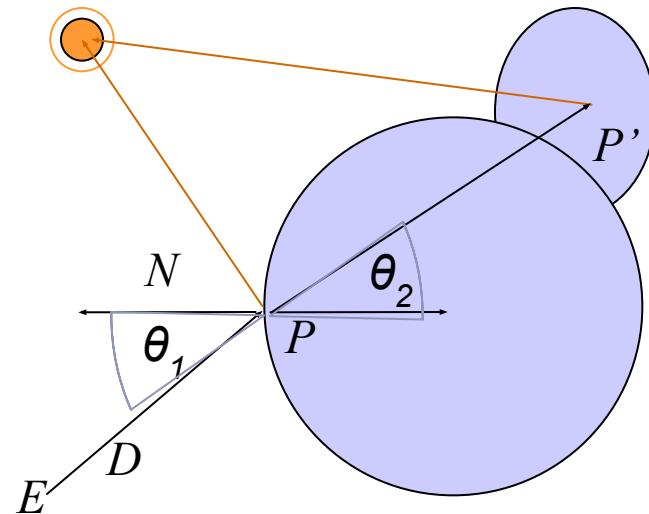
What if the arcsin parameter is > 1 ?

- Remember, arcsin is defined in $[-1,1]$.
- We call this the *angle of total internal reflection*: light is trapped completely inside the surface.

Total internal reflection



$$\theta_2 = \sin^{-1}\left(\frac{n_1}{n_2} \sin \theta_1\right)$$



Aliasing

aliasing

/'eɪliəsɪŋ/

noun: **aliasing**

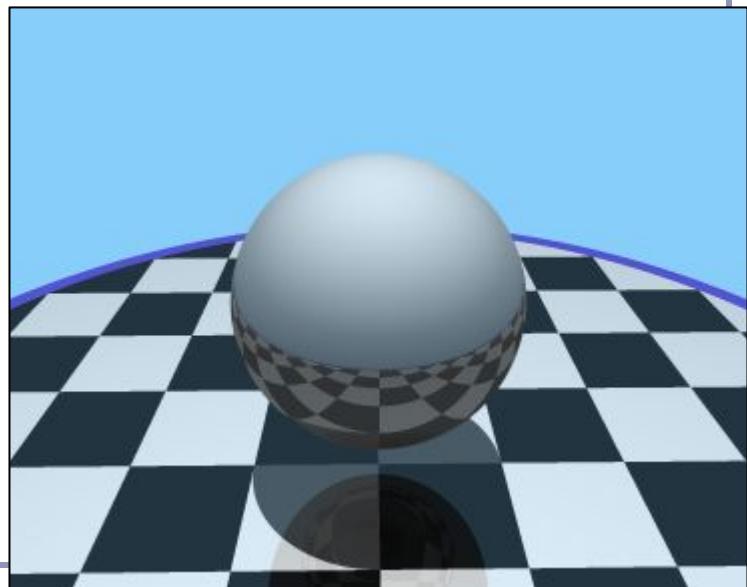
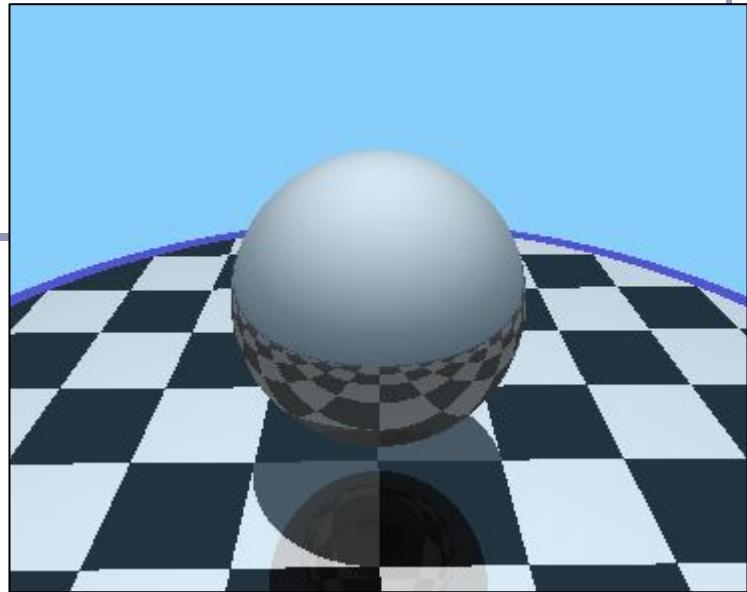
1. PHYSICS / TELECOMMUNICATIONS

the misidentification of a signal frequency,
introducing distortion or error.

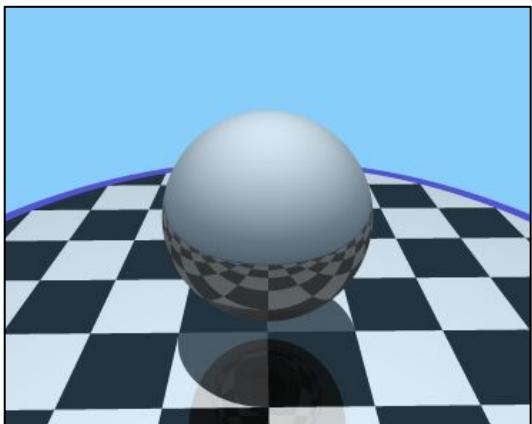
"high-frequency sounds are prone to aliasing"

2. COMPUTING

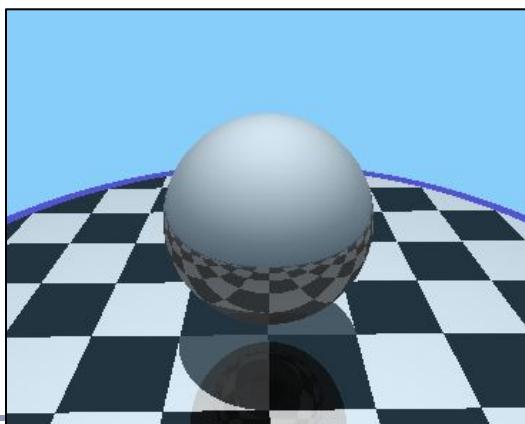
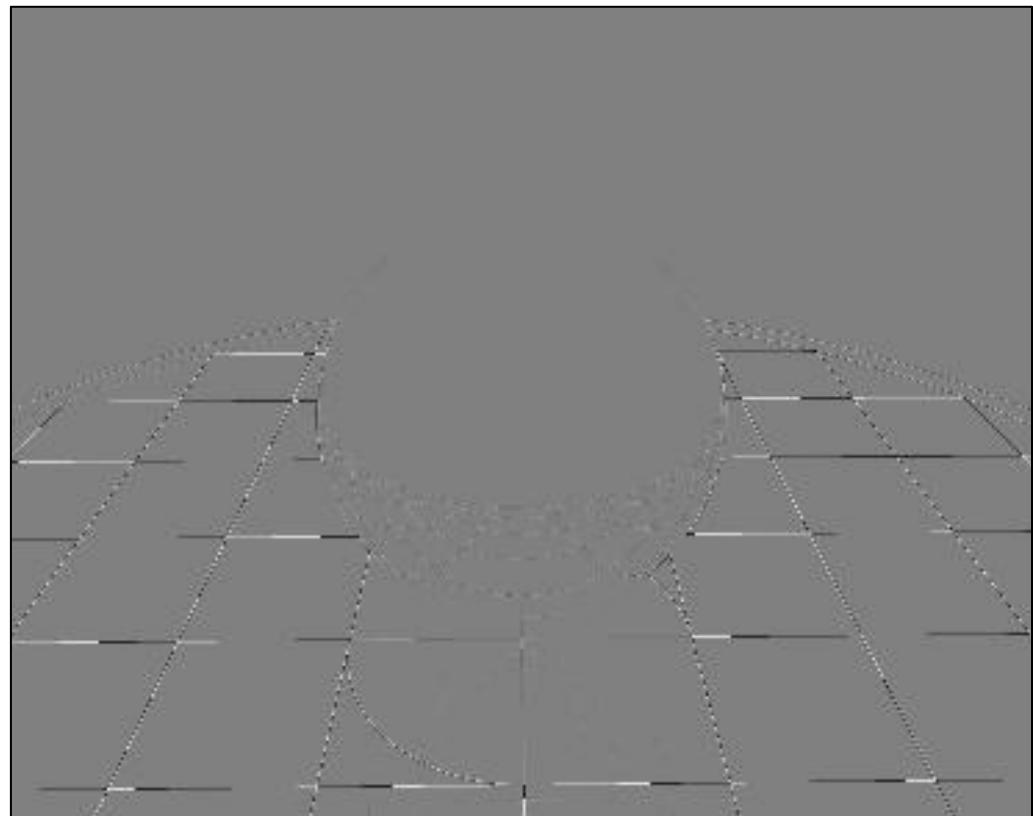
the distortion of a reproduced image so that
curved or inclined lines appear
inappropriately jagged, caused by the
mapping of a number of points to the same
pixel.



Aliasing



-



=

Anti-aliasing

Fundamentally, the problem with aliasing is that we're sampling an infinitely continuous function (the color of the scene) with a finite, discrete function (the pixels of the image).

One solution to this is *super-sampling*. If we fire multiple rays through each pixel, we can average the colors computed for every ray together to a single blended color.

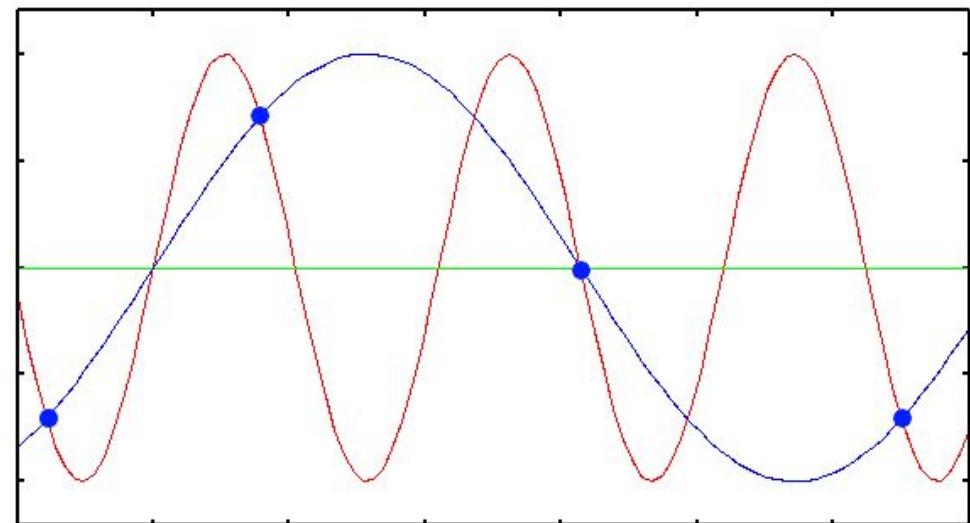
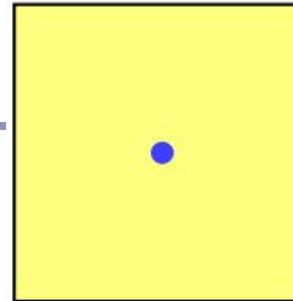


Image source: www.svi.nl

Anti-aliasing

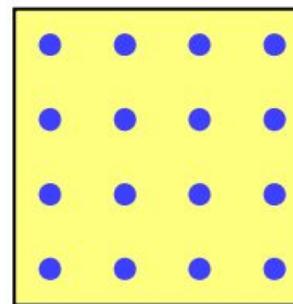
Single point

- Fire a single ray through the pixel's center



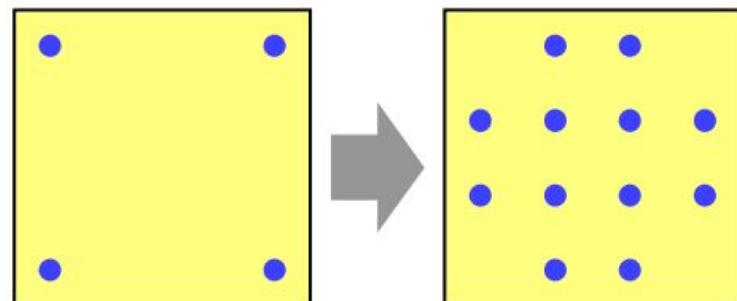
Super-sampling

- Fire multiple rays through the pixel and average the result
- Regular grid, random, jittered, Poisson disks



Adaptive super-sampling

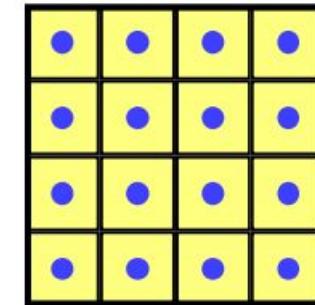
- Fire a few rays through the pixel, check the variance of the resulting values, if similar enough then stop else fire more rays



Types of super-sampling

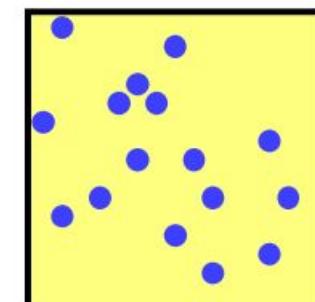
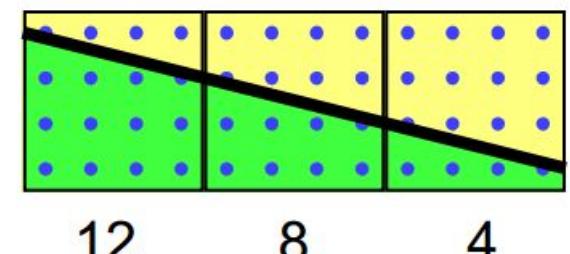
Regular grid

- Divide the pixel into a number of sub-pixels and fire a ray through the center of each
- This can still lead to noticeable aliasing unless a very high resolution of sub-pixel grid is used



Random

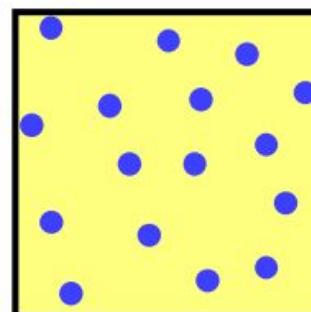
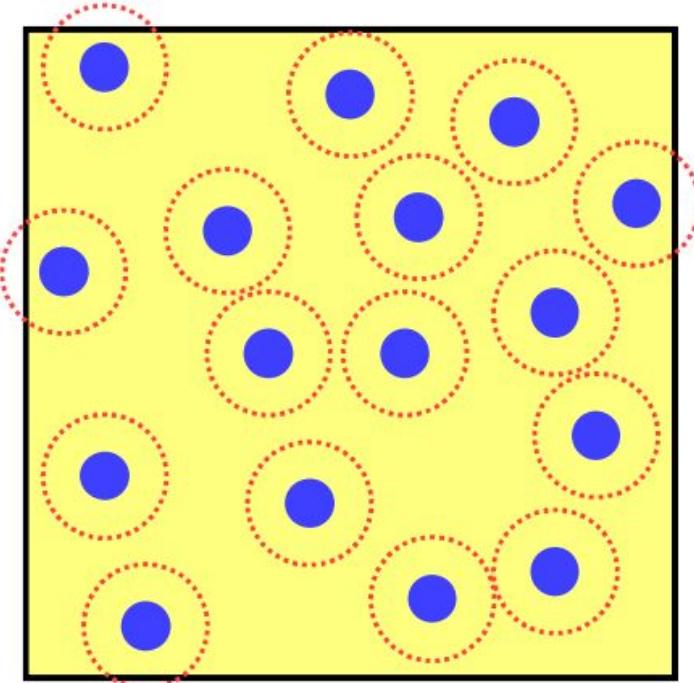
- Fire N rays at random points in the pixel
- Replaces aliasing artifacts with noise artifacts
 - But the human eye is much less sensitive to noise than to aliasing
- Requires special treatment for animation



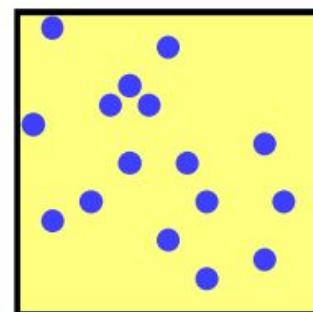
Types of super-sampling

Poisson disk

- Fire N rays at random points in the pixel, with the proviso that no two rays shall pass through the pixel closer than ε to one another
- For N rays this produces a better looking image than pure random sampling
- However, can be very hard to implement correctly / quickly



Poisson disk

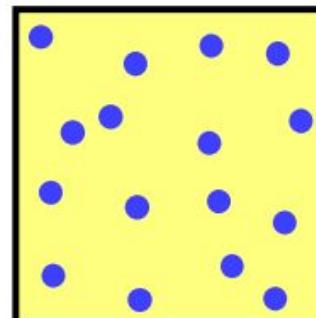
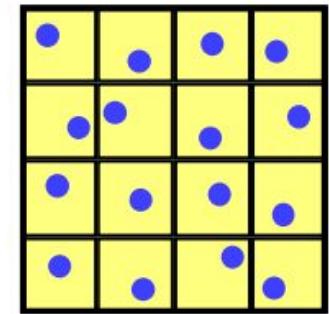


pure random

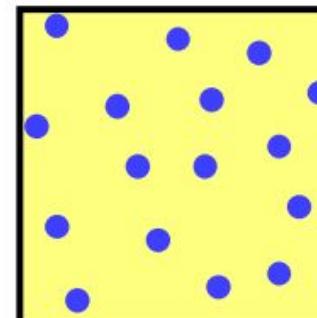
Types of super-sampling

Jittered

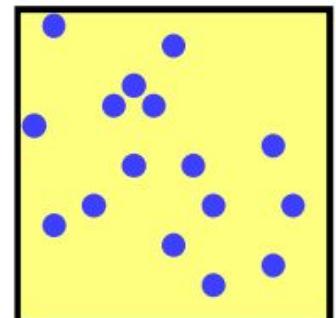
- Divide the pixel into N sub-pixels and fire one ray at a random point in each sub-pixel
- Approximates the Poisson disk behavior
- Better than pure random sampling, easier (and significantly faster) to implement than Poisson



jittered



Poisson disk



pure random

Applications of super-sampling

- Anti-aliasing
- Soft shadows
- Depth-of-field camera effects
(fixed focal depth, finite aperture)

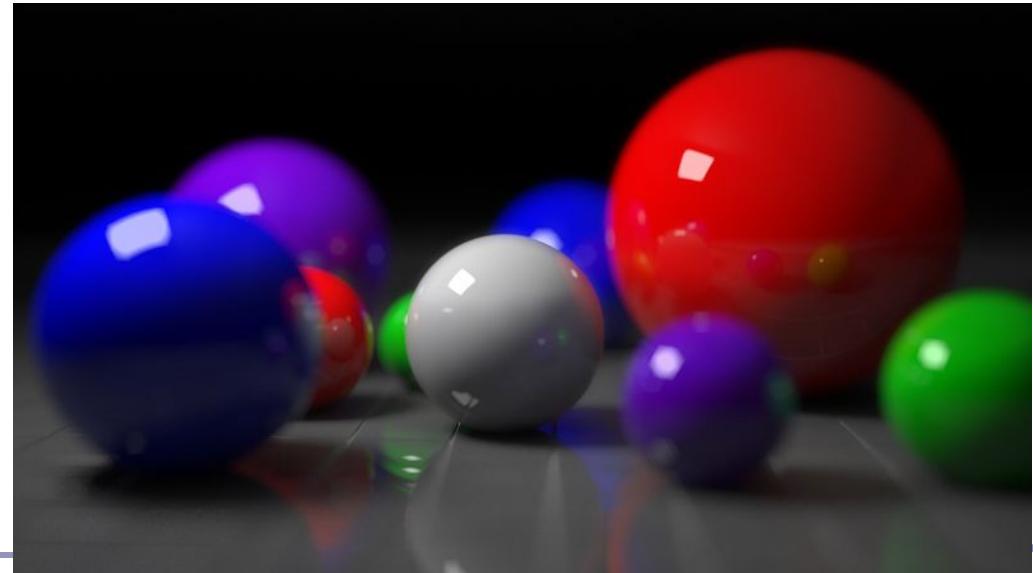
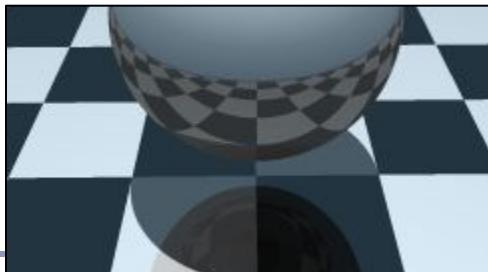
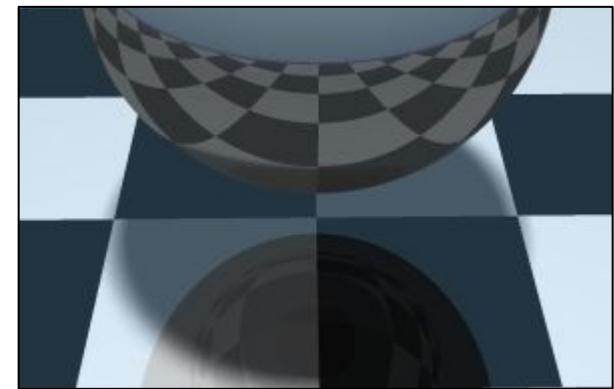
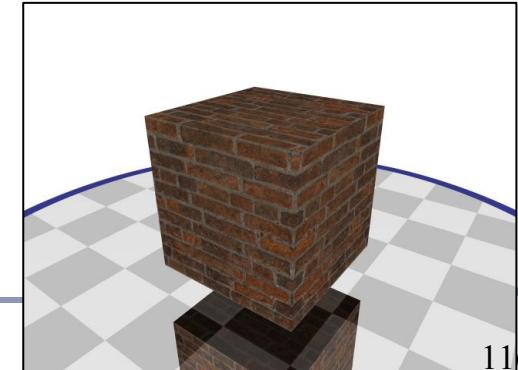
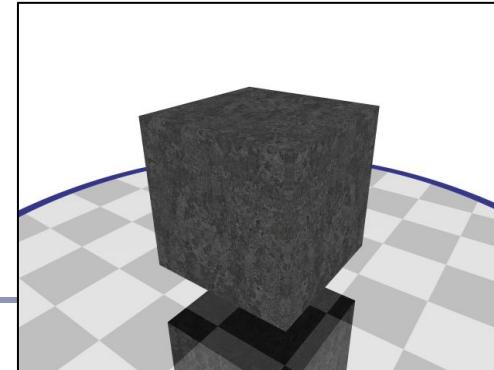
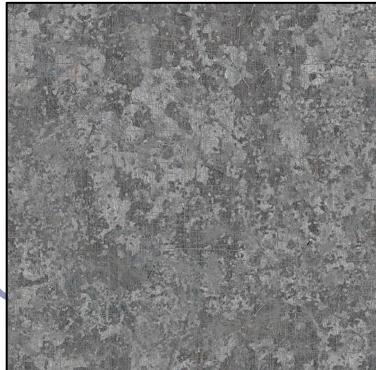


Image credit:
[http://en.wikipedia.org/wiki/Ray_tracing_\(graphics\)](http://en.wikipedia.org/wiki/Ray_tracing_(graphics))

Texture mapping

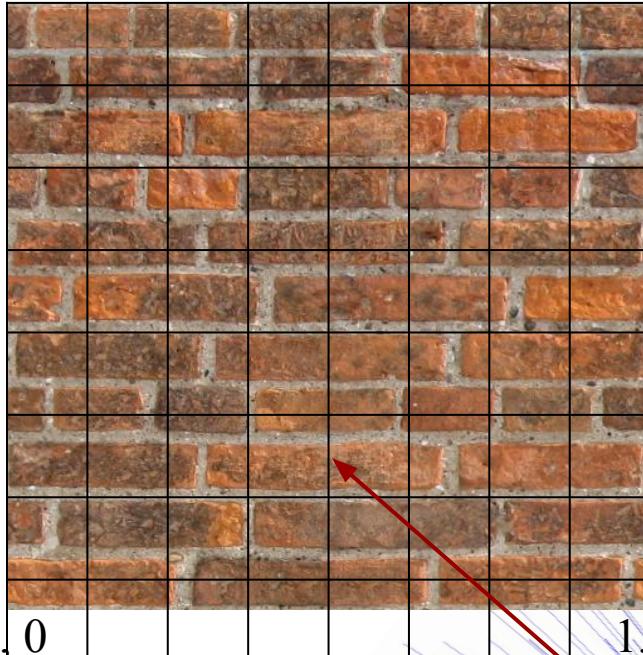
As observed in last year's course, real-life objects rarely consist of perfectly smooth, uniformly colored surfaces.

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.



Texture mapping

0, 1 1, 1



We'll need to query the color of the texture at the point in 3D space where the ray hits our surface. This is typically done by mapping
(3D point in local coordinates)
→ U,V coordinates bounded [0-1, 0-1]
→ Texture coordinates bounded by
[image width, image height]



UV mapping the primitives

UV mapping of a unit cube

if $|x| == 1$:

$$u = (z + 1) / 2$$

$$v = (y + 1) / 2$$

elif $|y| == 1$:

$$u = (x + 1) / 2$$

$$v = (z + 1) / 2$$

else:

$$u = (x + 1) / 2$$

$$v = (y + 1) / 2$$

UV mapping of a unit sphere

$$u = 0.5 + \text{atan2}(z, x) / 2\pi$$

$$v = 0.5 - \text{asin}(y) / \pi$$

UV mapping of a torus of
major radius R

$$u = 0.5 + \text{atan2}(z, x) / 2\pi$$

$$v = 0.5 + \text{atan2}(y, ((x^2 + z^2)^{1/2} - R) / 2\pi$$

UV mapping is easy for primitives but can be very difficult for arbitrary shapes.

Texture mapping

One constraint on using images for texture is that images have a finite resolution, and a virtual (ray-traced) camera can get quite near to the surface of an object.

This can lead to a single image pixel covering multiple ray-traced pixels (or vice-versa), leading to blurry or aliased pixels in your texture.



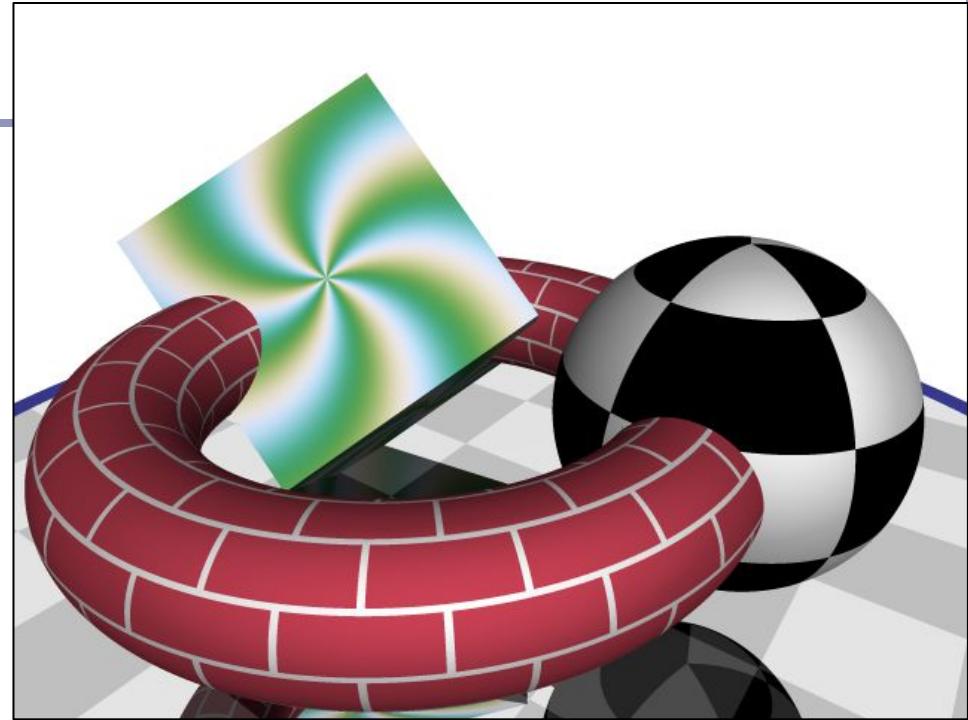
Procedural texture

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):

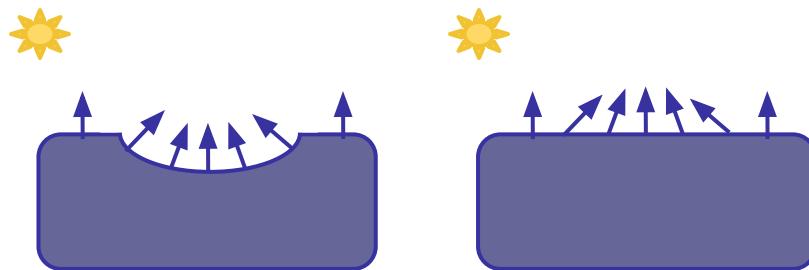
```
tx = (int) 10 * u  
ty = (int) 10 * v  
oddity = (tx & 0x01) == (ty & 0x01)  
edge = ((10 * u - tx < 0.1) && oddity) || (10 * v - ty < 0.1)  
return edge ? WHITE : RED
```



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

Non-color textures: normal mapping

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

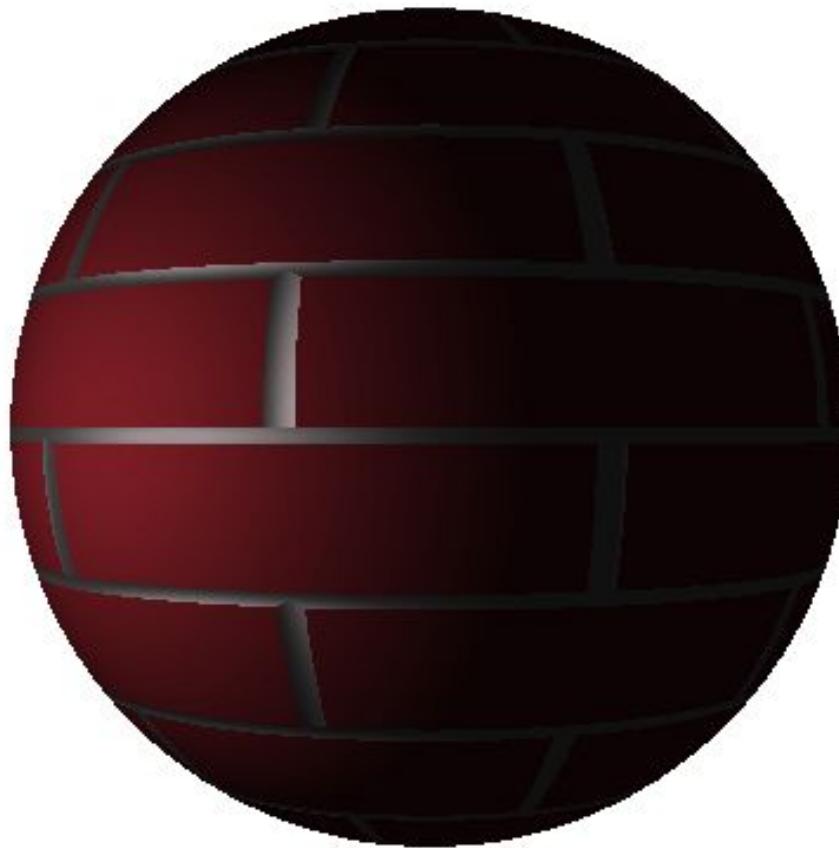


The specular and diffuse shading of the surface varies with the normals in a dent on the surface.

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

In a sense, the ray tracer computes a *trompe-l'oeil* image on the fly and ‘paints’ the surface with more detail than is actually present in the geometry.

Non-color textures: normal mapping



Anisotropic shading

Anisotropic shading occurs in nature when light reflects off a surface differently in one direction from another, as a function of the surface itself. The specular component is modified by the direction of the light.



Procedural volumetric texture

By mapping 3D coordinates to colors, we can create *volumetric texture*. The input to the texture is local model coordinates; the output is color and surface characteristics.

For example, to produce wood-grain texture, trees grow rings, with darker wood from earlier in the year and lighter wood from later in the year.

- Choose shades of early and late wood
- $f(P) = (X_P^2 + Z_P^2) \bmod 1$
- $\text{color}(P) = \text{earlyWood} + f(P) * (\text{lateWood} - \text{earlyWood})$



$$f(P)=0$$

$$f(P)=1$$



Adding realism

The teapot on the previous slide doesn't look very wooden, because it's perfectly uniform. One way to make the surface look more natural is to add a randomized noise field to $f(P)$:

$$f(P) = (X_P^2 + Z_P^2 + \text{noise}(P)) \bmod 1$$

where $\text{noise}(P)$ is a function that maps 3D coordinates in space to scalar values chosen at random.

For natural-looking results, use *Perlin noise*, which interpolates smoothly between noise values.



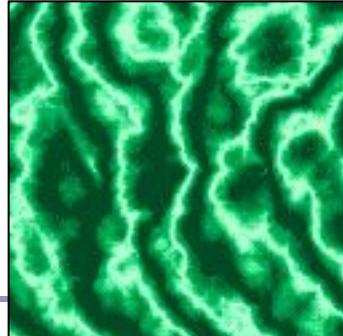
Perlin noise

Perlin noise (invented by Ken Perlin) is a method for generating noise which has some useful traits:

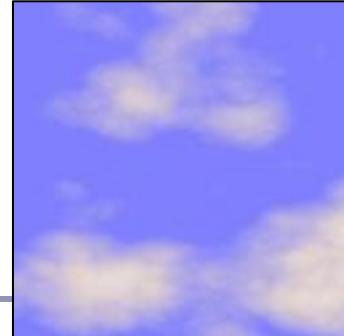
- It is a *band-limited repeatable pseudorandom* function (in the words of its author, Ken Perlin)
- It is bounded within a range close $[-1, 1]$
- It varies continuously, without discontinuity
- It has regions of relative stability
- It can be initialized with random values, extended arbitrarily in space, yet cached deterministically
 - Perlin's talk: <http://www.noisemachine.com/talk1/>



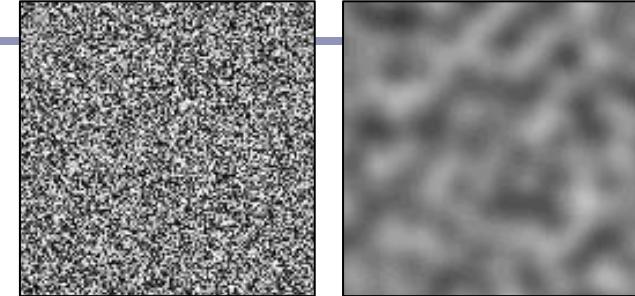
Matt Zucker



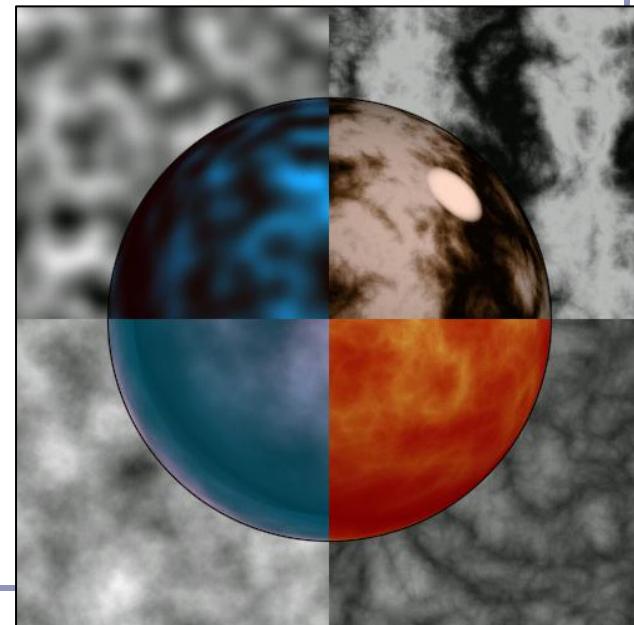
Matt Zucker



Matt Zucker



Non-coherent noise (left) and Perlin noise (right)
Image credit: Matt Zucker



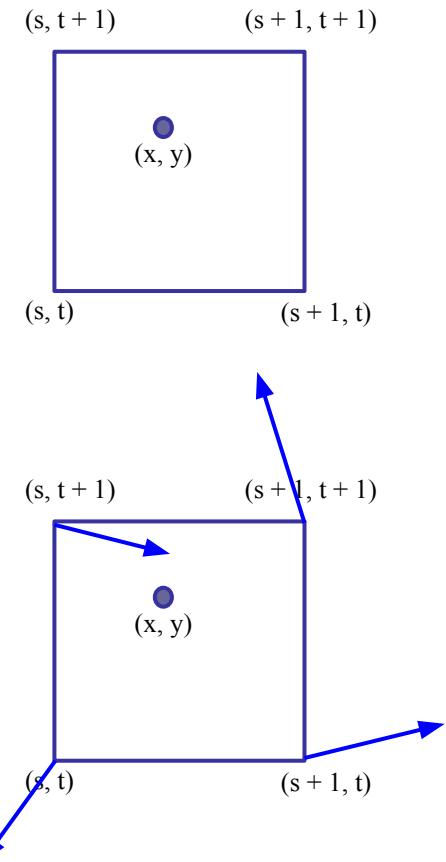
Ken Perlin

Perlin noise 1

Perlin noise caches ‘seed’ random values on a grid at integer intervals. You’ll look up noise values at arbitrary points in the plane, and they’ll be determined by the four nearest seed randoms on the grid.

Given point (x, y) , let $(s, t) = (\text{floor}(x), \text{floor}(y))$.

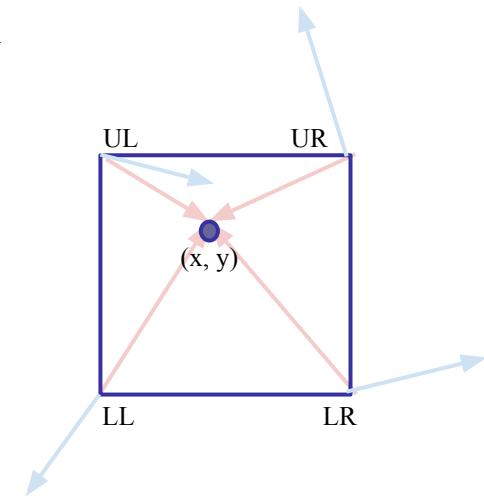
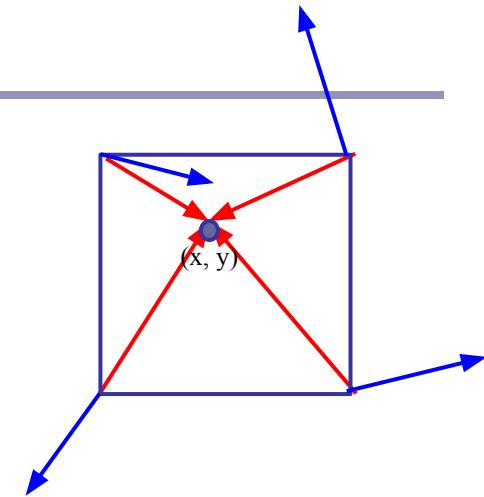
For each grid vertex in
 $\{(s, t), (s+1, t), (s+1, t+1), (s, t+1)\}$
choose and cache a random vector of length one.



Perlin noise 2

For each of the four corners, take the dot product of the random seed vector with the vector from that corner to (x, y) . This gives you a unique scalar value per corner.

- As (x, y) moves across this cell of the grid, the values of the dot products will change smoothly, with no discontinuity.
- As (x, y) approaches a grid point, the contribution from that point will approach zero.
- The values of LL, LR, UL, UR are clamped to a range close to $[-1, 1]$.



Perlin noise 3

Now we take a weighted average of *LL*, *LR*, *UL*, *UR*.

Perlin noise uses a weighted averaging function chosen such that values close to zero and one are moved closer to zero and one, called the *ease curve*:

$$S(t) = 3t^2 - 2t^3$$

We interpolate along one axis first:

$$L(x, y) = LL + S(x - \text{floor}(x))(LR - LL)$$

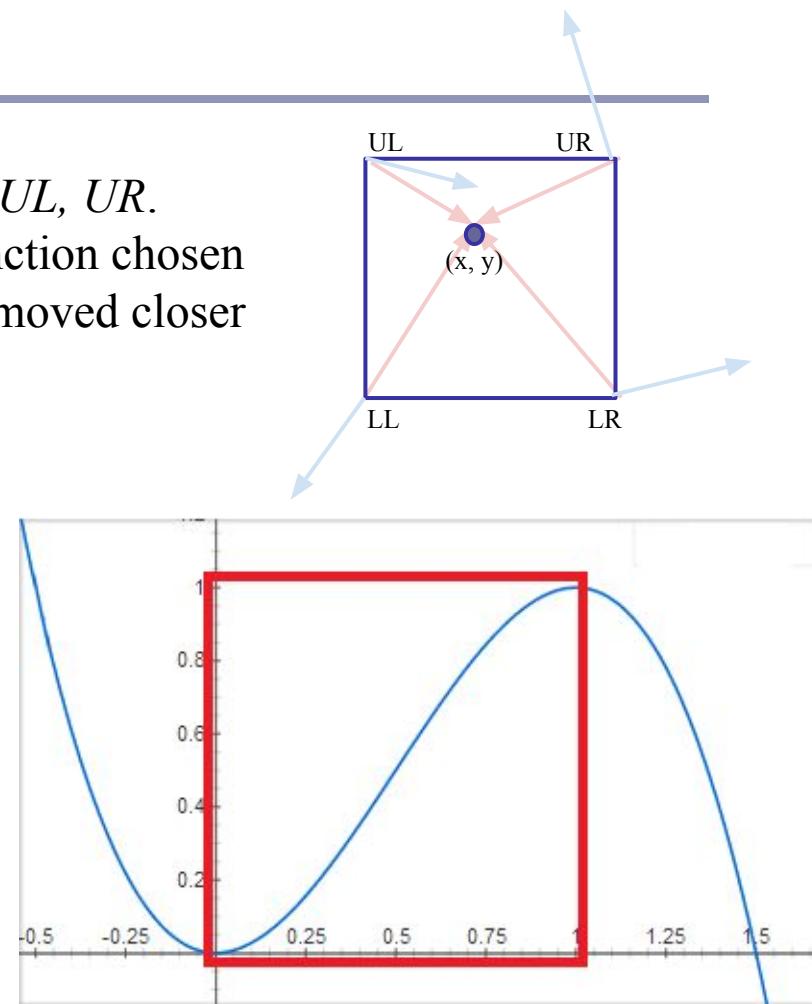
$$U(x, y) = UL + S(x - \text{floor}(x))(UR - UL)$$

Then we interpolate again to merge the two upper and lower functions:

$$\text{noise}(x, y) =$$

$$L(x, y) + S(y - \text{floor}(y))(U(x, y) - L(x, y))$$

Voila!



The 'ease curve'

Tuning noise



Texture frequency
 $1 \rightarrow 3$

Noise frequency
 $1 \rightarrow 3$

Noise amplitude
 $1 \rightarrow 3$

References

Ray tracing

Peter Shirley, Steve Marschner. *Fundamentals of Computer Graphics*. Taylor & Francis, 21 Jul 2009

Hughes, Van Dam et al. *Computer Graphics: Principles and Practice*. Addison Wesley, 3rd edition (10 July 2013)

Anisotropic shading

Greg Ward, “Measuring and Modeling Anisotropic Reflection”, Computer Graphics (SIGGRAPH ’92 Proceedings), pp. 265–272, July 1992

(<http://radsite.lbl.gov/radiance/papers/sg92/paper.html>)

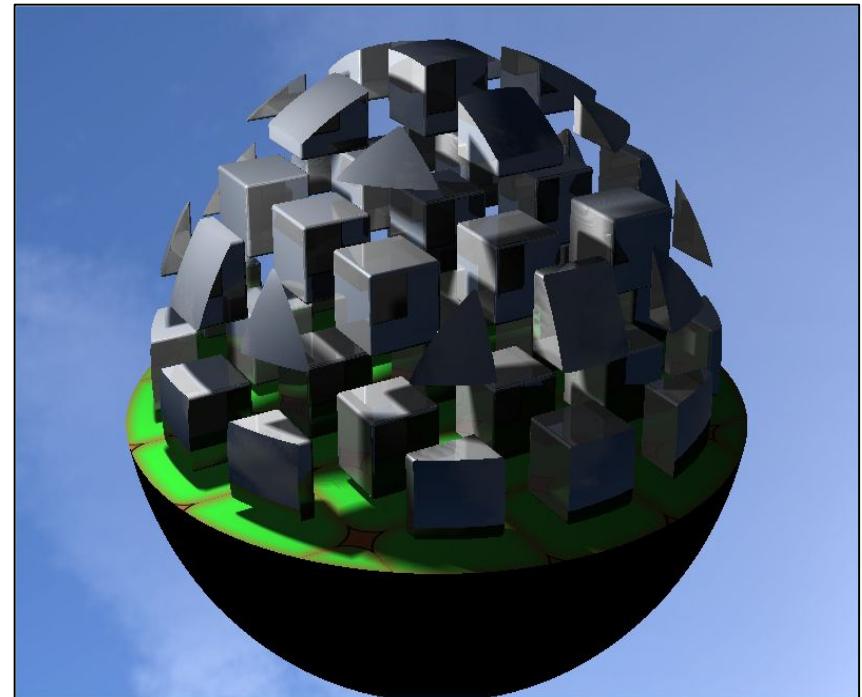
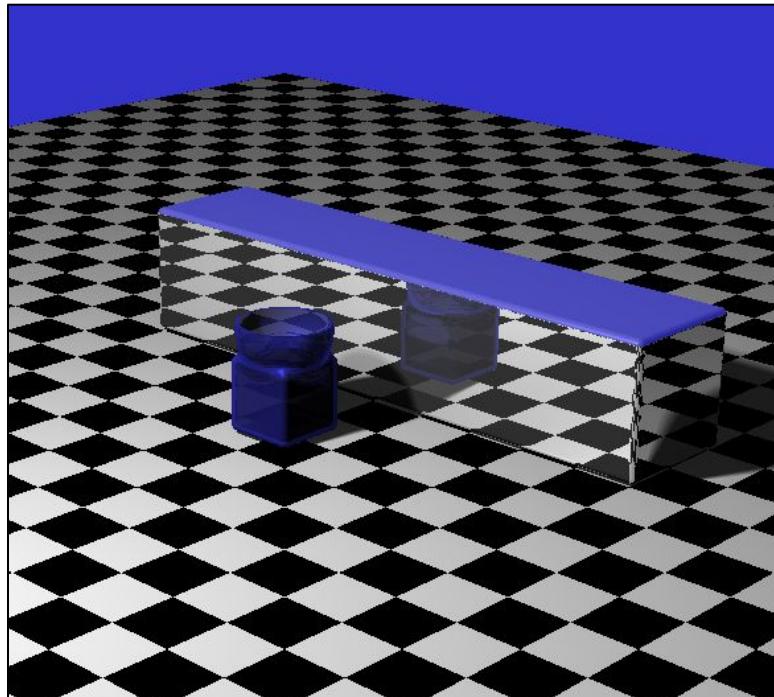
https://en.wikibooks.org/wiki/GLSL_Programming/Unity/Brushed_Metal

Perlin noise

<http://www.noisemachine.com/talk1/>

<http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html>

Advanced Graphics

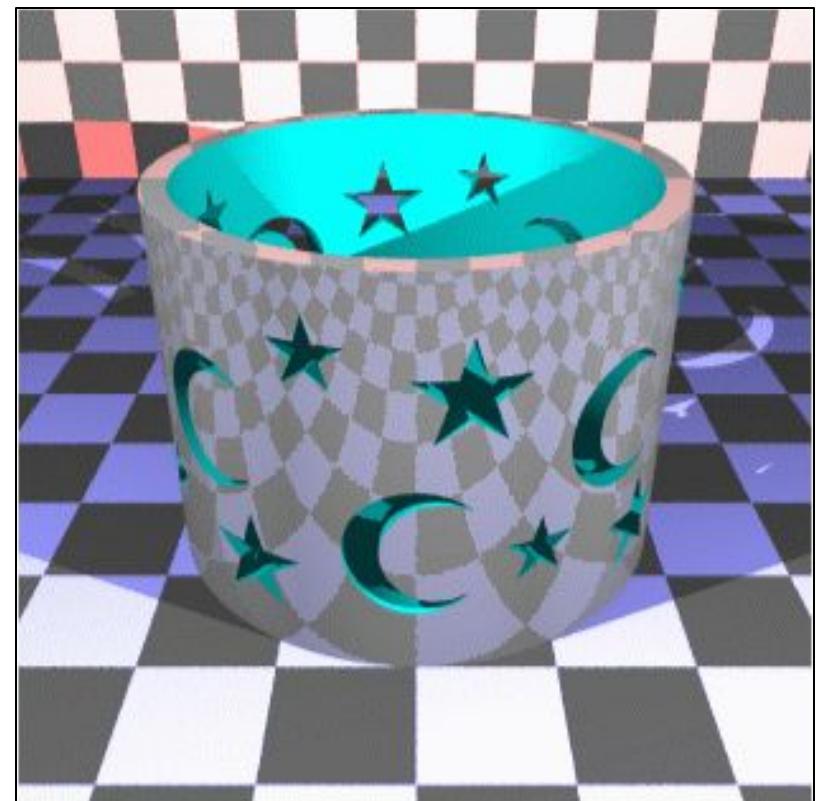


Ray Marching and Advanced Scenes

Constructive Solid Geometry

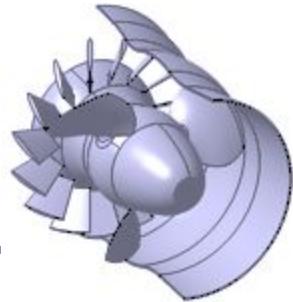
Constructive Solid Geometry (CSG) builds complicated forms out of simple primitives.

These primitives are combined with basic boolean operations: add, subtract, intersect.



CSG figure by Neil Dodgson

Constructive Solid Geometry



CSG models are easy to ray-trace but difficult to polygonalize

- Issues include choosing polygon boundaries at edges; converting adequately from pure smooth primitives to discrete (flat) faces; handling ‘infinitely thin’ sheet surfaces; and others.
- This is an ongoing research topic.

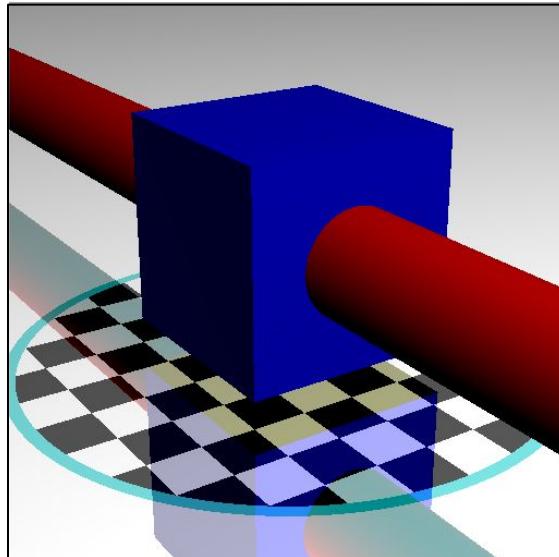
CSG models are well-suited to machine milling, automated manufacture, etc

- Great for 3D printers!

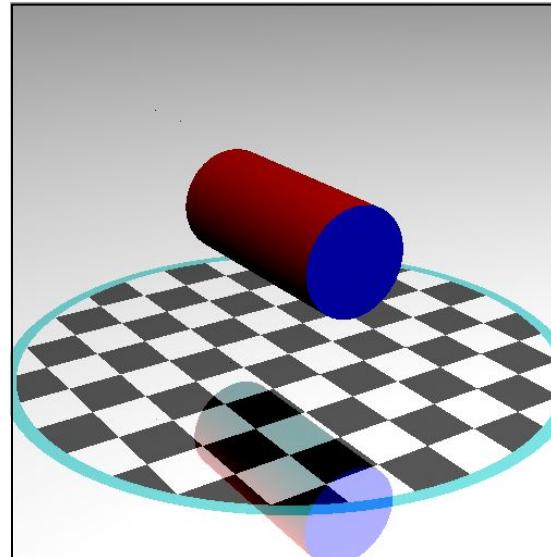
Constructive Solid Geometry

Three operations:

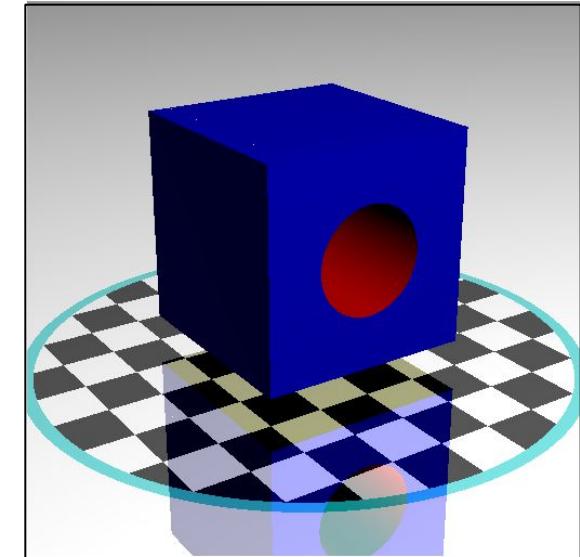
1. *Union*



2. *Intersection*



3. *Difference*



Constructive Solid Geometry

CSG surfaces can be described by a binary tree, where each leaf node is a primitive and each non-leaf node is a boolean operation.

(What would the *not* of a surface look like?)

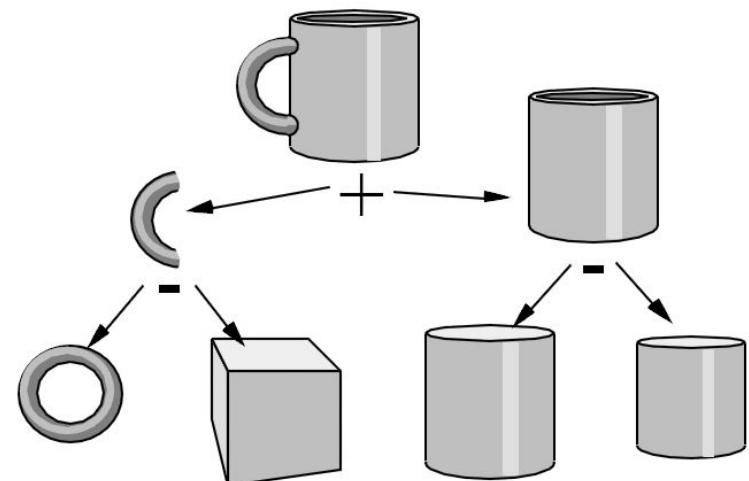
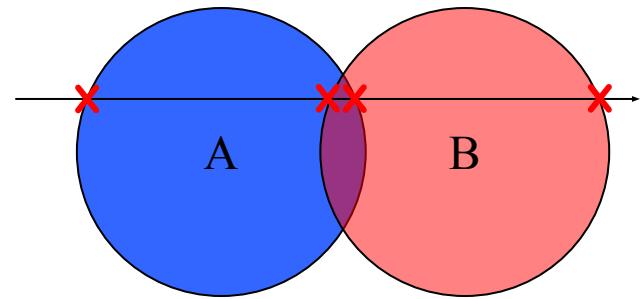


Figure from Wyvill (1995) part two, p. 4

Ray-tracing CSG models

For each node of the binary tree:

- Fire ray r at A and B .
- List in t -order all points where r enters or leaves A or B .
 - You can think of each intersection as a quad of booleans--
 $(wasInA, isInA, wasInB, isInB)$
- Discard from the list all intersections which don't matter to the current boolean operation.
- Pass the list up to the parent node and recurse.

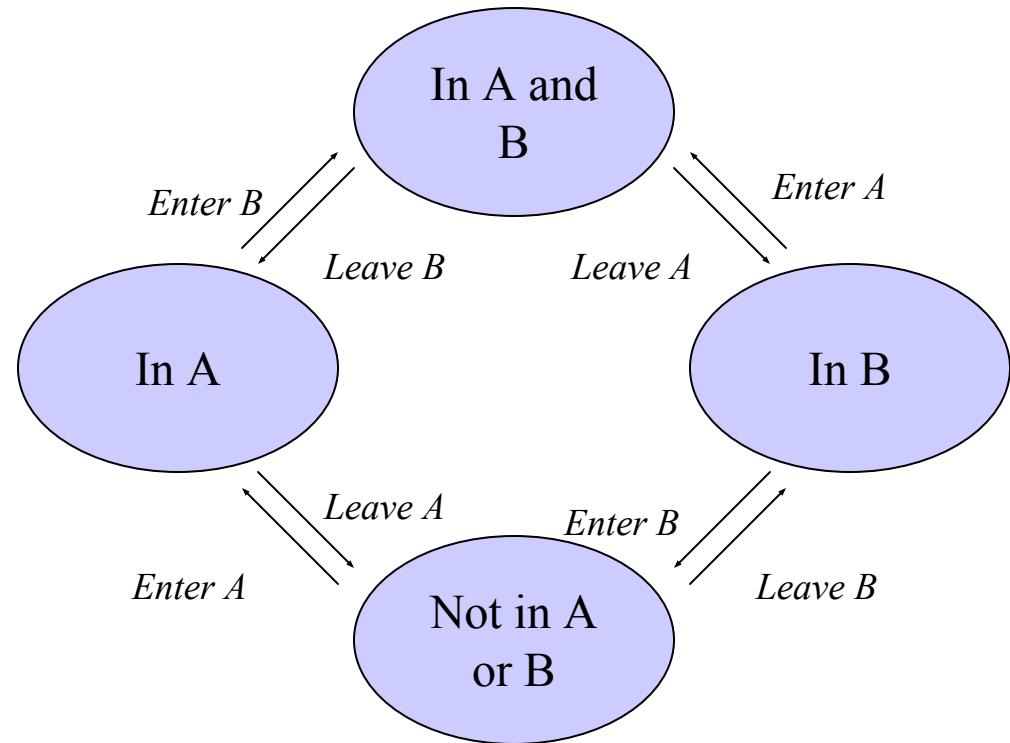


Ray-tracing CSG models

Each boolean operation can be modeled as a state machine.

For each operation, retain those intersections that transition into or out of the critical state(s).

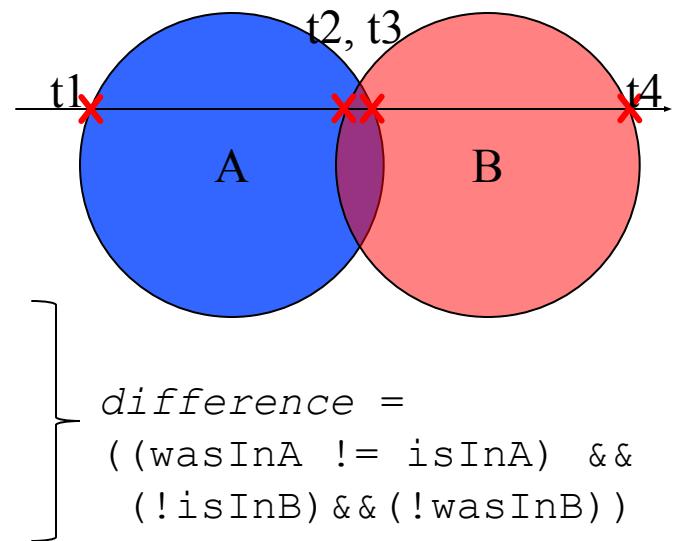
- Union: $\{\text{In A} \mid \text{In B} \mid \text{In A and B}\}$
- Intersection: $\{\text{In A and B}\}$
- Difference: $\{\text{In A}\}$



Ray-tracing CSG models

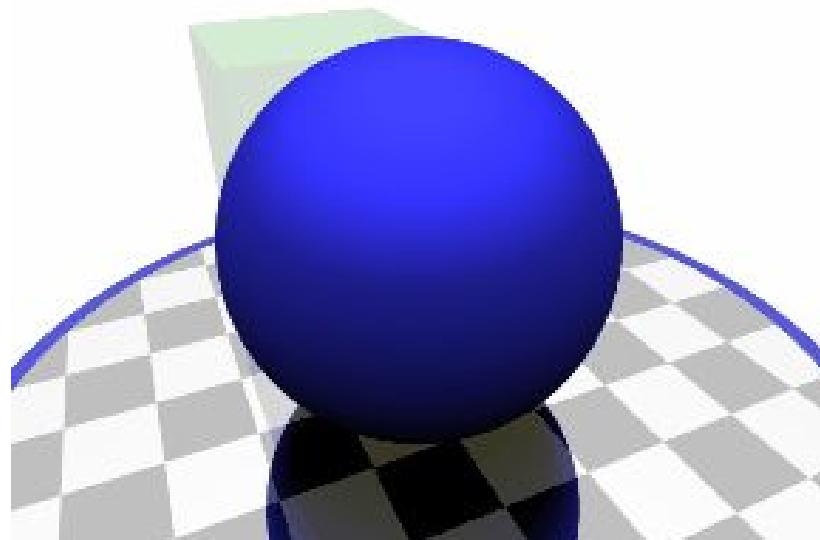
Example: Difference (A-B)

A-B	Was In A	Is In A	Was In B	Is In B
t1	No	Yes	No	No
t2	Yes	Yes	No	Yes
t3	Yes	No	Yes	Yes
t4	No	No	Yes	No

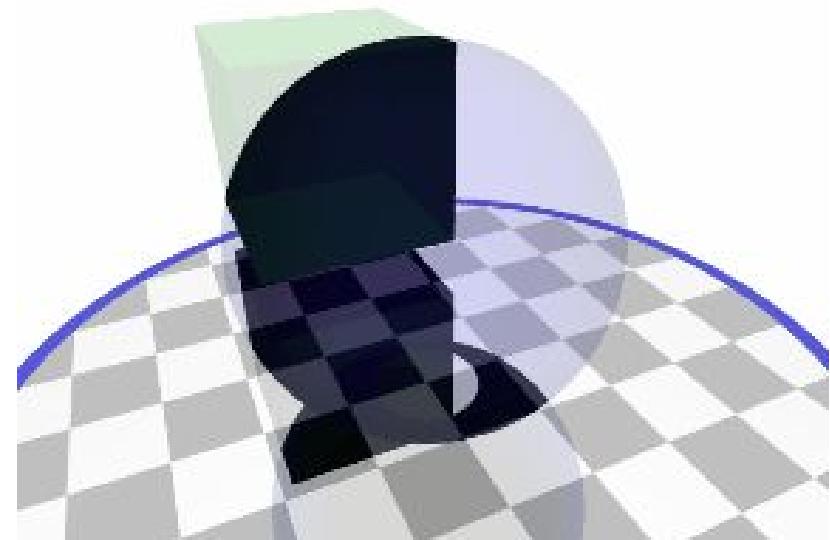


```
difference =  
((wasInA != isInA) &&  
(!isInB) && (!wasInB))  
||  
((wasInB != isInB) &&  
(wasInA || isInA))
```

CSG in action



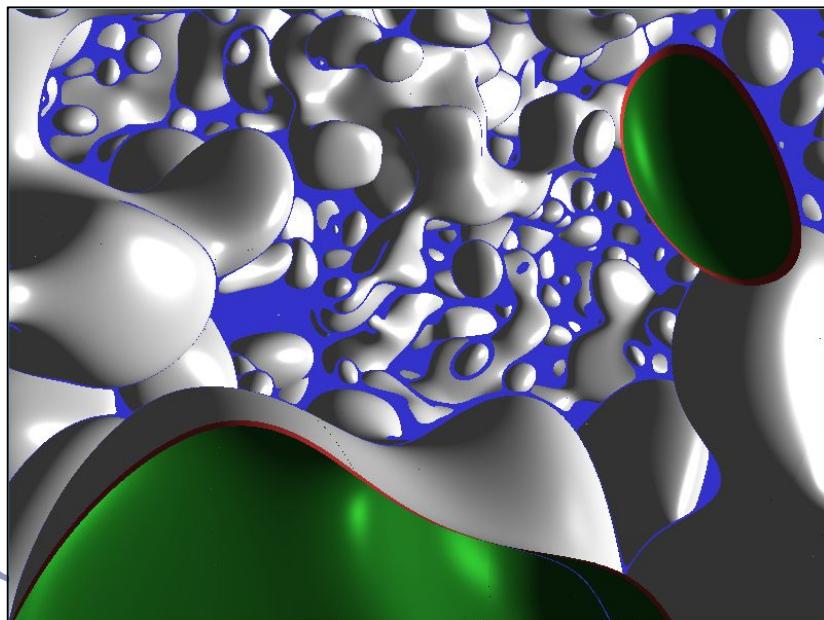
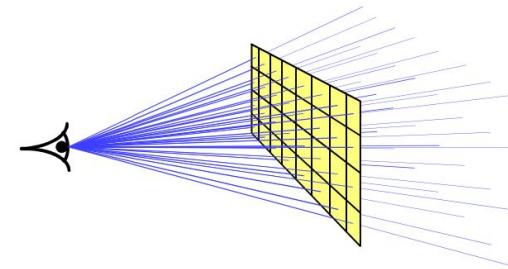
Difference



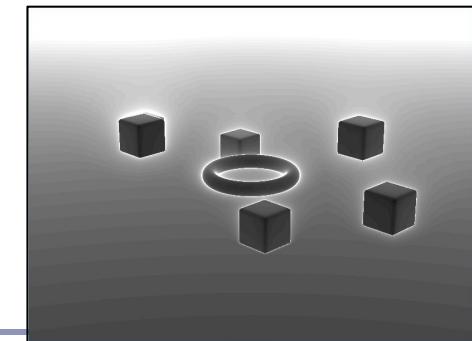
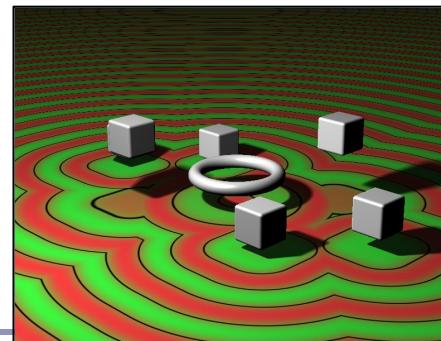
Intersection

GPU Ray-tracing

Ray tracing 101: “Choose the color of the pixel by firing a ray through and seeing what it hits.”

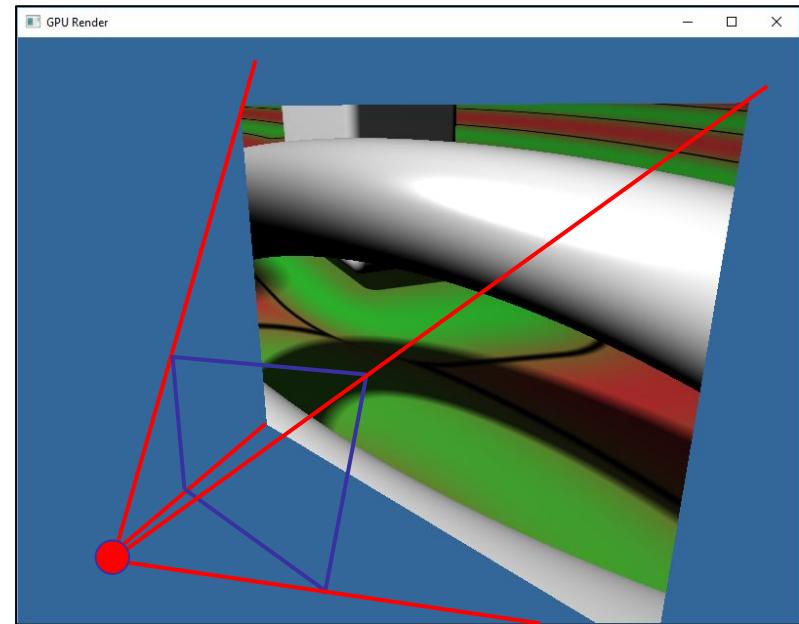


Ray tracing 102:
“Let the pixel make up its own mind.”



GPU Ray-tracing

1. Use a minimal fragment shader (no transforms)
2. Set up OpenGL with minimal geometry, a single quad
3. Bind a `vec2` to each vertex specifying ‘texture’ coordinates
4. Implement raytracing in GLSL per pixel:
 - a. For each pixel, compute the ray from the eye through the pixel, using the interpolated texture coordinate to identify the pixel
 - b. Run the ray tracing algorithm for every ray



GPU Ray-tracing

```
// Window dimensions
uniform vec2 iResolution;
// Camera position
uniform vec3 iRayOrigin;
// Camera facing direction
uniform vec3 iRayDir;
// Camera up direction
uniform vec3 iRayUp;
// Distance to viewing plane
uniform float iPlaneDist;

// 'Texture' coordinate of each
// vertex, interpolated across
// fragments
in vec2 texCoord;
```

```
vec3 getRayDir(
    vec3 camDir,
    vec3 camUp,
    vec2 texCoord) {
    vec3 xAxis = normalize(
        cross(camDir, camUp));
    vec2 p = 2.0 * texCoord - 1.0;
    p.x *= iResolution.x
        / iResolution.y;
    return normalize(
        p.x * xAxis
        + p.y * camUp
        + iPlaneDist * camDir);
}
```

GPU Ray-tracing

```
Hit traceSphere(vec3 rayorig, vec3 raydir, vec3 pos, float radius) {  
    float OdotD = dot(rayorig - pos, raydir);  
    float OdotO = dot(rayorig - pos, rayorig - pos);  
    float base = OdotD * OdotD - OdotO + radius * radius;  
  
    if (base >= 0) {  
        float root = sqrt(base);  
        float t1 = -OdotD + root;  
        float t2 = -OdotD - root;  
        if (t1 >= 0 || t2 >= 0) {  
            float t = (t1 < t2 && t1 >= 0) ? t1 : t2;  
            vec3 pt = rayorig + raydir * t;  
            vec3 normal = normalize(pt - pos);  
            return Hit(pt, normal, t);  
        }  
    }  
    return Hit(vec3(0), vec3(0), -1);  
}
```

GPU Ray-tracing

One key limitation of some GLSL platforms (specifically GLSL ES, for mobile devices and WebGL) is that **GLSL may not support recursion**. That makes recursing to find reflected / refracted /transparency colors difficult.

We can work around this by treating the illumination equation as a weighted polynomial, where the weight of each blended contribution is computed before the contribution itself.

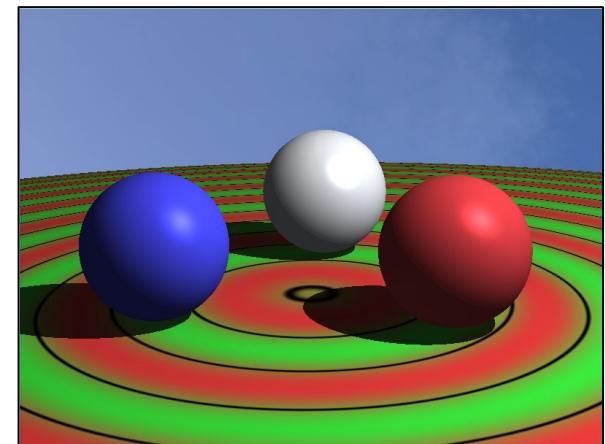
```
struct TBD {  
    vec3 src;  
    vec3 dir;  
    float weight;  
};
```

```
vec3 renderScene(vec3 rayorig, vec3 raydir) {  
    TBD tbd[10];  
    int numTbd = 0;  
    vec3 cumulativeColor = vec3(0);  
  
    tbd[numTbd++] = TBD(rayorig, raydir, 1.0);  
    for (int i = 0; i < 10 && numTbd > 0; i++) {  
        color = // fire ray, compute local color  
        cumulativeColor += tbd[i].weight * color;  
        tbd[numTbd++] = // reflection ray  
        tbd[numTbd++] = // refraction ray  
        ...  
    }
```

Textured skies

```
#define PI 3.14159
uniform sampler2D texture;

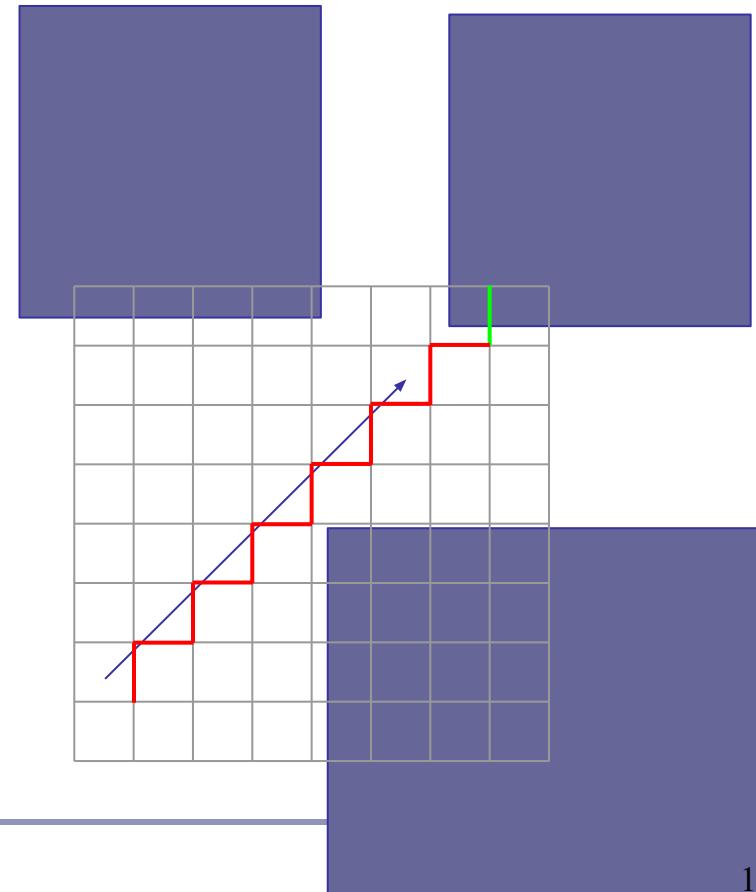
vec3 getBackground(vec3 dir) {
    float u = 0.5 + atan(dir.z, -dir.x) / (2 * PI);
    float v = 0.5 - asin(dir.y) / PI;
    vec4 texColor = texture2D(texture, vec2(u, v));
    return texColor.rgb;
}
```



An alternative to raytracing: *Ray-marching*

An alternative to classic ray-tracing is ray-marching, in which we take a series of finite steps along the ray until we strike an object or exceed the number of permitted steps.

- Also sometimes called ray casting
- Scene objects only need to answer,
“has this ray hit you? y/n”
- Great solution for data like height fields
- Unfortunately...
 - often involves many steps
 - too large a step size can lead to lost intersections (step over the object)
 - an `if()` test in the heart of a `for()` loop is very hard for the GPU to optimize



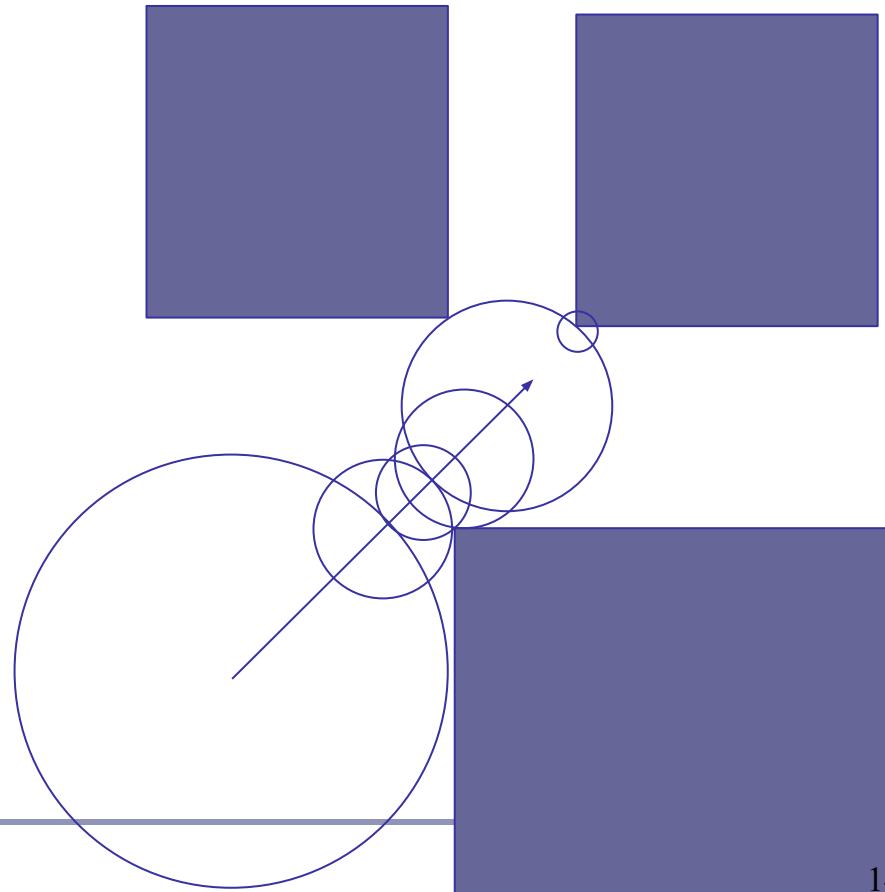
GPU Ray-marching: Signed Distance Fields

Ray-marching can be dramatically improved, to impressive realtime GPU performance, using *signed distance fields*:

1. Fire ray into scene
2. At each step, measure distance field function: $d(p) = [\text{distance to nearest object in scene}]$
3. Advance ray along ray heading by distance d , because the nearest intersection can be no closer than d

This is also sometimes called ‘sphere tracing’. Early paper:

<http://graphics.cs.illinois.edu/sites/default/files/rtqjs.pdf>



Signed distance functions

The theory is simple: the SDF computes the minimum possible distance to the surface.

The sphere, for instance, is the distance from p to the center of the sphere, less the radius.

Negative values indicate a sample inside the surface, and still express absolute distance to the surface.

```
float sphere(vec3 p, float r) {
    return length(p) - r;
}

float cube(vec3 p, vec3 c) {
    vec3 d = abs(p) - c;
    return min(max(d.x,
                    max(d.y, d.z)), 0.0)
        + length(max(d, 0.0));
}

float cylinder(vec3 p, vec3 c) {
    return
        length(p.xz - c.xy) - c.z;
}

float torus(vec3 p, vec2 t) {
    vec2 q = vec2(
        length(p.xz) - t.x, p.y);
    return length(q) - t.y;
}
```

Raymarching signed distance fields

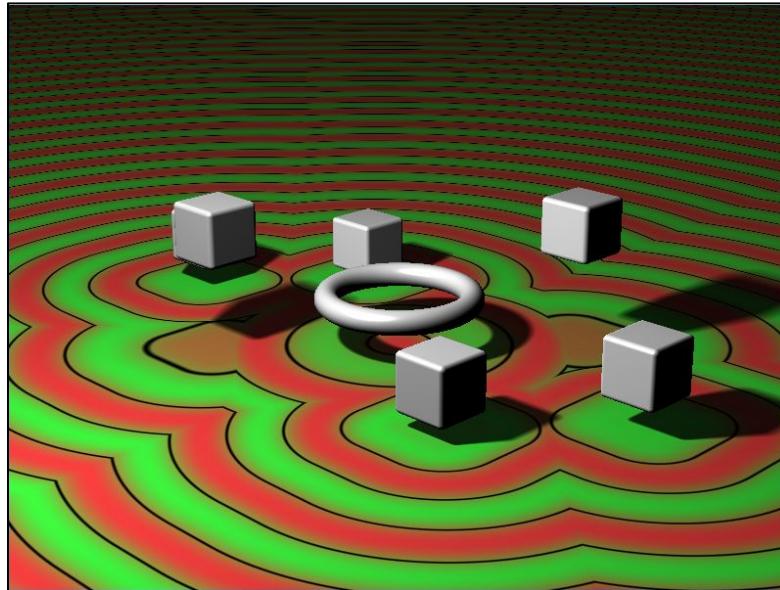
```
vec3 raymarch(vec3 pos, vec3 raydir) {
    int step = 0;
    float d = getSdf(pos);

    while (abs(d) > 0.001 && step < 50) {
        pos = pos + raydir * d;
        d = getSdf(pos); // Return sphere(pos) or any other
        step++;
    }

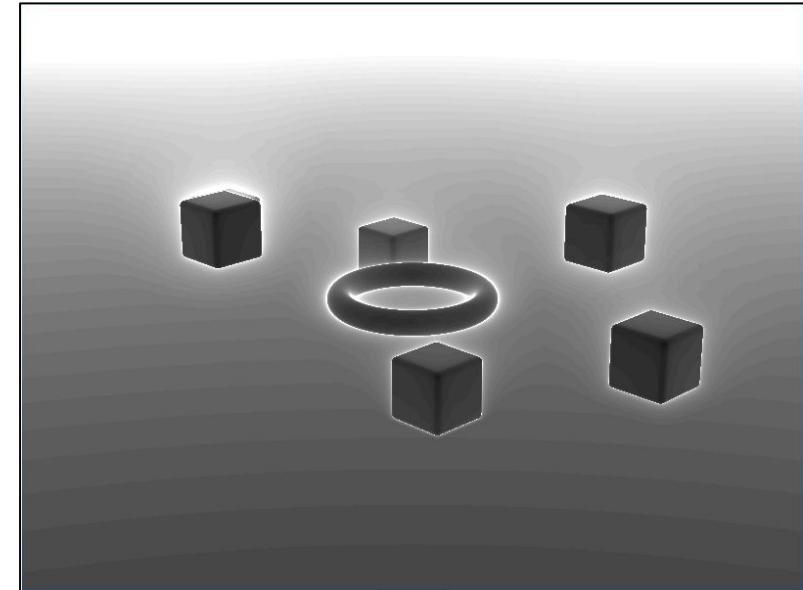
    return
        (step < 50) ? illuminate(pos, rayorig) : background;
}
```

Visualizing step count

Final image



Distance field



Brighter = more steps, up to 50

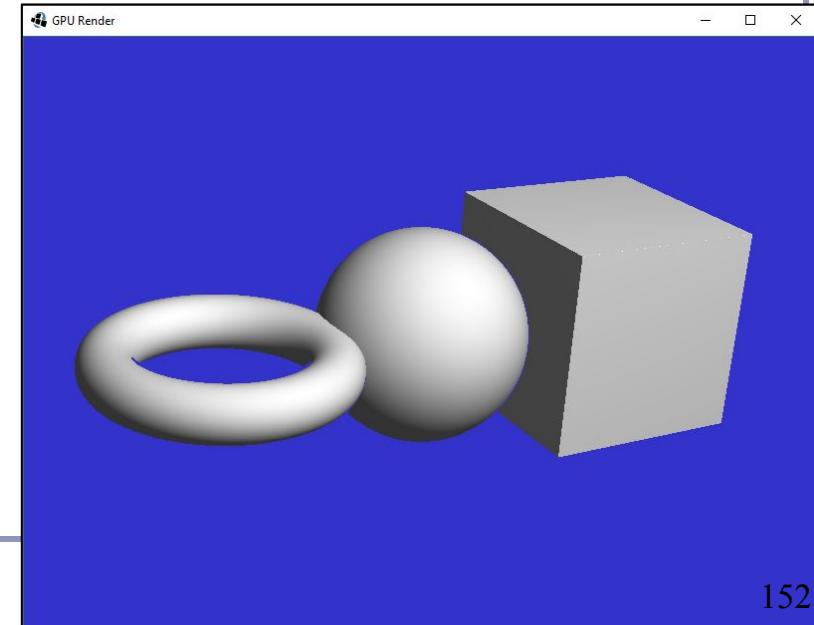
Find the normal to an SDF

Finding the normal: local gradient

```
float d = getSdf(pt);  
vec3 normal = normalize(vec3(  
    getSdf(vec3(pt.x + 0.0001, pt.y, pt.z)) - d,  
    getSdf(vec3(pt.x, pt.y + 0.0001, pt.z)) - d,  
    getSdf(vec3(pt.x, pt.y, pt.z + 0.0001)) - d));
```

The distance function is locally linear and changes most as the sample moves directly away from the surface. At the surface, the direction of greatest change is therefore equivalent to the normal to the surface.

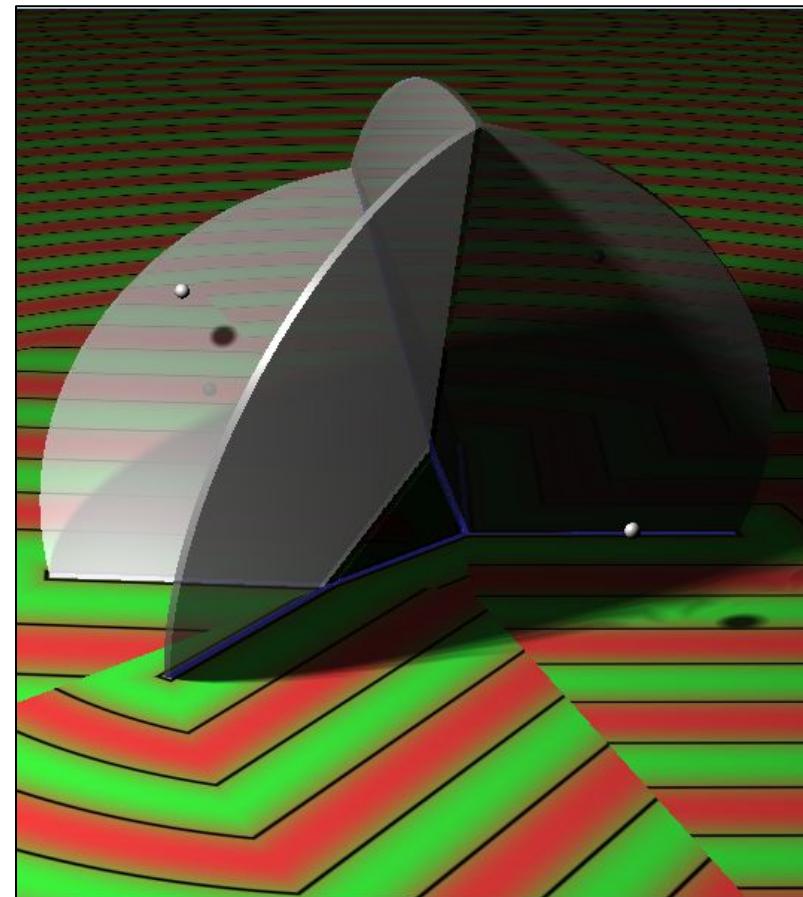
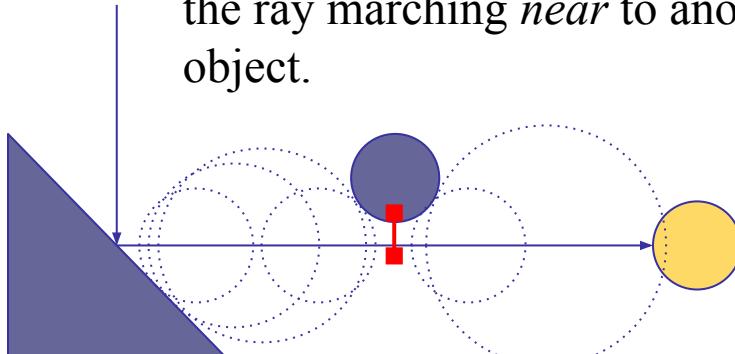
Thus the local gradient (the normal) can be approximated from the distance function.



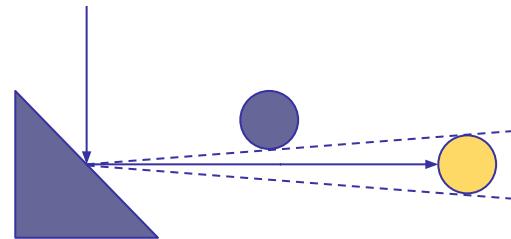
SDF shadows

Ray-marched shadows are straightforward: march a ray towards each light source, illuminate if the SDF ever drops too close to zero.

Unlike ray-tracing, soft shadows are almost free with SDFs: attenuate illumination by a linear function of the ray marching *near* to another object.

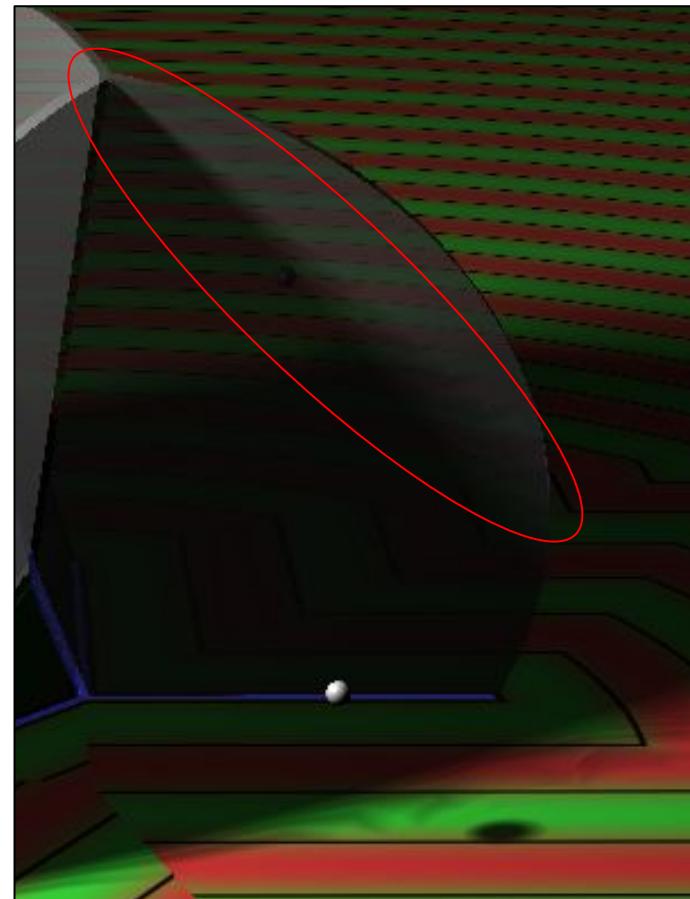


Soft SDF shadows



```
float shadow(vec3 pt) {  
    vec3 lightDir = normalize(lightPos - pt);  
    float kd = 1;  
    int step = 0;  
  
    for (float t = 0.1;  
        t < length(lightPos - pt)  
        && step < renderDepth && kd > 0.001; ) {  
        float d = abs(getSDF(pt + t * lightDir));  
        if (d < 0.001) {  
            kd = 0;  
        } else {  
            kd = min(kd, 16 * d / t);  
        }  
        t += d;  
        step++;  
    }  
    return kd;  
}
```

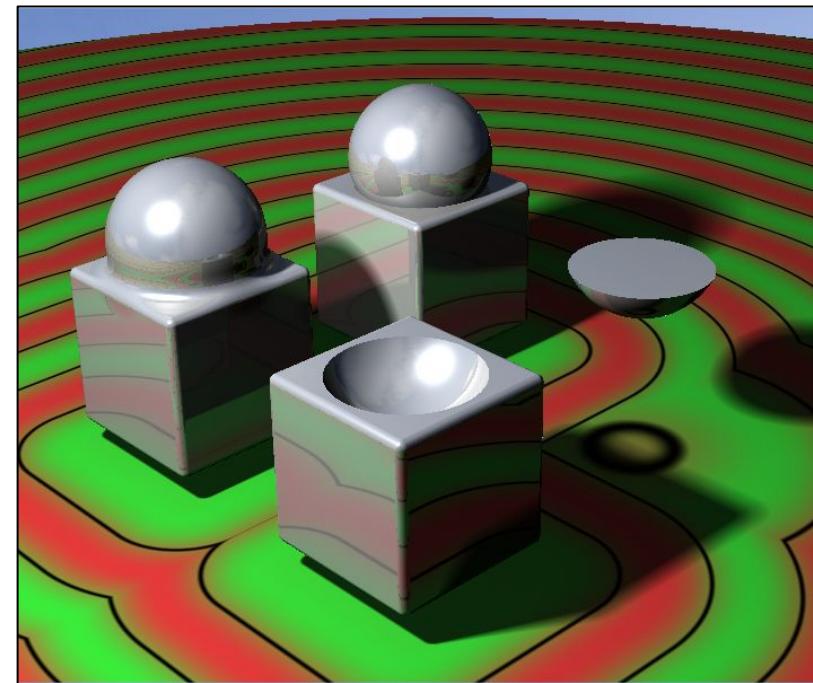
By dividing d by t , we attenuate the strength of the shadow as its source is further from the illuminated point.



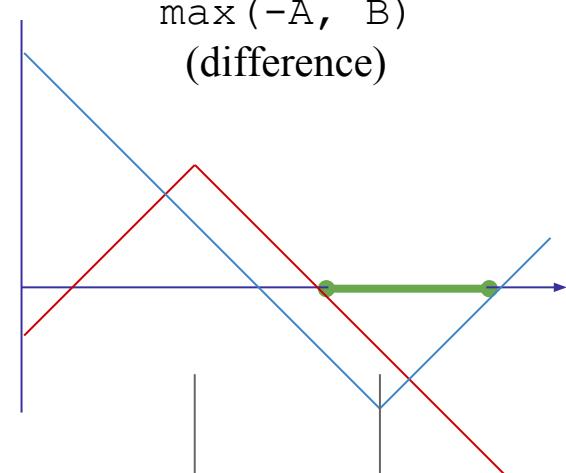
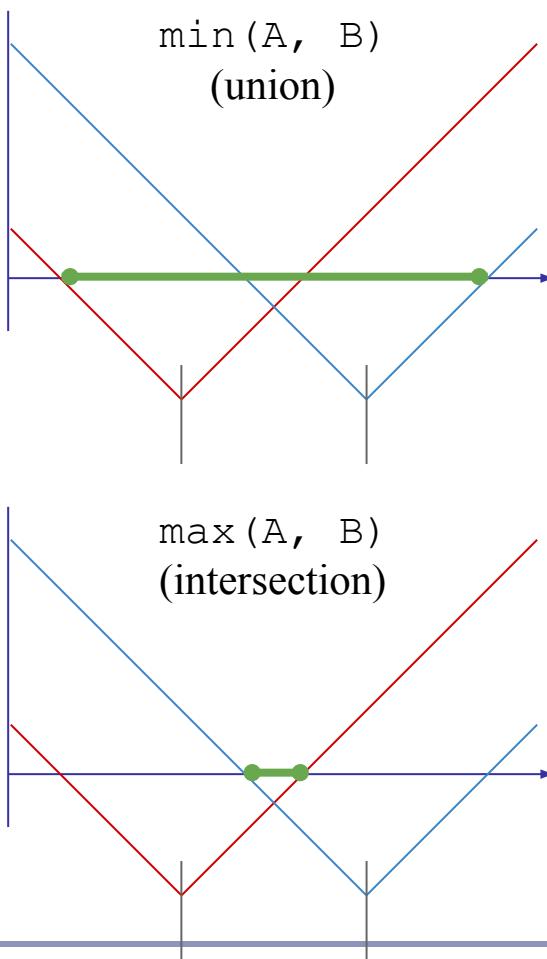
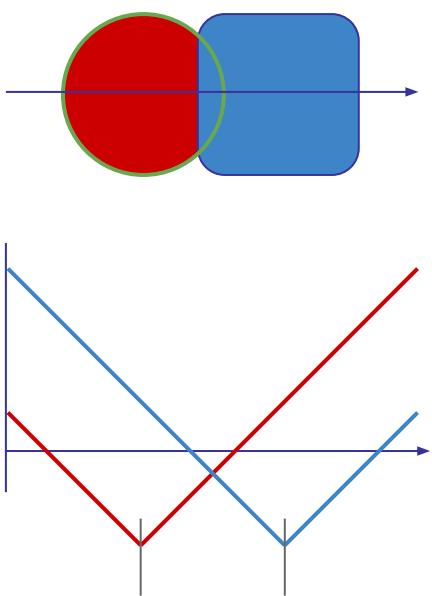
Combining SDFs

We combine SDF models by choosing which is closer to the sampled point.

- Take the **union** of two SDFs by taking the `min()` of their functions.
- Take the **intersection** of two SDFs by taking the `max()` of their functions.
- The `max()` of function A and the negative of function B will return the **difference** of A - B.

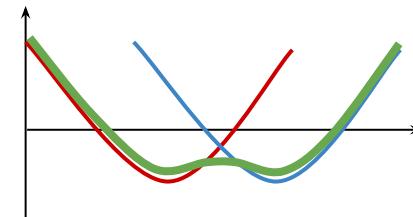
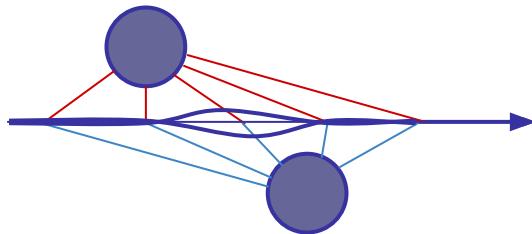


Combining SDFs



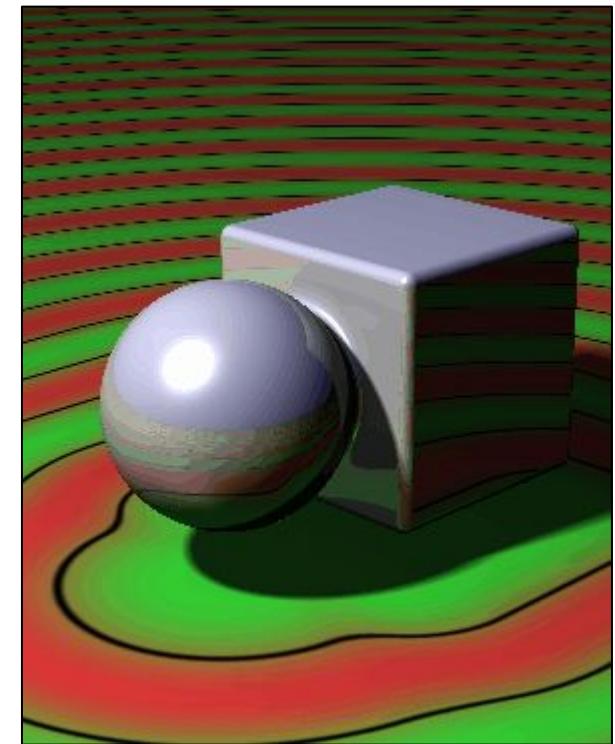
Blending SDFs

Taking the `min()`, `max()`, etc of two SDFs yields a sharp discontinuity. *Interpolating* the two SDFs with a smooth polynomial yields a smooth distance curve, blending the models:



Sample blending function (Quilez)

```
float blend(float a, float b, float k) {  
    a = pow(a, k);  
    b = pow(b, k);  
    return pow((a * b) / (a + b), 1.0 / k);  
}
```



Transforming SDF geometry

To rotate, translate or scale an SDF model, apply the inverse transform to the input point within your distance function.

Ex:

```
float sphere(vec3 pt, float radius) {
    return length(pt) - radius;
}

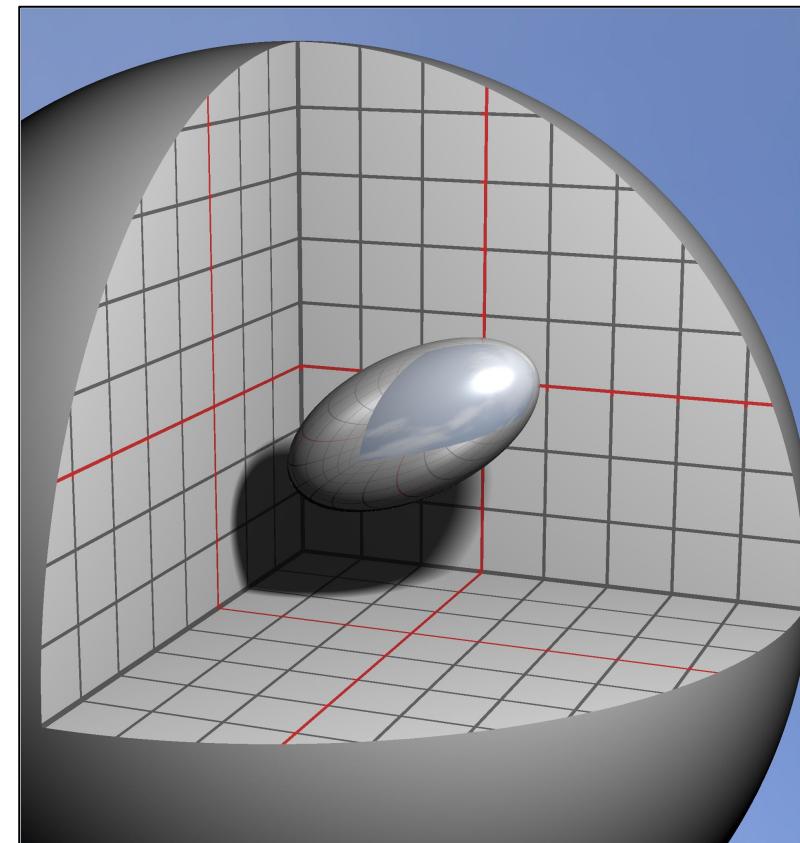
float f(vec3 pt) {
    return sphere(pt - vec3(0, 3, 0));
}
```

This renders a sphere centered at $(0, 3, 0)$.

More prosaically, assemble your local-to-world transform as usual, but apply its inverse to the pt within your distance function.

Transforming SDF geometry

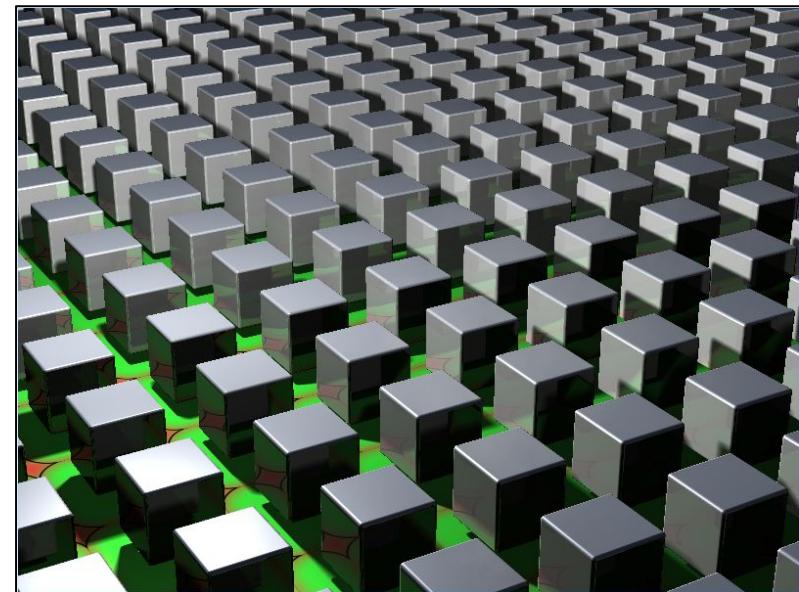
```
float fScene(vec3 pt) {  
  
    // Scale 2x along X  
    mat4 S = mat4(  
        vec4(2, 0, 0, 0),  
        vec4(0, 1, 0, 0),  
        vec4(0, 0, 1, 0),  
        vec4(0, 0, 0, 1));  
  
    // Rotation in XY  
    float t = sin(time) * PI / 4;  
    mat4 R = mat4(  
        vec4(cos(t), sin(t), 0, 0),  
        vec4(-sin(t), cos(t), 0, 0),  
        vec4(0, 0, 1, 0),  
        vec4(0, 0, 0, 1));  
  
    // Translate to (3, 3, 3)  
    mat4 T = mat4(  
        vec4(1, 0, 0, 3),  
        vec4(0, 1, 0, 3),  
        vec4(0, 0, 1, 3),  
        vec4(0, 0, 0, 1));  
  
    pt = (vec4(pt, 1) * inverse(S * R * T)).xyz;  
  
    return sdSphere(pt, 1);  
}
```



Repeating SDF geometry

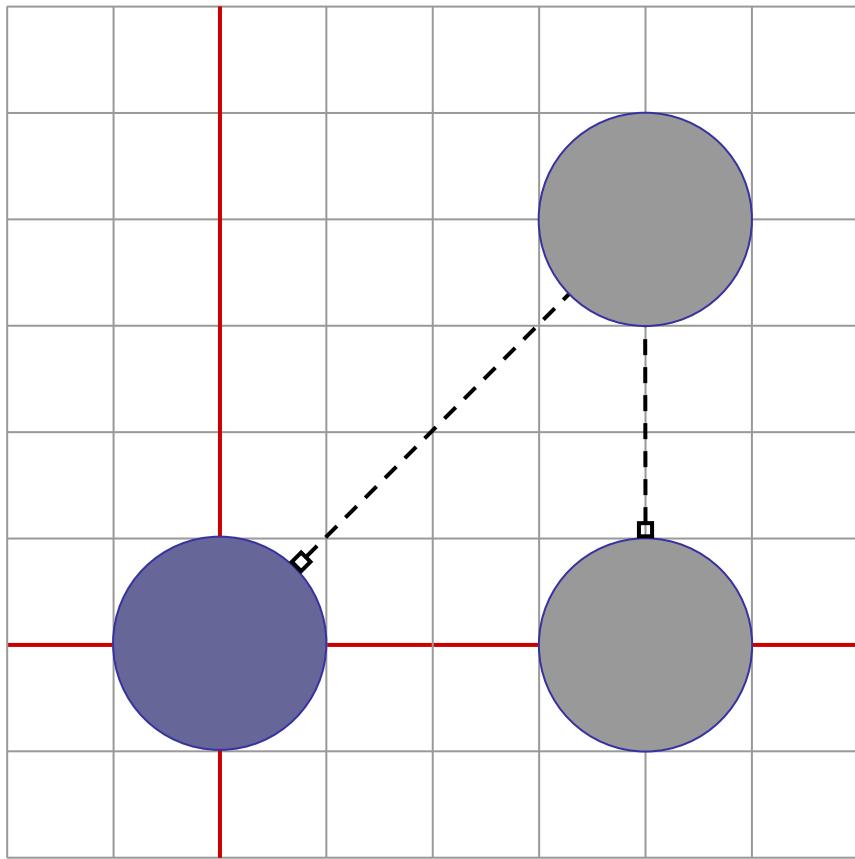
If we take the modulus of a point's position along one or more axes before computing its signed distance, then we segment space into infinite parallel regions of repeated distance. Space near the origin 'repeats'.

With SDFs we get infinite repetition of geometry for no extra cost.



```
float fScene(vec3 pt) {  
    vec3 pos;  
    pos = vec3(mod(pt.x + 2, 4) - 2, pt.y, mod(pt.z + 2, 4) - 2);  
    return sdCube(pos, vec3(1));  
}
```

Repeating SDF geometry



```
float sphere(vec3 pt, float radius) {  
    return length(pt) - radius;  
}
```

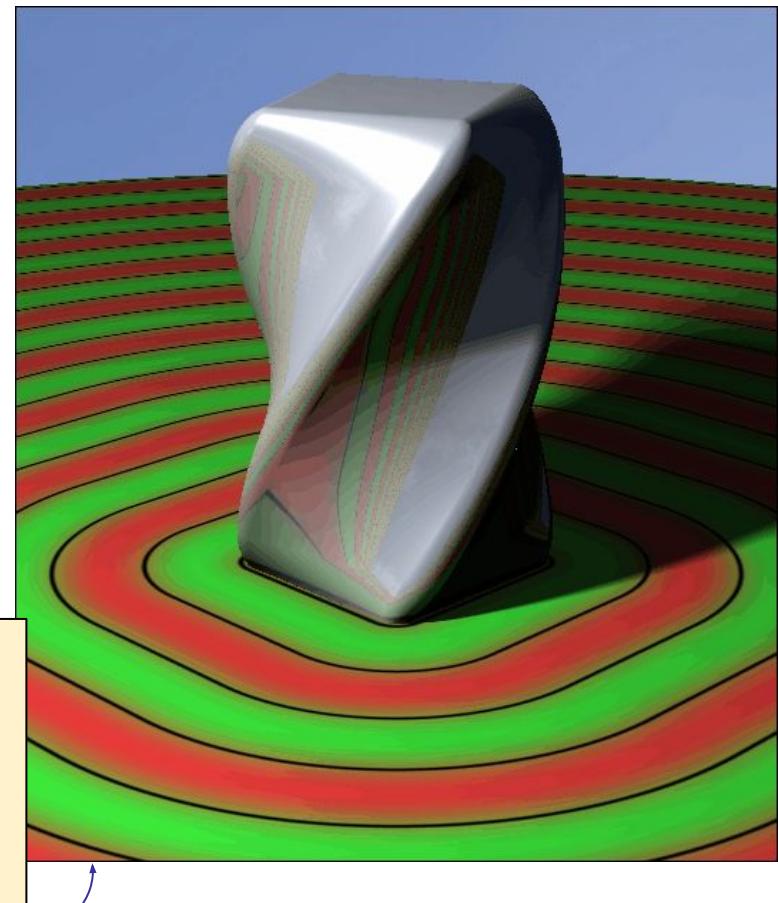
- $\text{sdSphere}(4, 4)$
 $= \sqrt{4*4+4*4} - 1$
 $= \sim 4.5$
- $\text{sdSphere}($
 $((4 + 2) \% 4) - 2, 4)$
 $= \sqrt{0*0+4*4} - 1$
 $= 3$
- $\text{sdSphere}($
 $((4 + 2) \% 4) - 2,$
 $((4 + 2) \% 4) - 2)$
 $= \sqrt{0*0+0*0} - 1$
 $= -1 // \text{ Inside surface}$

Transforming SDF geometry

The previous example modified ‘all of space’ with the same transform, so its distance functions retain their local linearity.

We can also apply non-uniform spatial distortion, such as by choosing how much we’ll modify space as a function of where in space we are.

```
float fScene(vec3 pt) {  
    pt.y -= 1;  
    float t = (pt.y + 2.5) * sin(time);  
    return sdCube(vec3(  
        pt.x * cos(t) - pt.z * sin(t),  
        pt.y / 2,  
        pt.x * sin(t) + pt.z * cos(t)), vec3(1));  
}
```



Recommended reading

Seminal papers:

- John C. Hart et al., “Ray Tracing Deterministic 3-D Fractals”,
<http://graphics.cs.illinois.edu/sites/default/files/rtqjs.pdf>
- John C. Hart, “Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces”, <http://graphics.cs.illinois.edu/papers/zeno>

Special kudos to Inigo Quilez and his amazing blog:

- <http://iquilezles.org/www/articles/smin/smin.htm>
- <http://iquilezles.org/www/articles/distfunctions/distfunctions.htm>

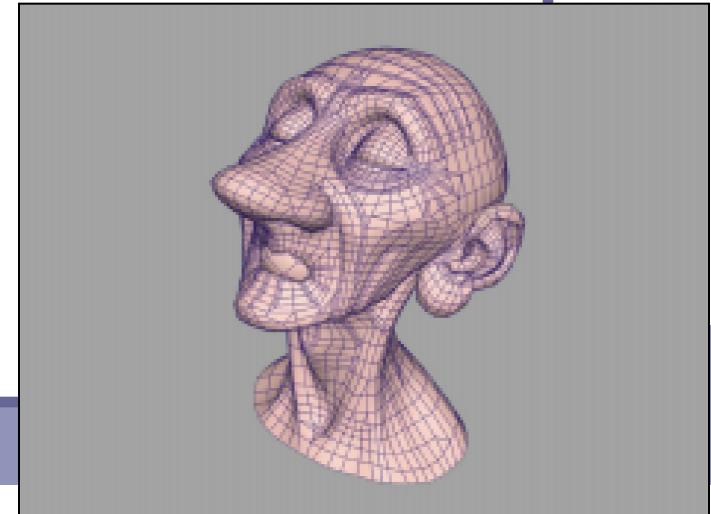
Other useful sources:

- Johann Korndorfer, “How to Create Content with Signed Distance Functions”,
<https://www.youtube.com/watch?v=s8nFqwOho-s>
- Daniel Wright, “Dynamic Occlusion with Signed Distance Fields”,
<http://advances.realtimerendering.com/s2015/DynamicOcclusionWithSignedDistanceFields.pdf>
- 9bit Science, “Raymarching Distance Fields”,
http://9bitscience.blogspot.co.uk/2013/07/raymarching-distance-fields_14.html



Advanced Graphics

*Subdivision
Surfaces*



CAD, CAM, and a new motivation: *shiny things*

Expensive products are sleek and smooth.

→ Expensive products are C2 continuous.



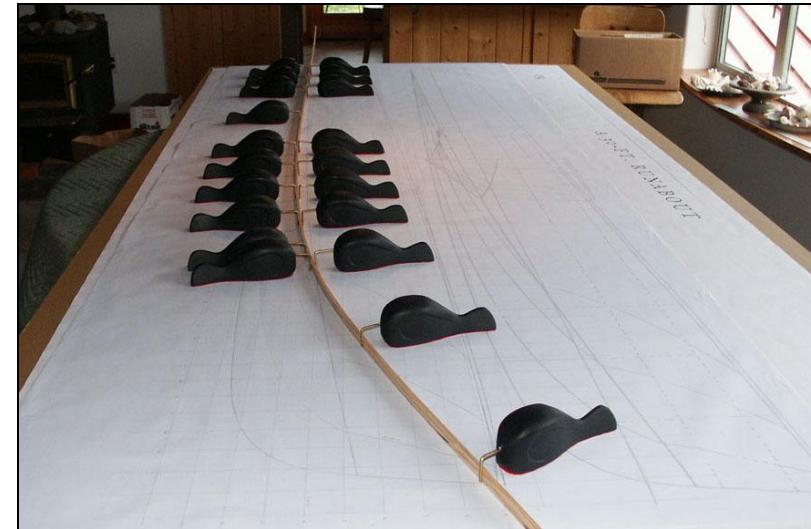
Shiny, but reflections are warped

Shiny, and reflections are perfect

History

The term *spline* comes from the shipbuilding industry: long, thin strips of wood or metal would be bent and held in place by heavy ‘ducks’, lead weights which acted as control points of the curve.

Wooden splines can be described by C_n -continuous Hermite polynomials which interpolate $n+1$ control points.



Top: Fig 3, P.7, Bray and Spectre, *Planking and Fastening*, Wooden Boat Pub (1996)

Bottom: http://www.pranos.com/boatsofwood/lofting%20ducks/lofting_ducks.htm

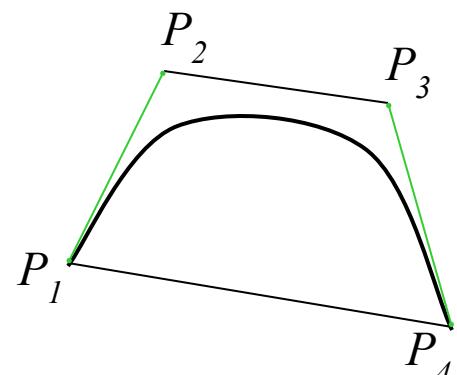
The drive for smooth CAD/CAM

- *Continuity* (smooth curves) can be essential to the perception of *quality*.
- The automotive industry wanted to design cars which were aerodynamic, but also visibly of high quality.
- Bezier (Renault) and de Casteljau (Citroen) invented Bezier curves in the 1960s. de Boor (GM) generalized them to B-splines.



Beziers—a quick review

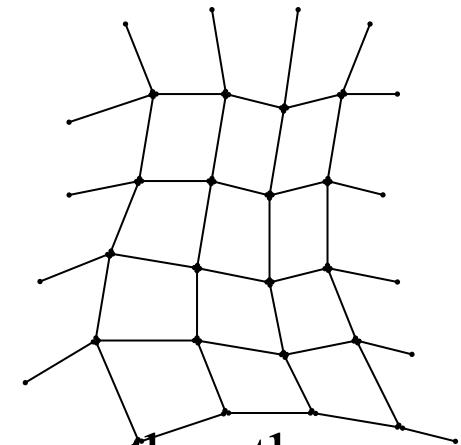
- A Bezier cubic is a function $P(t)$ defined by four control points:
 - P_1 and P_4 are the endpoints of the curve
 - P_2 and P_3 define the other two corners of the bounding polygon.
- The curve fits entirely within the convex hull of $P_1 \dots P_4$.
- Beziers are a subset of a broader class of splines and surfaces called *NURBS*: *Non Uniform Rational B-Splines*.
- For decades, NURBS patches have been the bedrock of CAD/CAM.



$$\text{Cubic: } P(t) = (1-t)^3 P_1 + 3t(1-t)^2 P_2 + 3t^2(1-t)P_3 + t^3 P_4$$

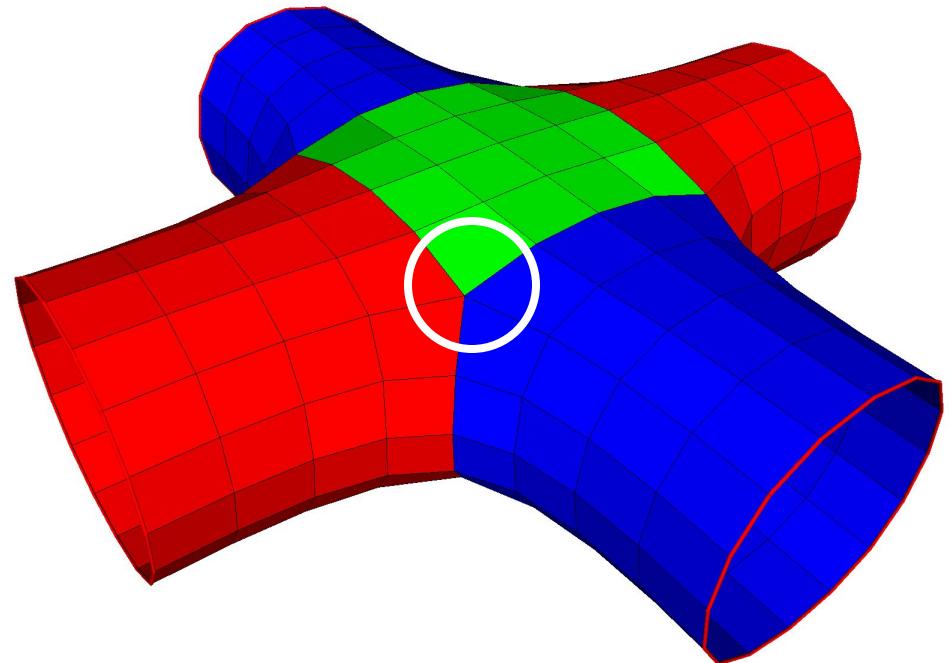
Bezier (NURBS) patches aren't the greatest

- NURBS patches are $n \times m$, forming a mesh of quadrilaterals.
 - What if you wanted triangles or pentagons?
 - A NURBS dodecahedron?
 - What if you wanted vertices of valence other than four?
- NURBS expressions for triangular patches, and more, do exist; but they're cumbersome.



Problems with NURBS patches

- Joining NURBS patches with C_n continuity across an edge is challenging.
- What happens to continuity at corners where the number of patches meeting isn't exactly four?
- Animation is tricky: bending and blending are doable, but not easy.



Sadly, the world isn't made up of shapes that can always be made from one smoothly-deformed rectangular surface.

Subdivision surfaces

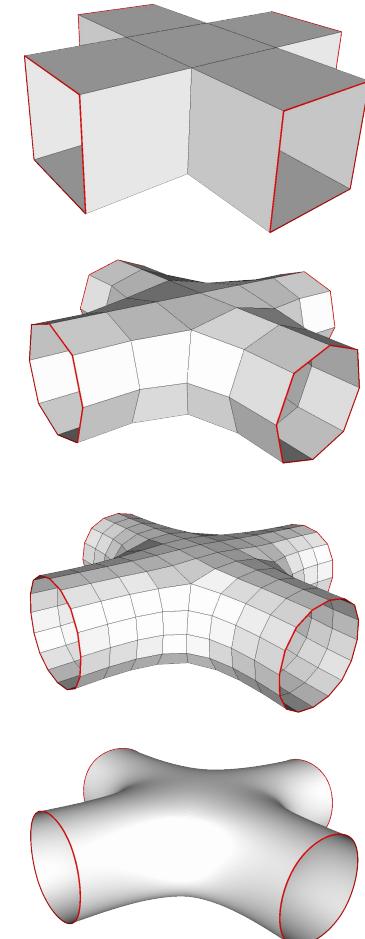
- Beyond shipbuilding: we want guaranteed continuity, without having to build everything out of rectangular patches.
 - Applications include CAD/CAM, 3D printing, museums and scanning, medicine, movies...
- The solution: *subdivision surfaces.*



Geri's Game, by Pixar (1997)

Subdivision surfaces

- Instead of ticking a parameter t along a parametric curve (or the parameters u, v over a parametric grid), subdivision surfaces repeatedly refine from a coarse set of *control points*.
- Each step of refinement adds new faces and vertices.
- The process converges to a smooth *limit surface*.

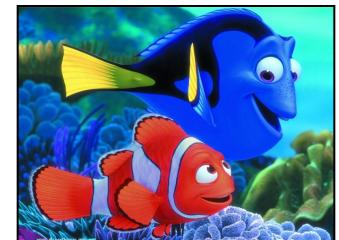


Subdivision surfaces – History

- de Rahm described a 2D (curve) subdivision scheme in 1947; rediscovered in 1974 by Chaikin
- Concept extended to 3D (surface) schemes by two separate groups during 1978:
 - Doo and Sabin found a biquadratic surface
 - Catmull and Clark found a bicubic surface
- Subsequent work in the 1980s (Loop, 1987; Dyn [Butterfly subdivision], 1990) led to tools suitable for CAD/CAM and animation

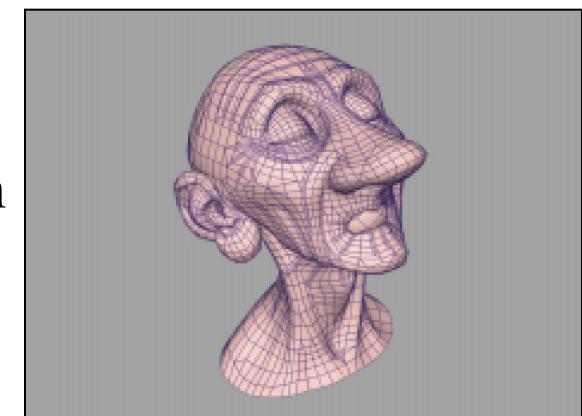
Subdivision surfaces and the movies

- Pixar first demonstrated subdivision surfaces in 1997 with Geri's Game.
 - Up until then they'd done everything in NURBS (Toy Story, A Bug's Life.)
 - From 1999 onwards everything they did was with subdivision surfaces (Toy Story 2, Monsters Inc, Finding Nemo...)
 - Two decades on, it's all heavily customized.
- It's not clear what Dreamworks uses, but they have recent patents on subdivision techniques.



Useful terms

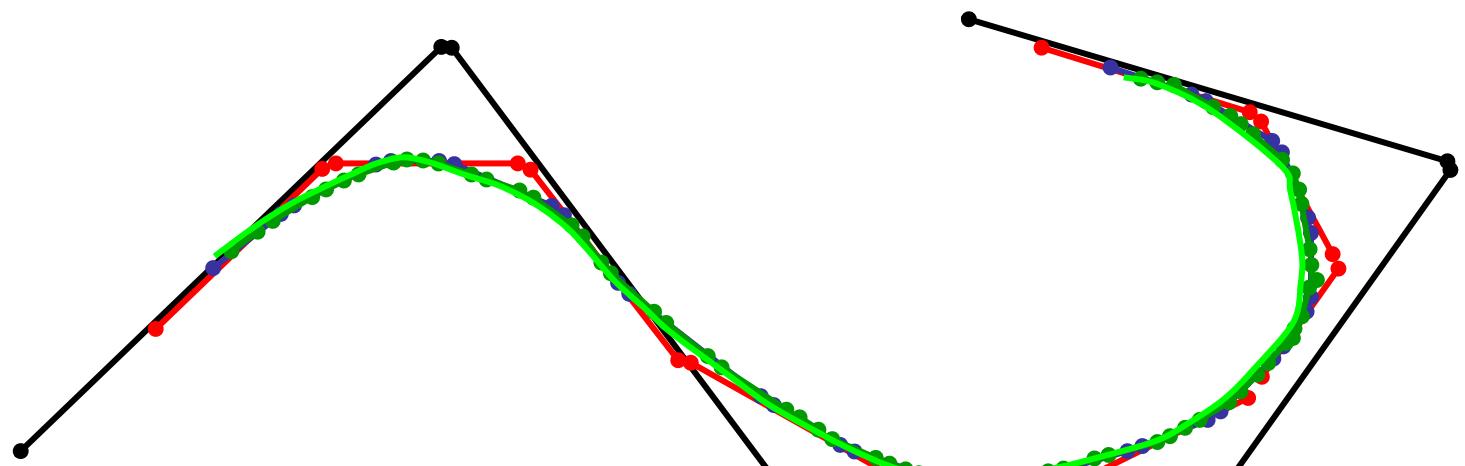
- A scheme which describes a 1D curve (even if that curve is travelling in 3D space, or higher) is called *univariate*, referring to the fact that the limit curve can be approximated by a polynomial in one variable (t).
- A scheme which describes a 2D surface is called *bivariate*, the limit surface can be approximated by a u, v parameterization.
- A scheme which retains and passes through its original control points is called an *interpolating* scheme.
- A scheme which moves away from its original control points, converging to a limit curve or surface nearby, is called an *approximating* scheme.



Control surface for Geri's head
175

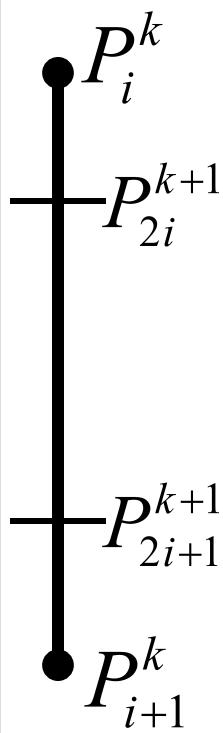
How it works

- Example: *Chaikin curve subdivision (2D)*
 - On each edge, insert new control points at $\frac{1}{4}$ and $\frac{3}{4}$ between old vertices; delete the old points
 - The *limit curve* is C1 everywhere (despite the poor figure.)



Notation

Chaikin can be written programmatically as:



$$P_{2i}^{k+1} = \left(\frac{3}{4}\right)P_i^k + \left(\frac{1}{4}\right)P_{i+1}^k \quad \leftarrow Even$$

$$P_{2i+1}^{k+1} = \left(\frac{1}{4}\right)P_i^k + \left(\frac{3}{4}\right)P_{i+1}^k \quad \leftarrow Odd$$

...where k is the ‘generation’; each generation will have twice as many control points as before.
Notice the different treatment of generating odd and even control points.
Borders (terminal points) are a special case.

Notation

Chaikin can be written in vector notation as:

$$\begin{bmatrix} \vdots \\ P_{2i-2}^{k+1} \\ P_{2i-1}^{k+1} \\ P_{2i}^{k+1} \\ P_{2i+1}^{k+1} \\ P_{2i+2}^{k+1} \\ P_{2i+3}^{k+1} \\ \vdots \end{bmatrix} = \frac{1}{4} \begin{bmatrix} \vdots \\ 0 & 3 & 1 & 0 & 0 & 0 \\ 0 & 1 & 3 & 0 & 0 & 0 \\ 0 & 0 & 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 3 & 0 & 0 \\ 0 & 0 & 0 & 3 & 1 & 0 \\ 0 & 0 & 0 & 1 & 3 & 0 \\ \vdots & & & & & \vdots \\ \vdots & & & & & \vdots \end{bmatrix} \begin{bmatrix} \vdots \\ P_{i-2}^k \\ P_{i-1}^k \\ P_i^k \\ P_{i+1}^k \\ P_{i+2}^k \\ P_{i+3}^k \\ \vdots \end{bmatrix}$$

Notation

- The standard notation compresses the scheme to a *kernel*:
 - $h = (1/4)[\dots, 0, 0, 1, 3, 3, 1, 0, 0, \dots]$
- The kernel interlaces the odd and even rules.
- It also makes matrix analysis possible: eigenanalysis of the matrix form can be used to prove the continuity of the subdivision limit surface.
 - The details of analysis are fascinating, lengthy, and sadly beyond the scope of this course
- The limit curve of Chaikin is a quadratic B-spline!

Reading the kernel

Consider the kernel

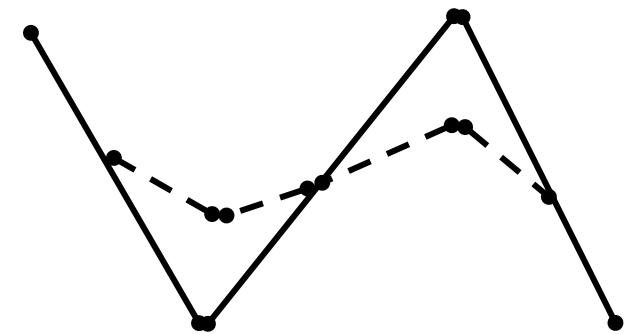
$$h=(1/8)[\dots, 0, 0, 1, 4, 6, 4, 1, 0, 0, \dots]$$

You would read this as

$$P_{2i}^{k+1} = (\textstyle{1}/{8})(P_{i-1}^k + 6P_i^k + P_{i+1}^k)$$

$$P_{2i+1}^{k+1} = (\textstyle{1}/{8})(4P_i^k + 4P_{i+1}^k)$$

The limit curve is provably C2-continuous.



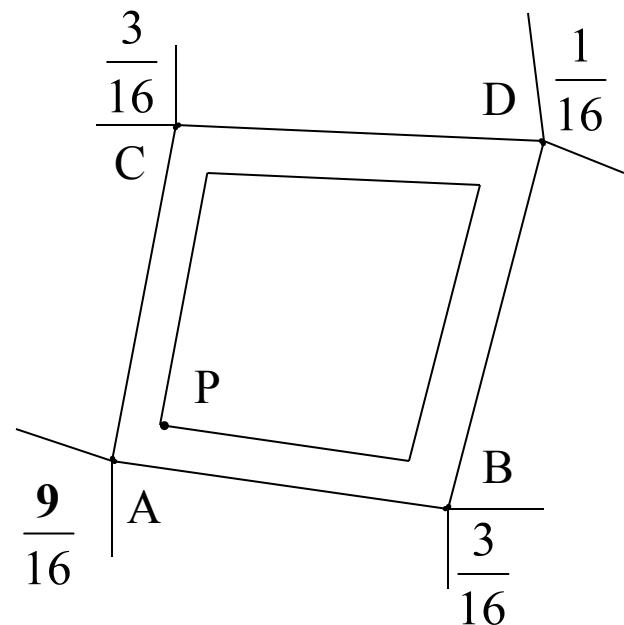
Making the jump to 3D: Doo-Sabin

Doo-Sabin takes Chaikin to 3D:

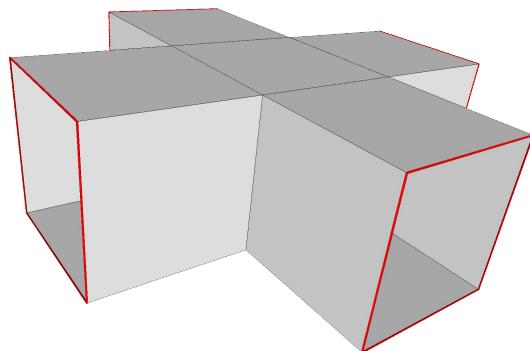
$$\begin{aligned} P = & (9/16) A + \\ & (3/16) B + \\ & (3/16) C + \\ & (1/16) D \end{aligned}$$

This replaces every old vertex with four new vertices.

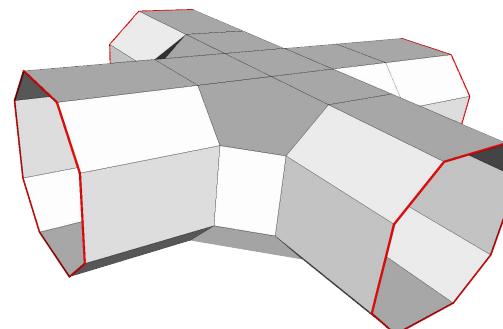
The limit surface is biquadratic,
C1 continuous everywhere.



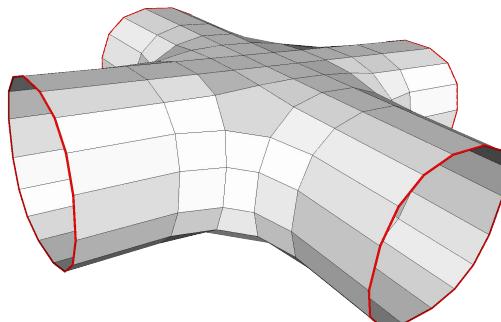
Doo-Sabin in action



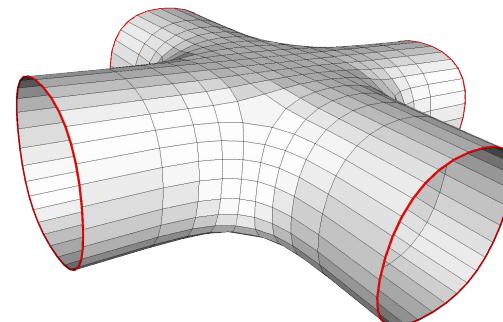
(0) 18 faces



(1) 54 faces



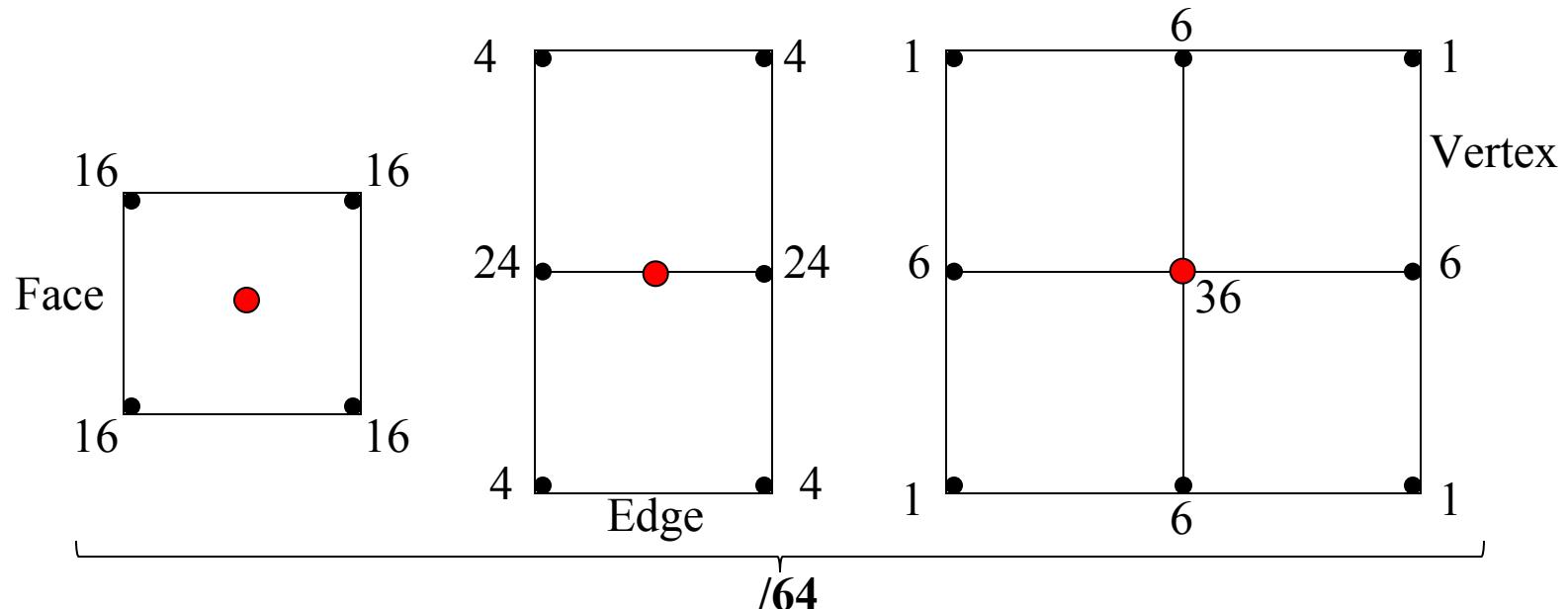
(2) 190 faces



(3) 702 faces

Catmull-Clark

- *Catmull-Clark* is a bivariate approximating scheme with kernel $h=(1/8)[1,4,6,4,1]$.
 - Limit surface is bicubic, C2-continuous.

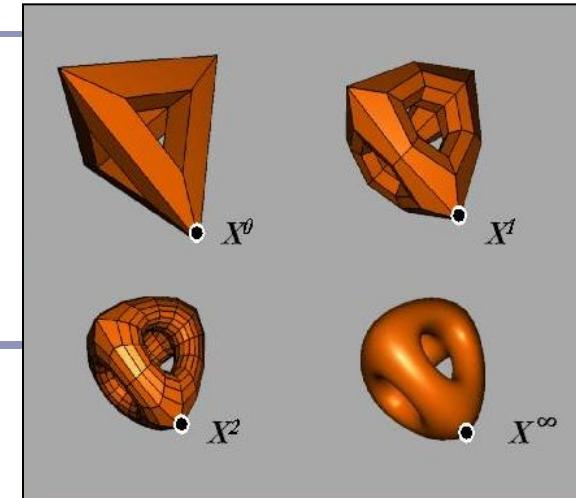


Catmull-Clark

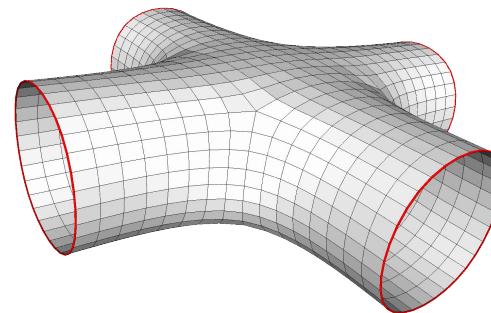
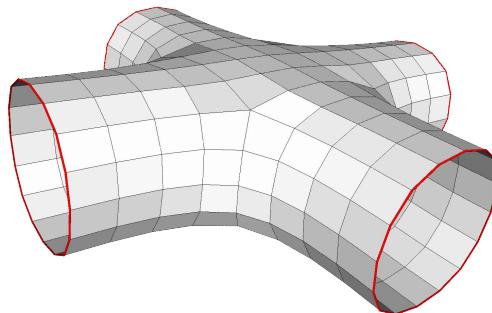
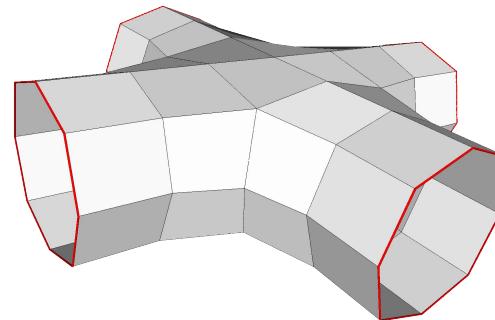
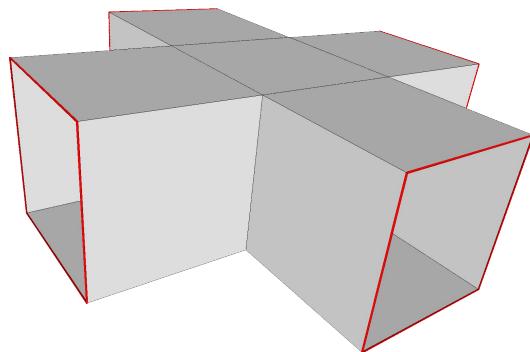
Getting tensor again:

$$\frac{1}{8} \begin{bmatrix} 1 \\ 4 \\ 6 \\ 4 \\ 1 \end{bmatrix} \otimes \frac{1}{8} \begin{bmatrix} 1 \\ 4 \\ 6 \\ 4 \\ 1 \end{bmatrix} = \frac{1}{64} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

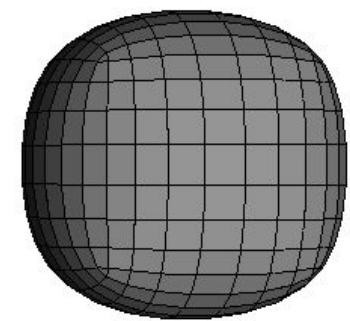
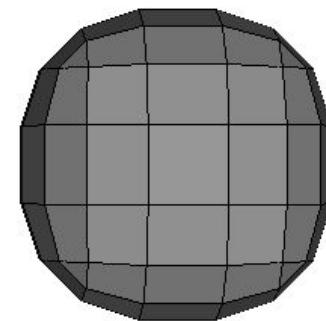
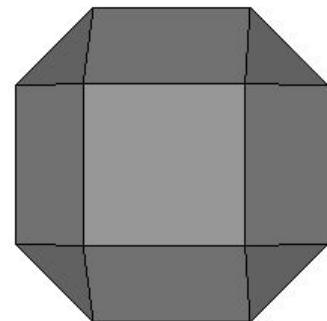
Vertex rule Face rule Edge rule



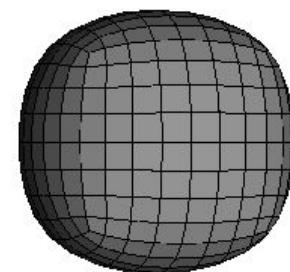
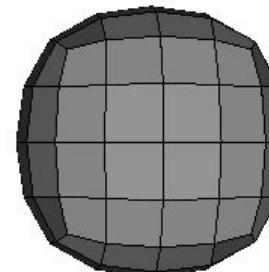
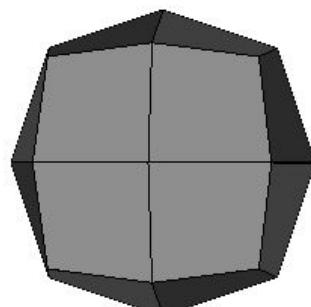
Catmull-Clark in action



Catmull-Clark vs Doo-Sabin



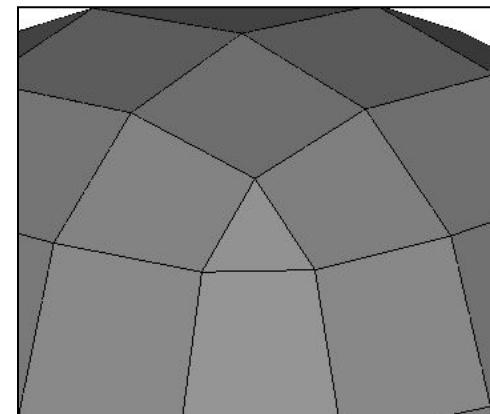
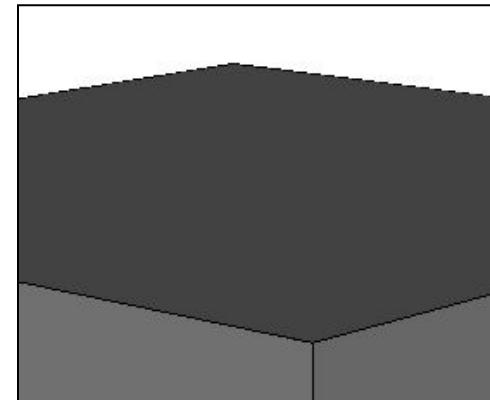
Doo-Sabin



Catmull-Clark

Extraordinary vertices

- Catmull-Clark and Doo-Sabin both operate on quadrilateral meshes.
 - All faces have four boundary edges
 - All vertices have four incident edges
- What happens when the mesh contains *extraordinary* vertices or faces?
 - For many schemes, adaptive weights exist which can continue to guarantee at least some (non-zero) degree of continuity, but not always the best possible.
- CC replaces extraordinary faces with extraordinary vertices; DS replaces extraordinary vertices with extraordinary faces.



Detail of Doo-Sabin at cube corner

Extraordinary vertices: Catmull-Clark

Catmull-Clark vertex rules generalized for extraordinary vertices:

- Original vertex:
 $(4n-7) / 4n$
- Immediate neighbors in the one-ring:
 $3/2n^2$
- Interleaved neighbors in the one-ring:
 $1/4n^2$

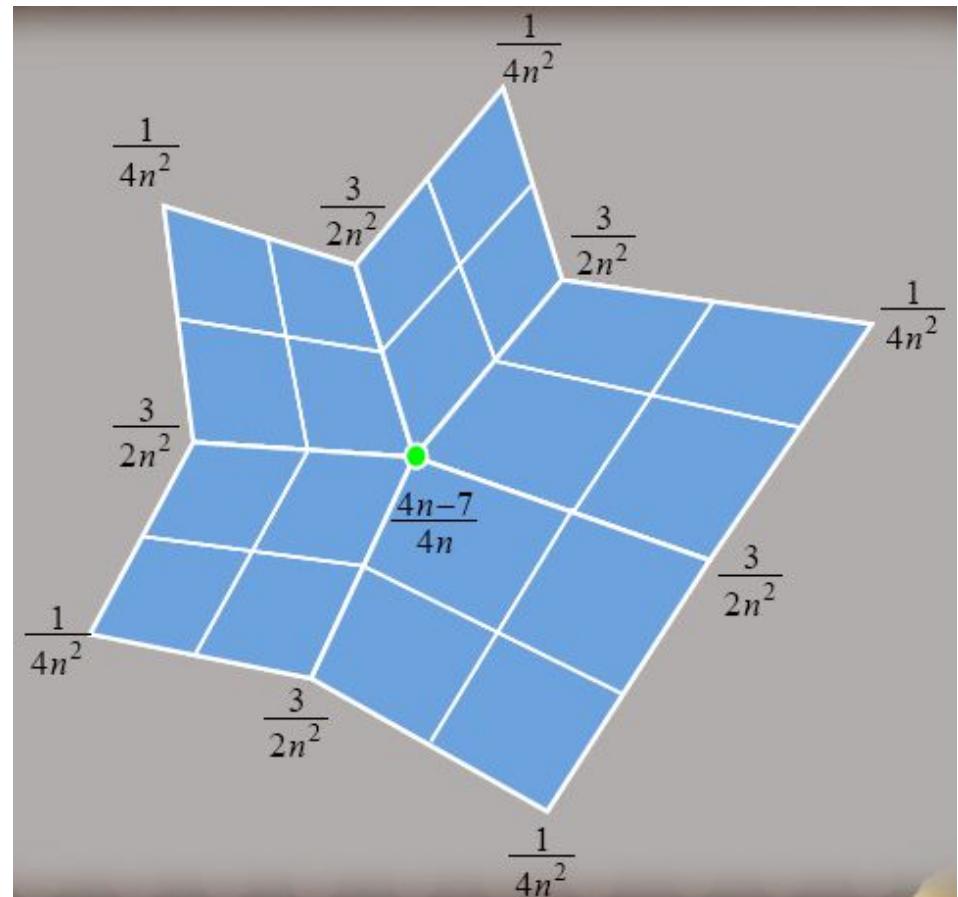
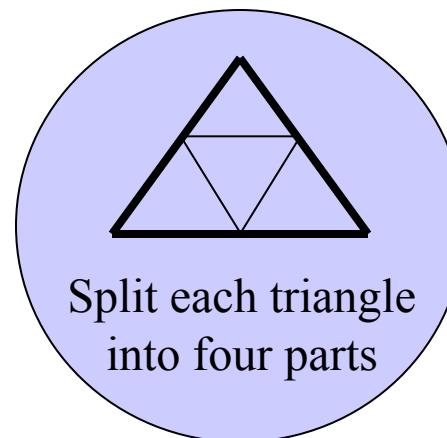
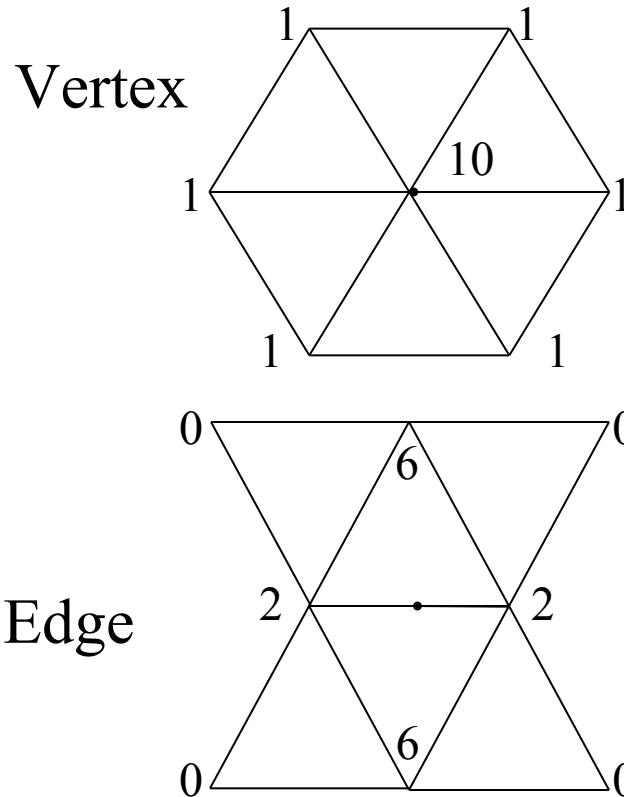


Image source: "Next-Generation Rendering of Subdivision Surfaces", Ignacio Castaño, SIGGRAPH 2008

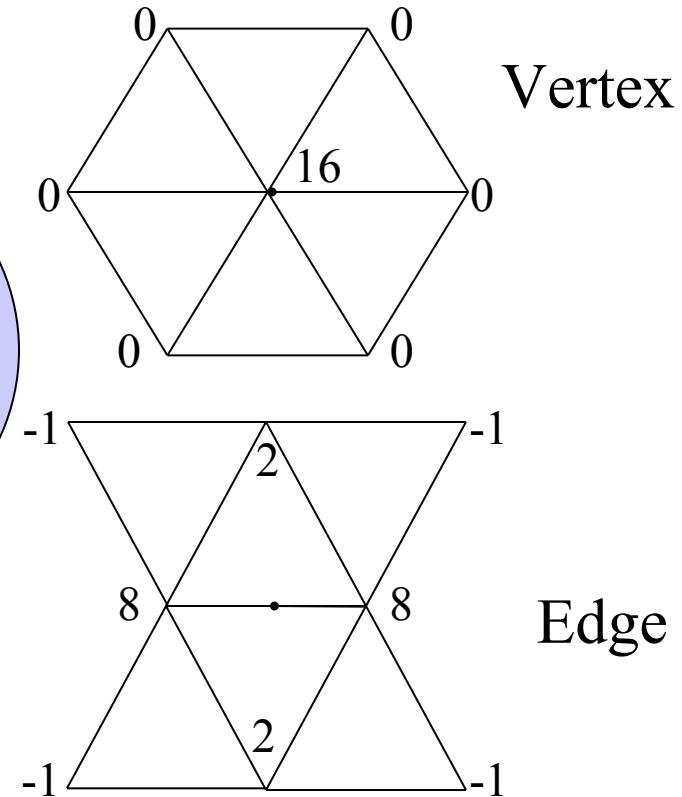
Schemes for simplicial (triangular) meshes

- *Loop scheme:*

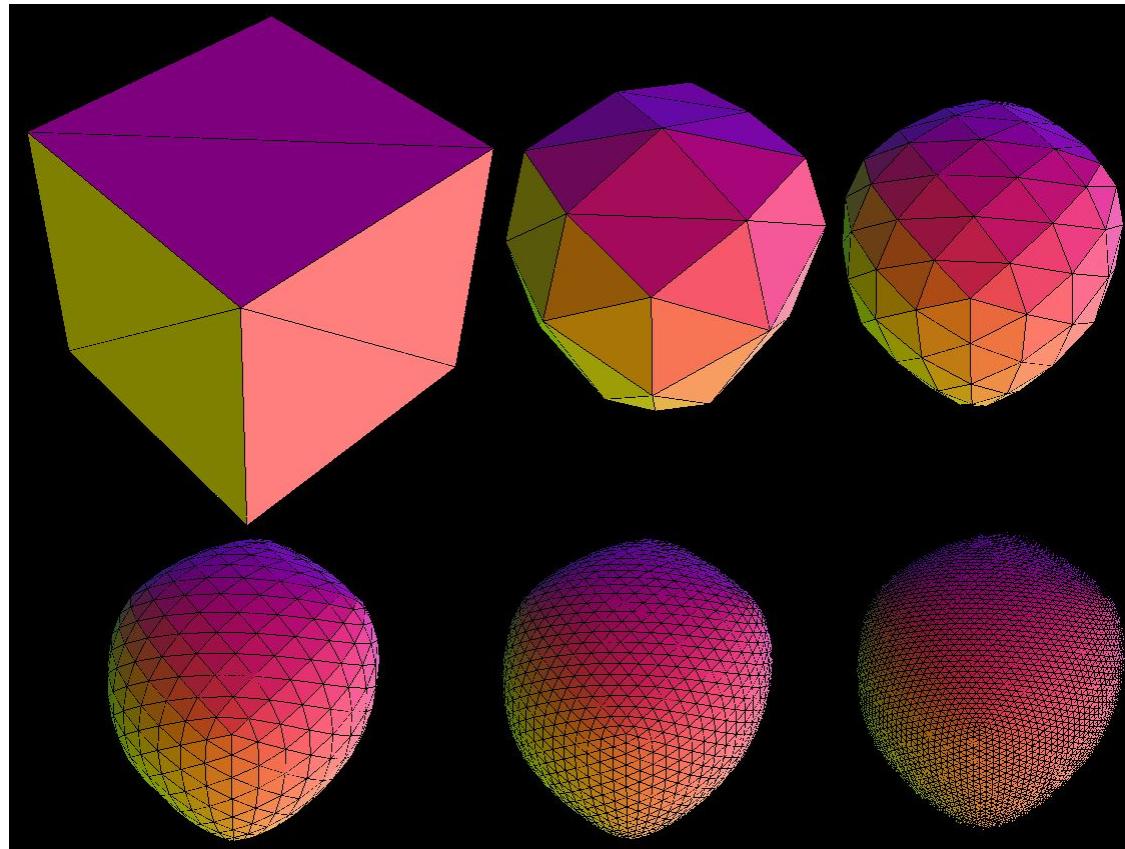


(All weights are /16)

- *Butterfly scheme:*



Loop subdivision

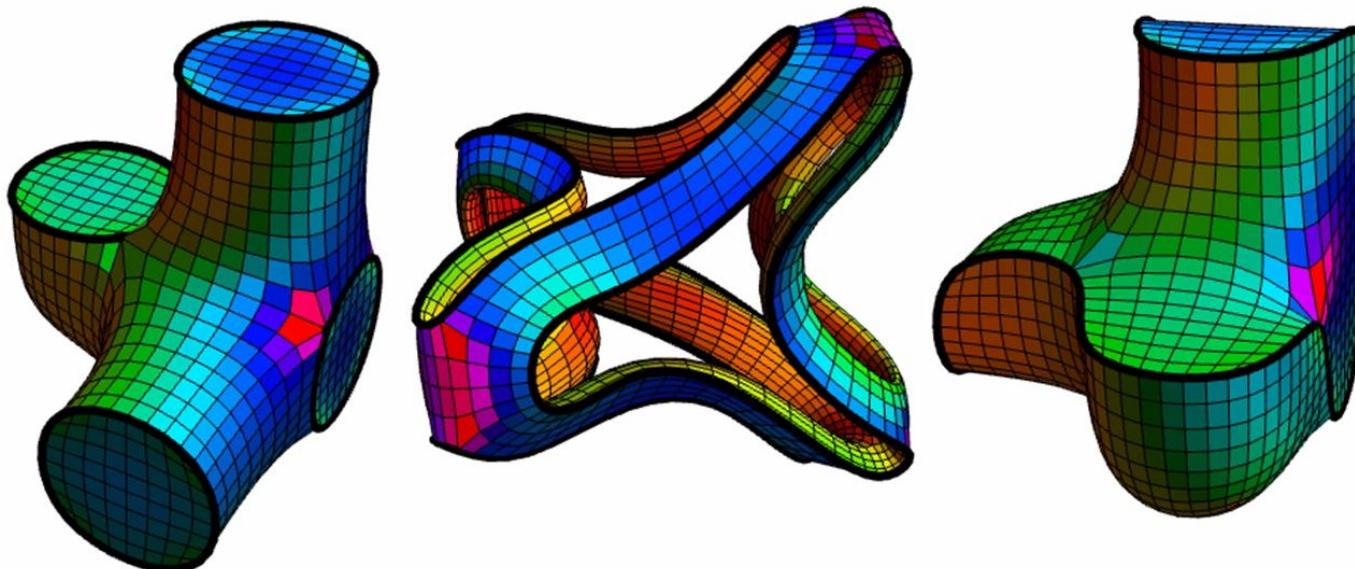


Loop subdivision in action. The asymmetry is due to the choice of face diagonals.

Image by Matt Fisher, <http://www.its.caltech.edu/~matthewf/Chatter/Subdivision.html>

Creases

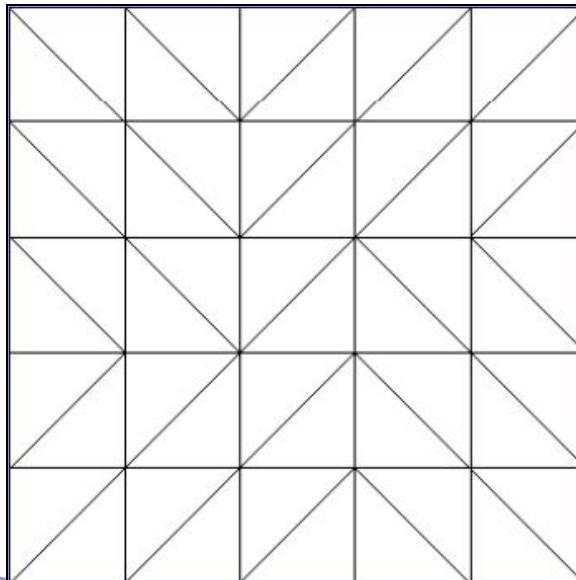
Extensions exist for most schemes to support *creases*, vertices and edges flagged for partial or hybrid subdivision.



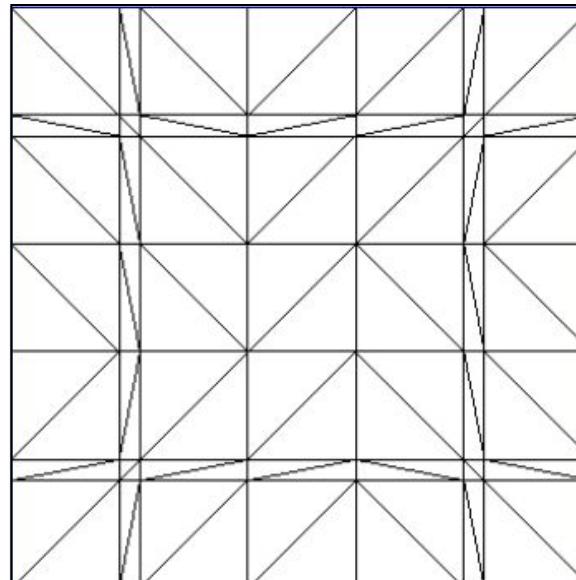
Still from “Volume Enclosed by Subdivision Surfaces with Sharp Creases” by Jan Hakenberg, Ulrich Reif, Scott Schaefer, Joe Warren
<http://vixra.org/pdf/1406.0060v1.pdf>

Continuous level of detail

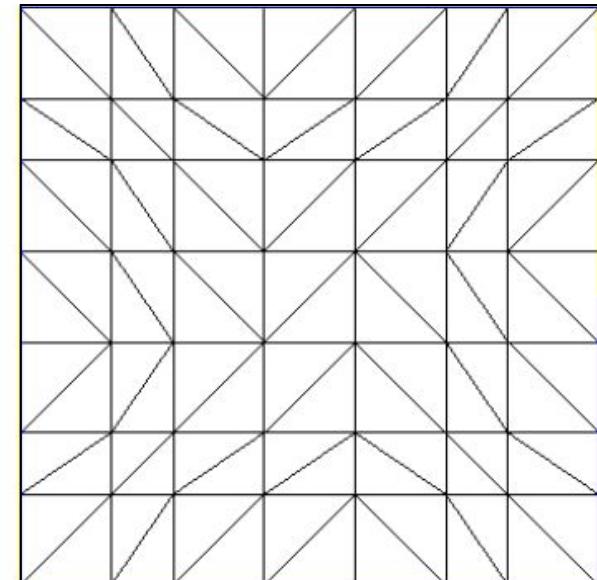
For live applications (e.g. games) can compute *continuous* level of detail, e.g. as a function of distance:



Level 5



Level 5.2



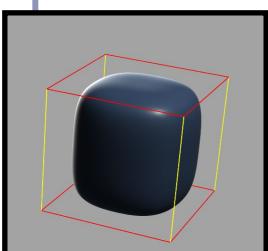
Level 5.8

Direct evaluation of the limit surface

- In the 1999 paper *Exact Evaluation Of Catmull-Clark Subdivision Surfaces at Arbitrary Parameter Values*, Jos Stam (now at Alias|Wavefront) describes a method for finding the exact final positions of the CC limit surface.
 - His method is based on calculating the tangent and normal vectors to the limit surface and then shifting the control points out to their final positions.
 - What's particularly clever is that he gives exact evaluation at the extraordinary vertices. (Non-trivial.)

Bounding boxes and convex hulls for subdivision surfaces

- The limit surface is (the weighted average of (the weighted averages of (the weighted averages of (repeat for eternity...)))) the original control points.
- This implies that for any scheme where all weights are positive and sum to one, the limit surface lies entirely within the convex hull of the original control points.
- For schemes with negative weights:
 - Let $L = \max_t \sum_i |N_i(t)|$ be the greatest sum throughout parameter space of the absolute values of the weights.
 - For a scheme with negative weights, L will exceed 1.
 - Then the limit surface must lie within the convex hull of the original control points, expanded unilaterally by a ratio of $(L-1)$.



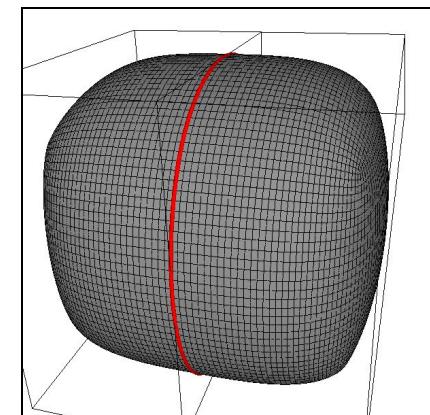
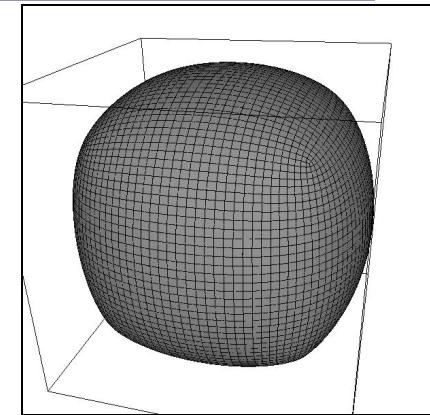
Splitting a subdivision surface

Many algorithms rely on subdividing a surface and examining the bounding boxes of smaller facets.

- Rendering, ray/surface intersections...

It's not enough just to delete half your control points: the limit surface will change (see right)

- Need to include all control points from the previous generation, which influence the limit surface in this smaller part.

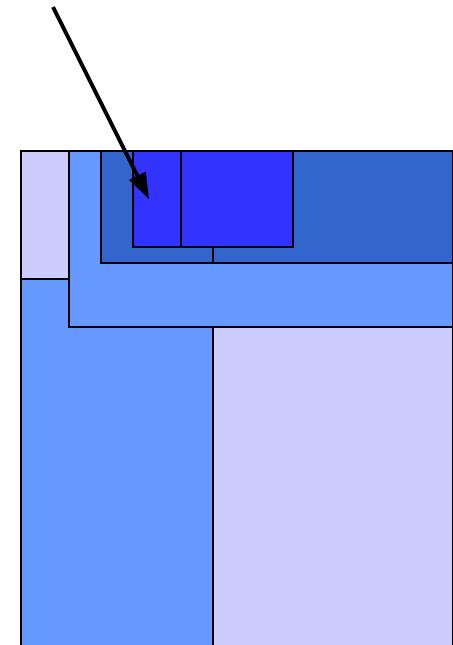


(Top) 5x Catmull-Clark subdivision of a cube

(Bottom) 5x Catmull-Clark subdivision of two halves of a cube; the limit surfaces are clearly different.

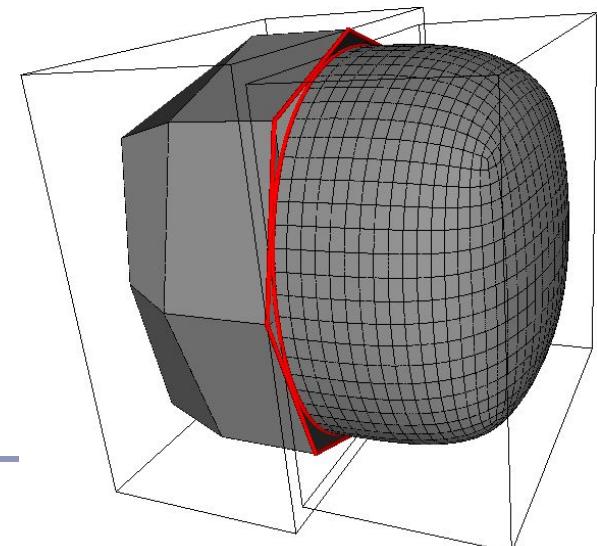
Ray/surface intersection

- To intersect a ray with a subdivision surface, we recursively split and split again, discarding all portions of the surface whose bounding boxes / convex hulls do not lie on the line of the ray.
- Any subsection of the surface which is ‘close enough’ to flat is treated as planar and the ray/plane intersection test is used.
- This is essentially a binary tree search for the nearest point of intersection.
 - You can optimize by sorting your list of subsurfaces in increasing order of distance from the origin of the ray.



Rendering subdivision surfaces

- The algorithm to render any subdivision surface is exactly the same as for Bezier curves:
“If the surface is simple enough, render it directly; otherwise split it and recurse.”
- One fast test for “simple enough” is,
“Is the convex hull of the limit surface sufficiently close to flat?”
- Caveat: splitting a surface and subdividing one half but not the other can lead to tears where the different resolutions meet. →



Rendering subdivision surfaces on the GPU

- Subdivision algorithms have been ported to the GPU using geometry (tesselation) shaders.
 - This subdivision can be done completely independently of geometry, imposing no demands on the CPU.
 - Uses a complex blend of precalculated weights and shader logic
 - Impressive effects in use at id, Valve, et al

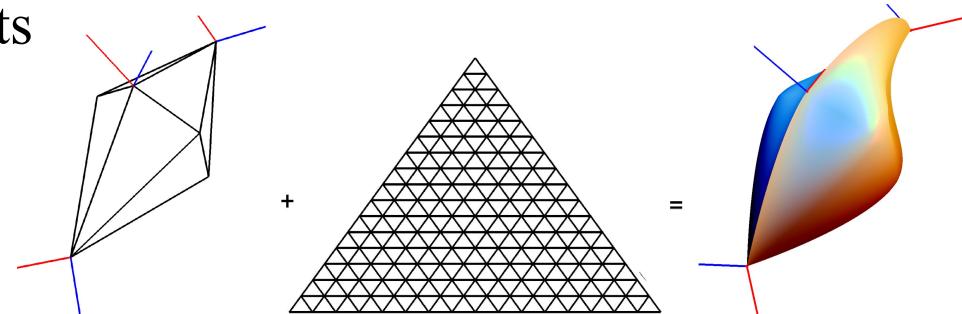


Figure from *Generic Mesh Renement on GPU*,
Tamy Boubekeur & Christophe Schlick (2005)
LaBRI INRIA CNRS University of Bordeaux, France

Subdivision Schemes—A partial list

- Approximating
 - Quadrilateral
 - $(1/2)[1,2,1]$
 - $(1/4)[1,3,3,1]$
(Doo-Sabin)
 - $(1/8)[1,4,6,4,1]$
(Catmull-Clark)
 - *Mid-Edge*
 - Triangles
 - Loop

- Interpolating
 - Quadrilateral
 - *Kobbelt*
 - Triangle
 - Butterfly
 - “ $\sqrt{3}$ ” *Subdivision*

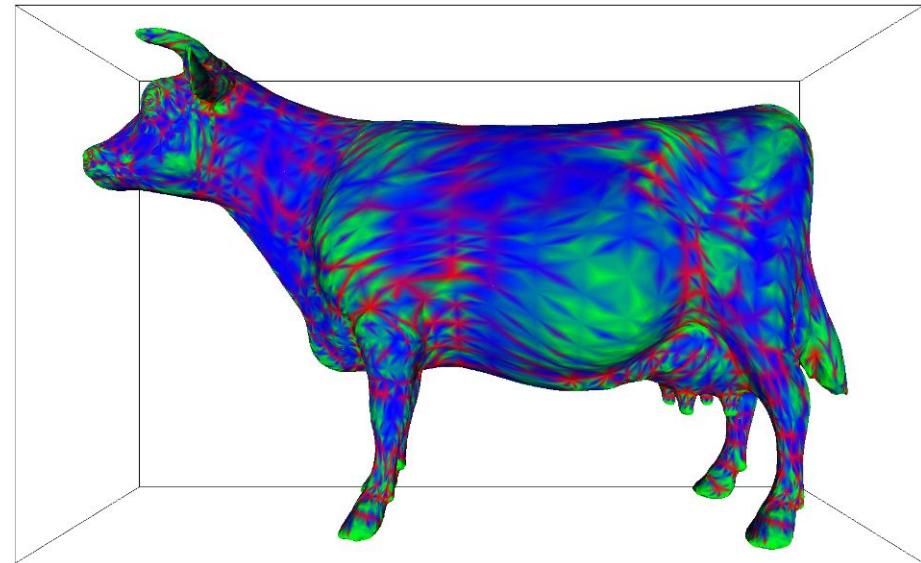
Many more exist, some much more complex

This is a major topic of ongoing research

References

- Catmull, E., and J. Clark. “Recursively Generated B-Spline Surfaces on Arbitrary Topological Meshes.” *Computer Aided Design*, 1978.
- Dyn, N., J. A. Gregory, and D. A. Levin. “Butterfly Subdivision Scheme for Surface Interpolation with Tension Control.” *ACM Transactions on Graphics*. Vol. 9, No. 2 (April 1990): pp. 160–169.
- Halstead, M., M. Kass, and T. DeRose. “Efficient, Fair Interpolation Using Catmull-Clark Surfaces.” *Siggraph '93*. p. 35.
- Zorin, D. “Stationary Subdivision and Multiresolution Surface Representations.” Ph.D. diss., California Institute of Technology, 1997
- Ignacio Castano, “Next-Generation Rendering of Subdivision Surfaces.” Siggraph '08, <http://developer.nvidia.com/object/siggraph-2008-Subdiv.html>
- Dennis Zorin’s SIGGRAPH course, “Subdivision for Modeling and Animation”, <http://www.mrl.nyu.edu/publications/subdiv-course2000/>

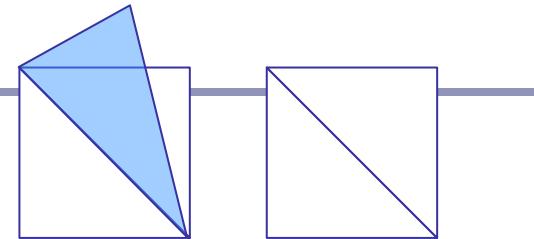
Advanced Graphics



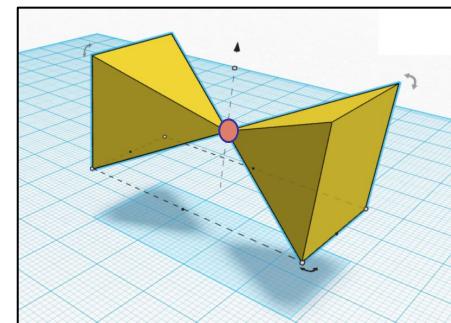
Surfaces - Methods and Mathematics

Terminology

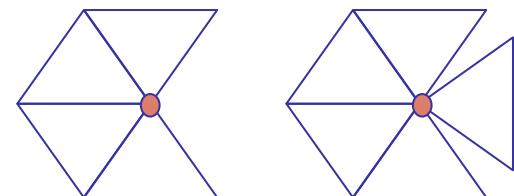
- We'll be focusing on *discrete* (as opposed to continuous) representation of geometry; i.e., polygon meshes
 - Many rendering systems limit themselves to triangle meshes
 - Many require that the mesh be *manifold*
- In a *closed manifold* polygon mesh:
 - Exactly two triangles meet at each edge
 - The faces meeting at each vertex belong to a single, connected loop of faces
- In a *manifold with boundary*:
 - At most two triangles meet at each edge
 - The faces meeting at each vertex belong to a single, connected strip of faces



Edge: Non-manifold vs manifold



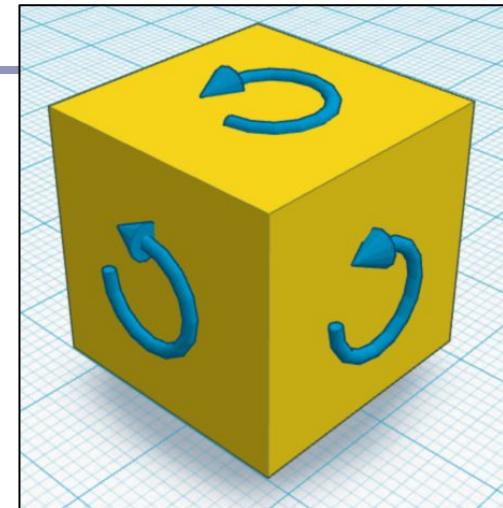
Non-manifold vertex



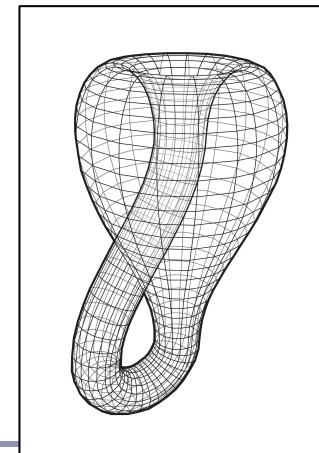
Vertex: Good boundary vs bad

Terminology

- We say that a surface is *oriented* if:
 - a. the vertices of every face are stored in a fixed order
 - b. if vertices i, j appear in both faces $f1$ and $f2$, then the vertices appear in order i, j in one and j, i in the other
- We say that a surface is *embedded* if, informally, “nothing pokes through”:
 - a. No vertex, edge or face shares any point in space with any other vertex, edge or face except where dictated by the data structure of the polygon mesh
- A closed, embedded surface must separate 3-space into two parts: a bounded *interior* and an unbounded *exterior*.



A cube with “anti-clockwise” oriented faces



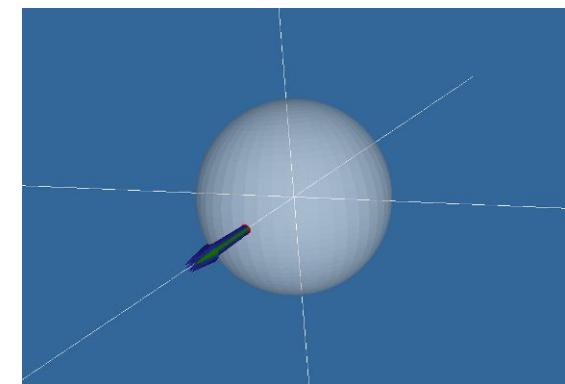
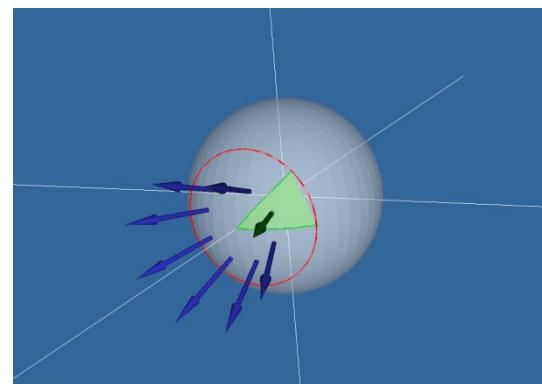
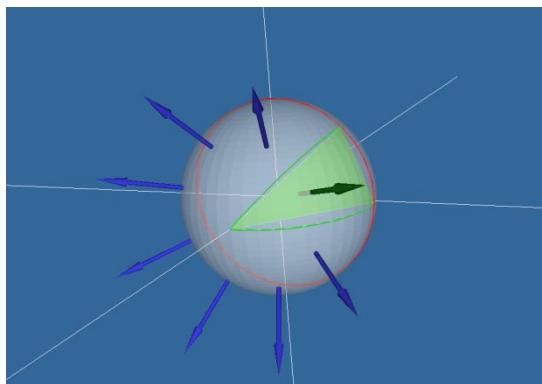
Klein bottle:
not an
embedded
surface.

Also, terrible
for holding
drinks.

Normal at a vertex

Expressed as a limit,

The *normal of surface S at point P* is the limit of the cross-product between two (non-collinear) vectors from P to the set of points in S at a distance r from P as r goes to zero. [Excluding orientation.]



Normal at a vertex

Using the limit definition, is the ‘normal’ to a discrete surface necessarily a vector?

- The normal to the surface at any point on a face is a constant vector.
- The ‘normal’ to the surface at any edge is an arc swept out on a unit sphere between the two normals of the two faces.
- The ‘normal’ to the surface at a vertex is a space swept out on the unit sphere between the normals of all of the adjacent faces.

Finding the normal at a vertex

Take the weighted average of the normals of surrounding polygons, weighted by each polygon's *face angle* at the vertex

Face angle: the angle α formed at the vertex v by the vectors to the next and previous vertices in the face F

$$\alpha(F, v_i) = \cos^{-1} \left(\frac{v_{i+1} - v_i}{|v_{i+1} - v_i|} \bullet \frac{v_{i-1} - v_i}{|v_{i-1} - v_i|} \right)$$

$$N(v) = \frac{\sum_F \alpha(F, v) N_F}{|\sum_F \alpha(F, v)|}$$

Note: In this equation, \arccos implies a convex polygon. Why?

Gaussian curvature on smooth surfaces

Informally speaking, the *curvature* of a surface expresses “how flat the surface isn’t”.

- One can measure the directions in which the surface is curving *most*; these are the directions of *principal curvature*, k_1 and k_2 .
- The product of k_1 and k_2 is the scalar *Gaussian curvature*.

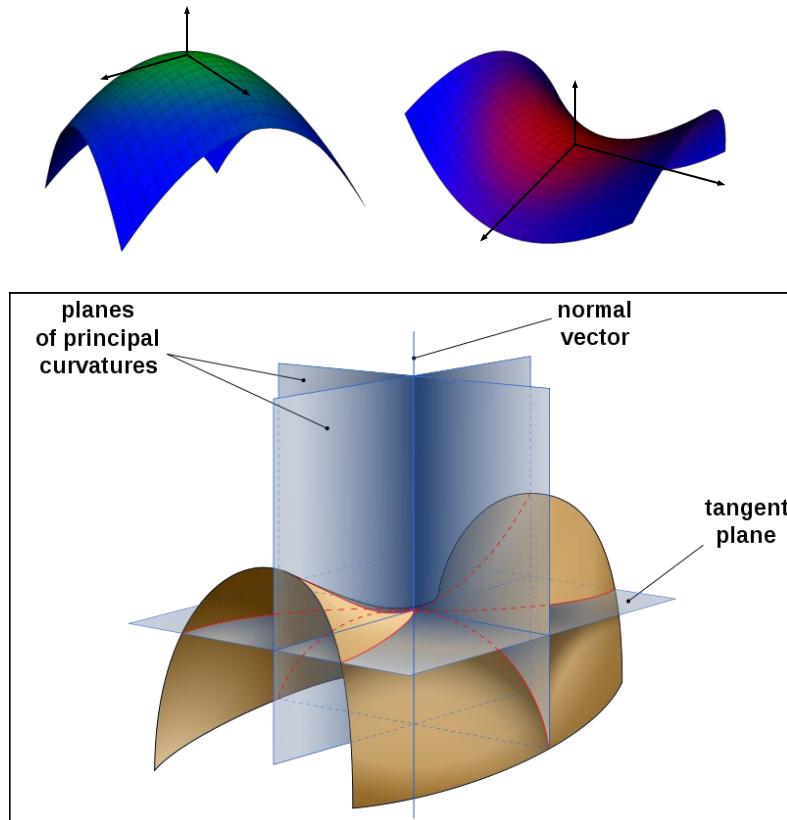


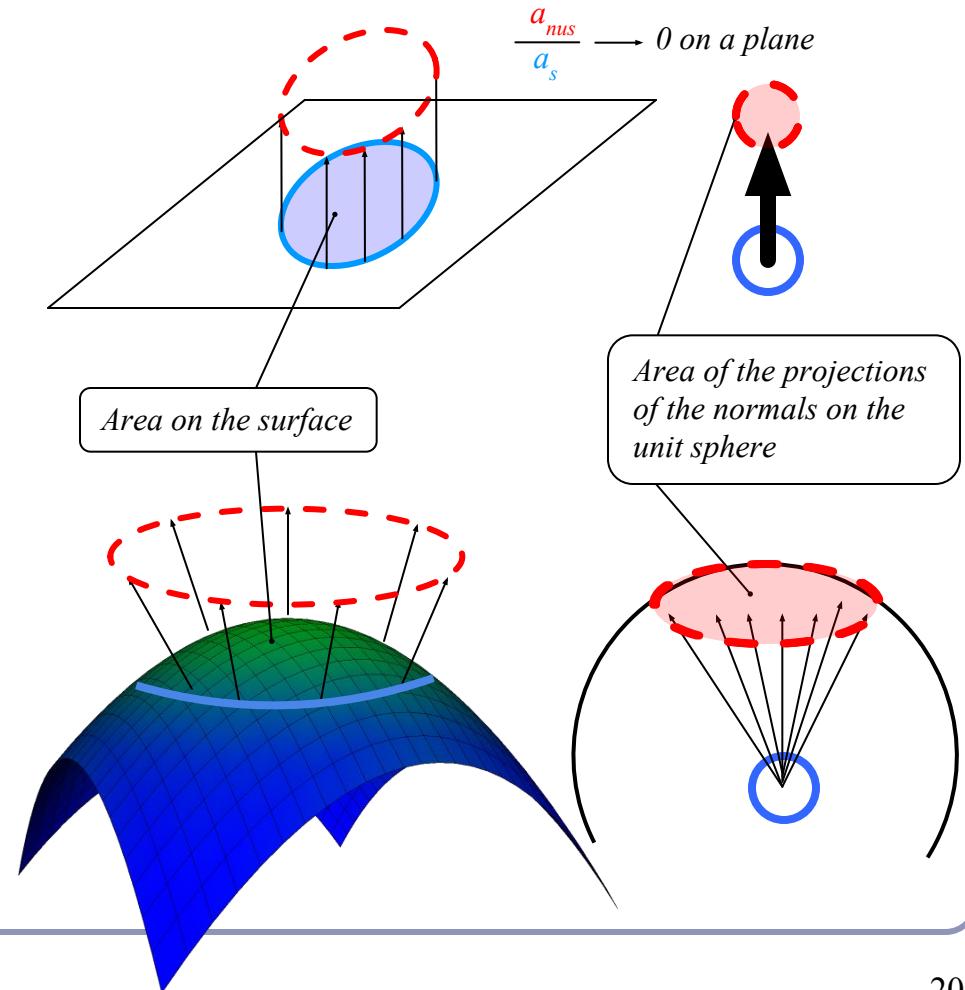
Image by Eric Gaba, from Wikipedia

Gaussian curvature on smooth surfaces

Formally, the *Gaussian curvature of a region* on a surface is the ratio between the **area of the surface of the unit sphere swept out by the normals of that region** and the **area of the region itself**.

The Gaussian curvature of a point is the limit of this ratio as the region tends to zero area.

$$\frac{a_{nus}}{a_s} \xrightarrow{} r^2 \text{ on a sphere of radius } r \\ (\text{please pretend that this is a sphere})$$



Gaussian curvature on discrete surfaces

On a discrete surface, normals do not vary smoothly: the normal to a face is constant on the face, and at edges and vertices the normal is—strictly speaking—undefined.

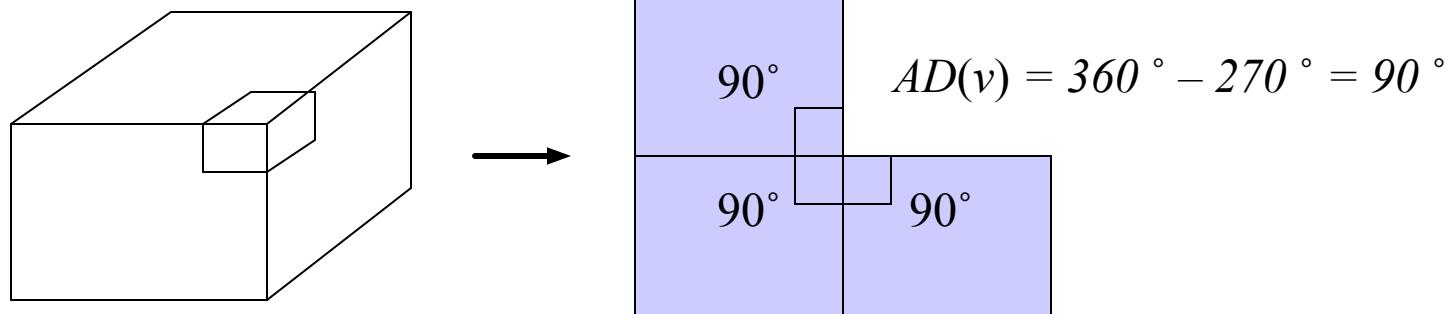
- Normals change instantaneously (as one's point of view travels across an edge from one face to another) or not at all (as one's point of view travels within a face.)

The Gaussian curvature of the surface of any polyhedral mesh is **zero** everywhere except at the vertices, where it is **infinite**.

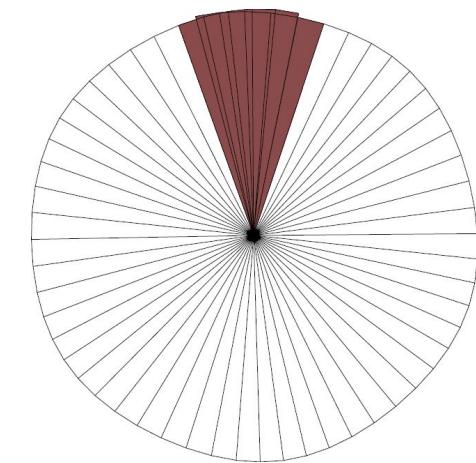
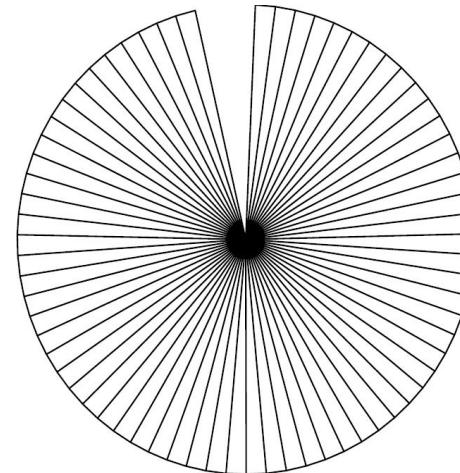
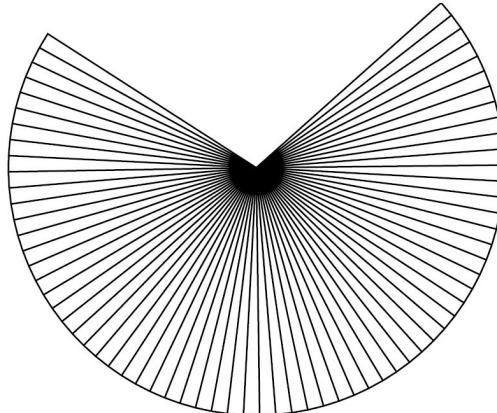
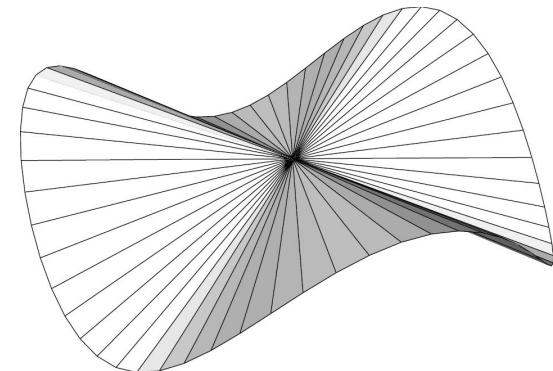
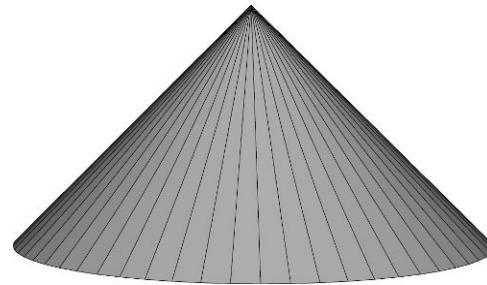
Angle deficit – a better solution for measuring discrete curvature

The *angle deficit* $AD(v)$ of a vertex v is defined to be two π minus the sum of the face angles of the adjacent faces.

$$AD(v) = 2\pi - \sum_F \alpha(F, v)$$



Angle deficit

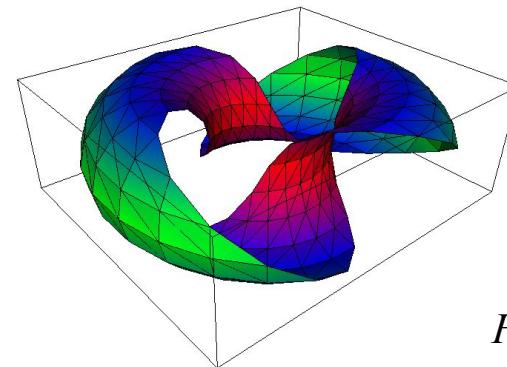
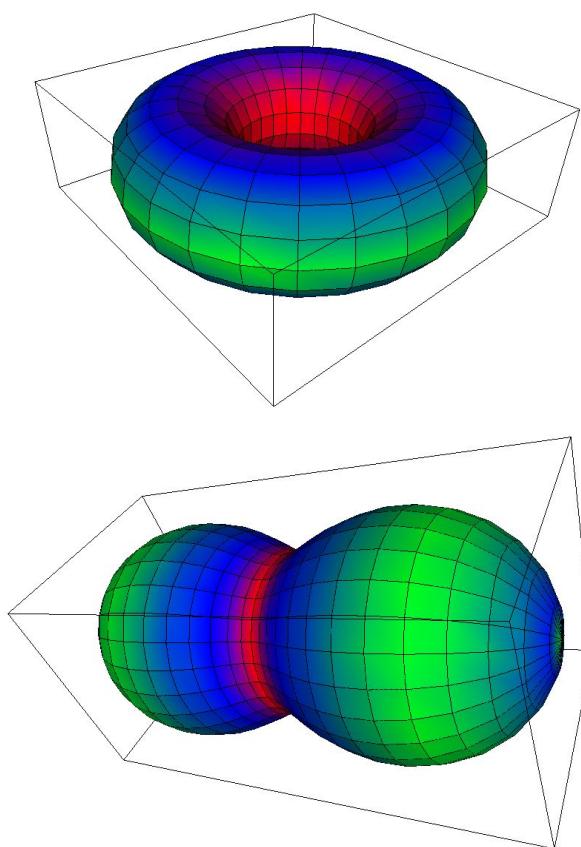


High angle deficit

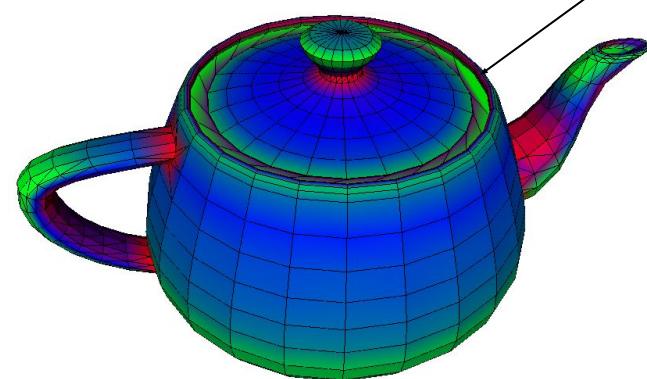
Low angle deficit

Negative angle deficit

Angle deficit

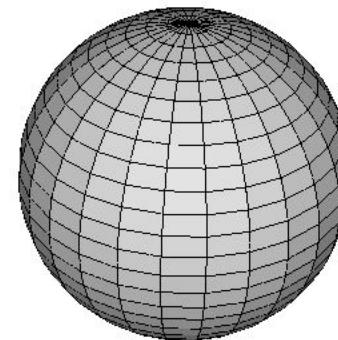


Hmmm...

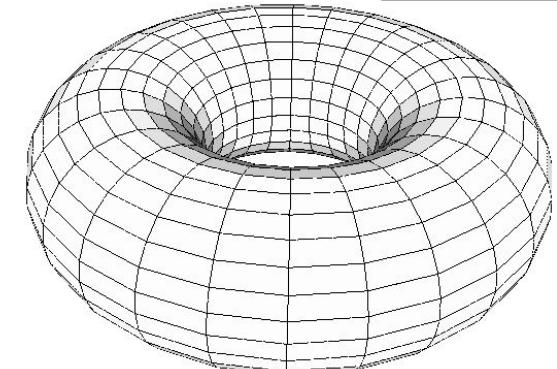


Genus, Poincaré and the Euler Characteristic

- Formally, the *genus* g of a closed surface is
 - ...“a topologically invariant property of a surface defined as the largest number of nonintersecting simple closed curves that can be drawn on the surface without separating it.”
--*mathworld.com*
- Informally, it’s the number of coffee cup handles in the surface.



Genus 0



Genus 1

Genus, Poincaré and the Euler Characteristic

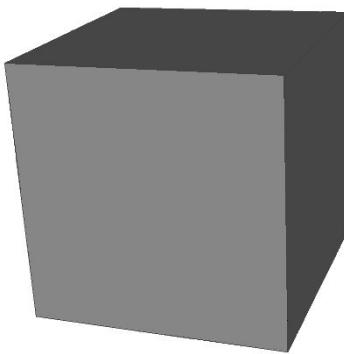
Given a polyhedral surface S without border where:

- V = the number of vertices of S ,
- E = the number of edges between those vertices,
- F = the number of faces between those edges,
- χ is the *Euler Characteristic* of the surface,

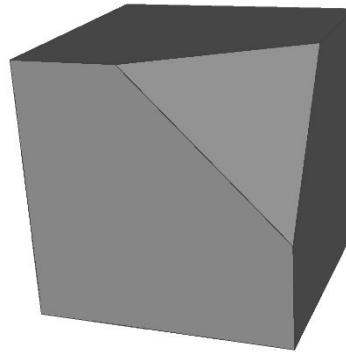
the Poincaré Formula states that:

$$V - E + F = 2 - 2g = \chi$$

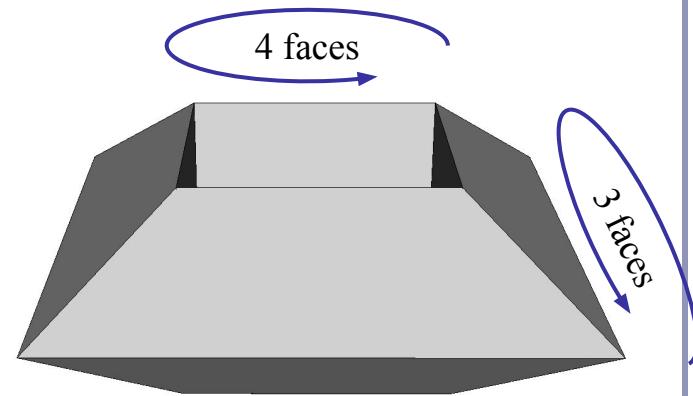
Genus, Poincaré and the Euler Characteristic



$$\begin{aligned}g &= 0 \\E &= 12 \\F &= 6 \\V &= 8 \\ \underline{V-E+F = 2-2g = 2}\end{aligned}$$



$$\begin{aligned}g &= 0 \\E &= 15 \\F &= 7 \\V &= 10 \\ \underline{V-E+F = 2-2g = 2}\end{aligned}$$



$$\begin{aligned}g &= 1 \\E &= 24 \\F &= 12 \\V &= 12 \\ \underline{V-E+F = 2-2g = 0}\end{aligned}$$

The Euler Characteristic and angle deficit

Descartes' *Theorem of Total Angle Deficit* states that on a surface S with Euler characteristic χ , the sum of the angle deficits of the vertices is $2\pi\chi$:

$$\sum_S AD(v) = 2\pi\chi$$

Cube:

- $\chi = 2 - 2g = 2$
- $AD(v) = \pi/2$
- $8(\pi/2) = 4\pi = 2\pi\chi$

Tetrahedron:

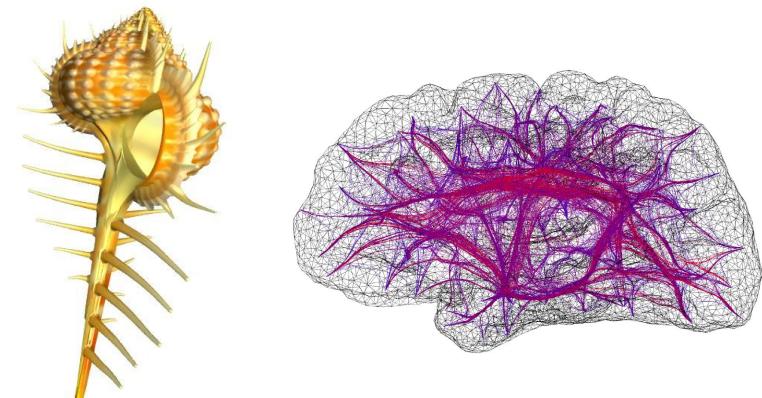
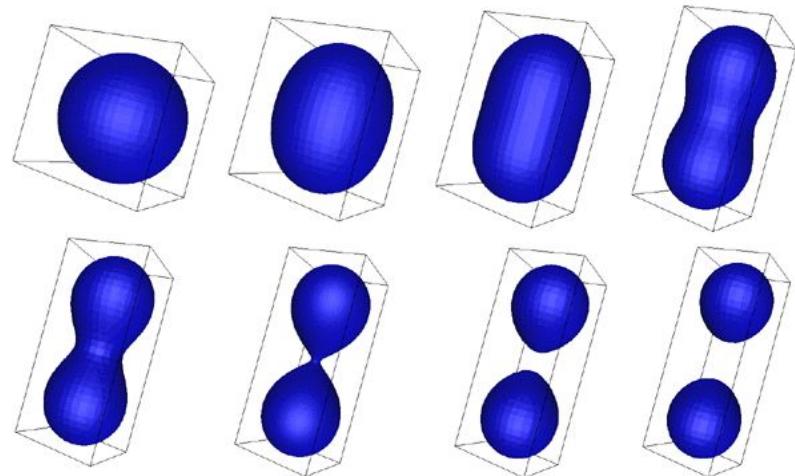
- $\chi = 2 - 2g = 2$
- $AD(v) = \pi$
- $4(\pi) = 4\pi = 2\pi\chi$

Implicit surfaces

Implicit surface modeling⁽¹⁾ is a way to produce very ‘organic’ or ‘bulbous’ surfaces very quickly without subdivision or NURBS.

Uses of implicit surface modelling:

- Organic forms and nonlinear shapes
- Scientific modeling (electron orbitals, gravity shells in space, some medical imaging)
- Muscles and joints with skin
- Rapid prototyping
- CAD/CAM solid geometry

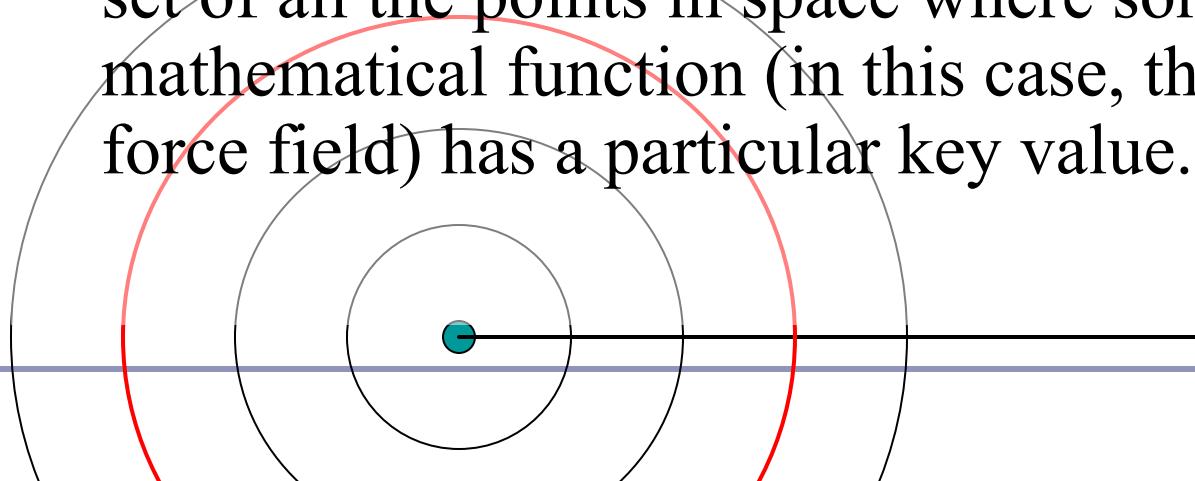


⁽¹⁾ AKA “metaball modeling”, “force functions”, “blobby modeling”...

How it works

The user controls a set of *control points*, like NURBS; each point in space generates a field of force, which drops off as a function of distance from the point (like gravity weakening with distance.)

This 3D field of forces defines an *implicit surface*: the set of all the points in space where some mathematical function (in this case, the value of the force field) has a particular key value.



Force =
2
1
0.5
0.25 ...

Force functions

A few popular force field functions:

- “Blobby Molecules” – Jim Blinn

$$F(r) = a e^{-br^2}$$

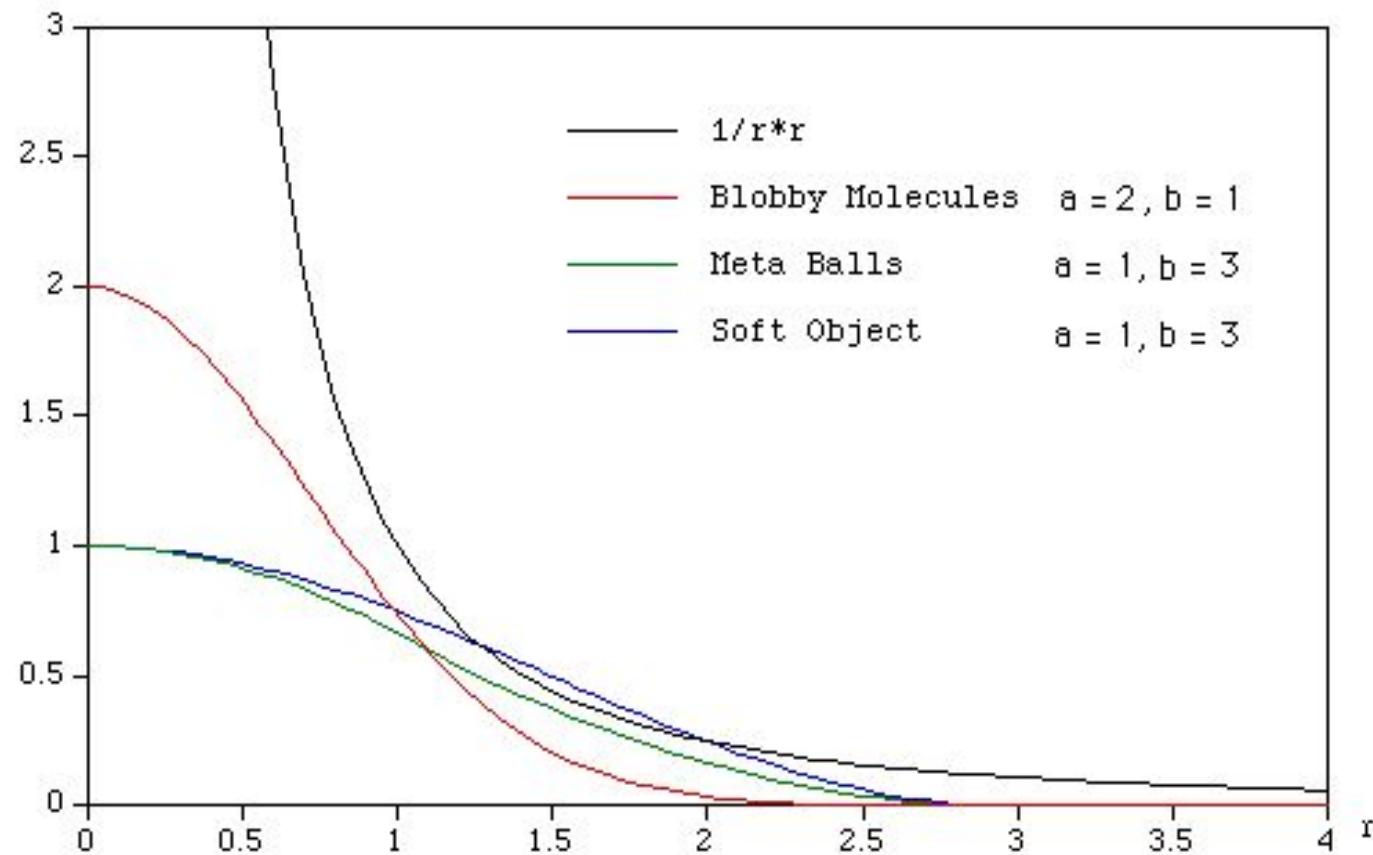
- “Metaballs” – Jim Blinn

$$F(r) = \begin{cases} a(1 - 3r^2 / b^2) & 0 \leq r < b/3 \\ (3a/2)(1-r/b)^2 & b/3 \leq r < b \\ 0 & b \leq r \end{cases}$$

- “Soft Objects” – Wyvill & Wyvill

$$F(r) = a(1 - 4r^6/9b^6 + 17r^4/9b^4 - 22r^2 / 9b^2)$$

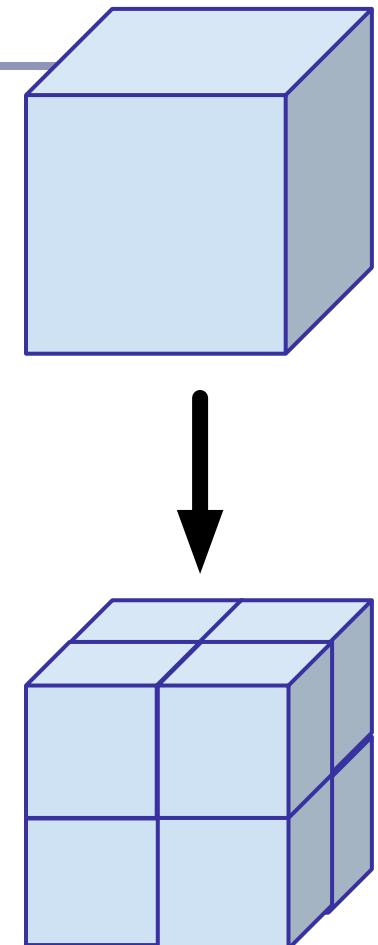
Comparison of force functions



Discovering the surface

An *octree* is a recursive subdivision of space which “homes in” on the surface, from larger to finer detail.

- An octree encloses a cubical volume in space. You evaluate the force function $F(v)$ at each vertex v of the cube.
- As the octree subdivides and splits into smaller octrees, only the octrees which contain some of the surface are processed; empty octrees are discarded.



Polygonizing the surface

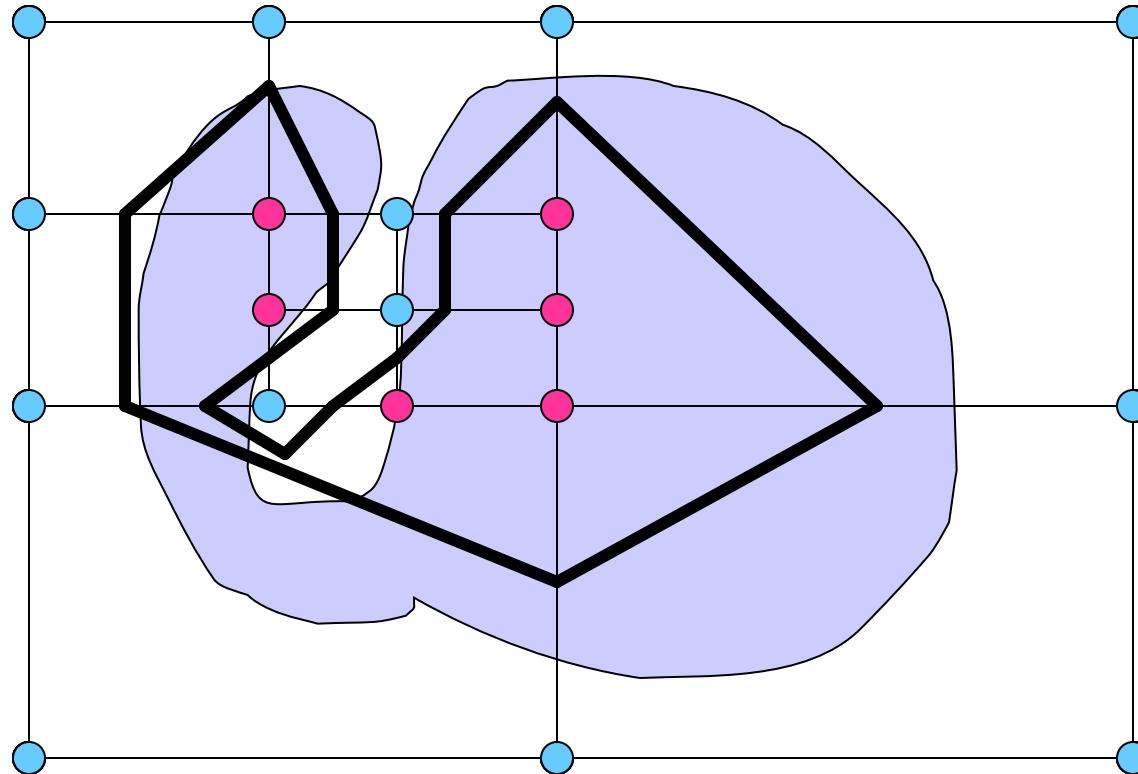
To display a set of octrees, convert the octrees into polygons.

- If some corners are “hot” (above the force limit) and others are “cold” (below the force limit) then the implicit surface crosses the cube edges in between.
- The set of midpoints of adjacent crossed edges forms one or more rings, which can be triangulated. The normal is known from the hot/cold direction on the edges.

To refine the polygonization, subdivide recursively; discard any child whose vertices are all hot or all cold.

Polygonizing the surface

Recursive subdivision (on a quadtree):

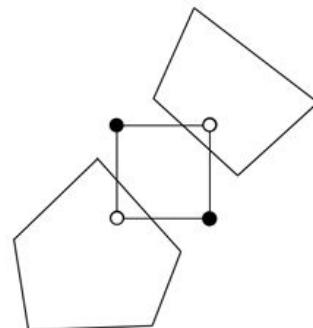


Polygonizing the surface

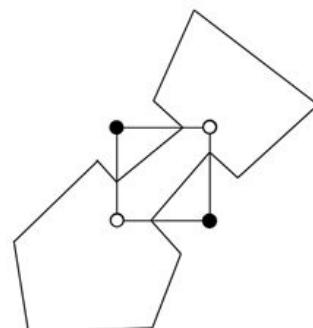
There are fifteen possible configurations (up to symmetry) of hot/cold vertices in the cube. →

- With rotations, that's 256 cases.

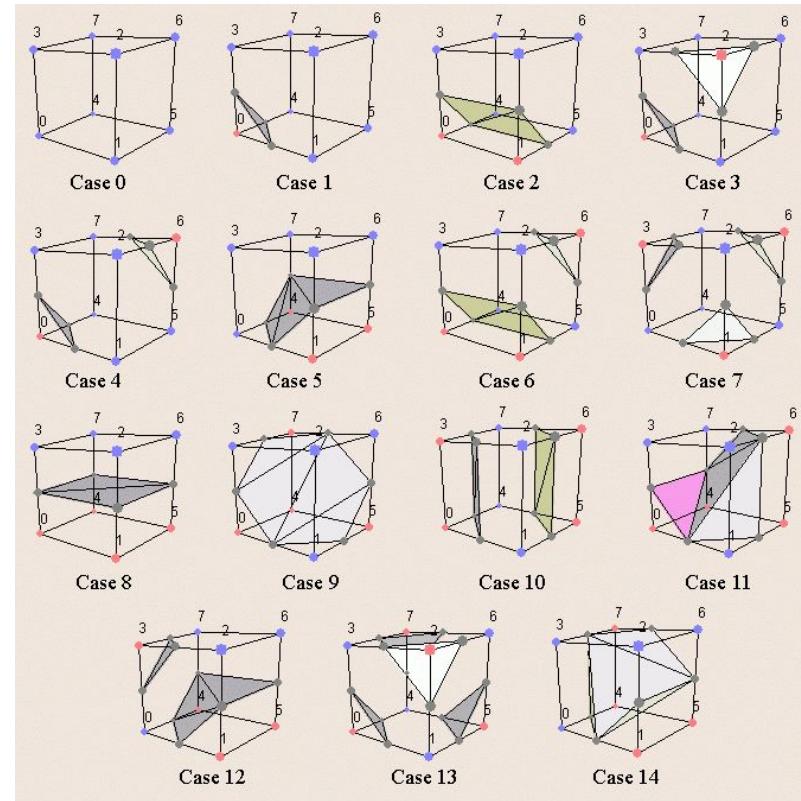
Beware: there are *ambiguous cases* in the polygonization which must be addressed separately. ↓



Break contour



Join contour



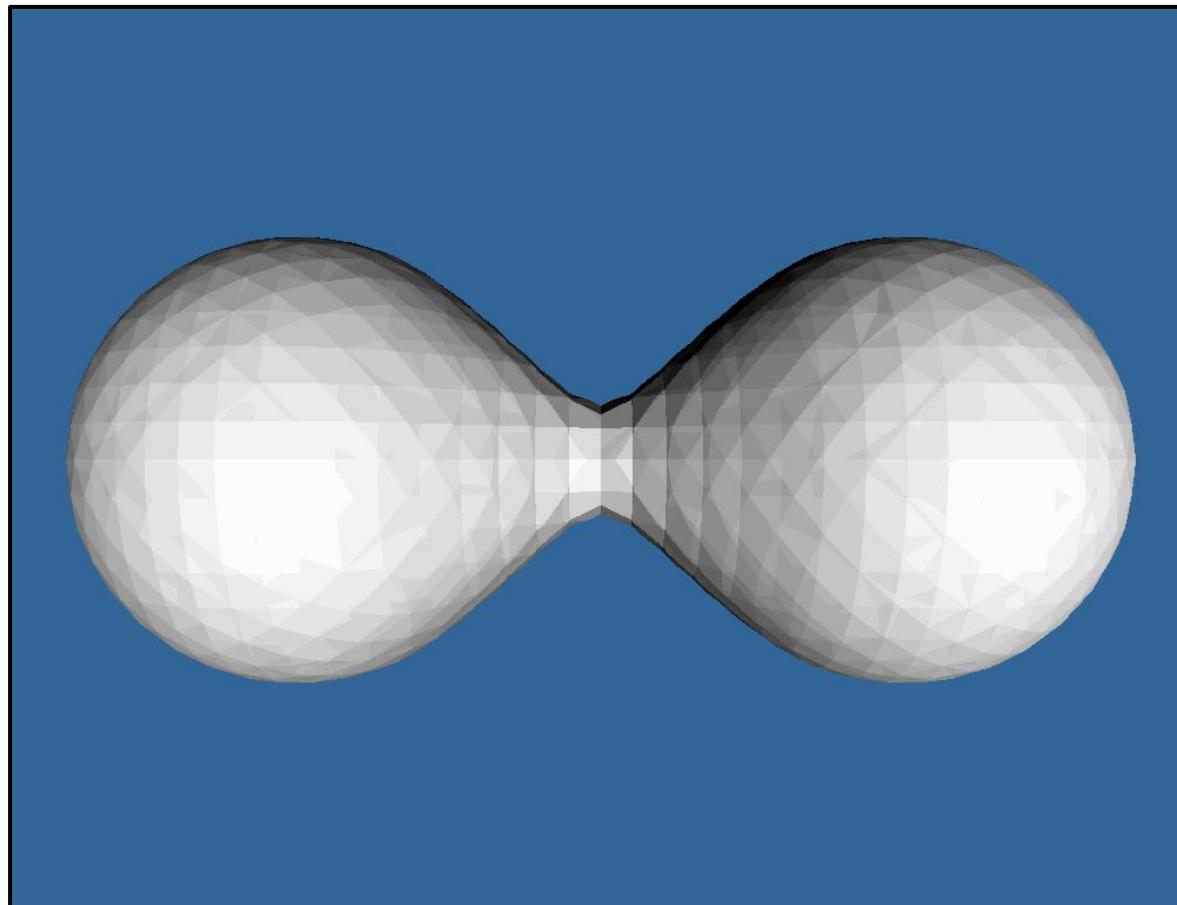
Images courtesy of [Diane Lingrand](#)

Smoothing the surface

Improved edge vertices

- The naïve implementation builds polygons whose vertices are the midpoints of the edges which lie between hot and cold vertices.
- The vertices of the implicit surface can be more closely approximated by points linearly interpolated along the edges of the cube by the weights of the relative values of the force function.
 - $t = (0.5 - F(P1)) / (F(P2) - F(P1))$
 - $P = P1 + t (P2 - P1)$

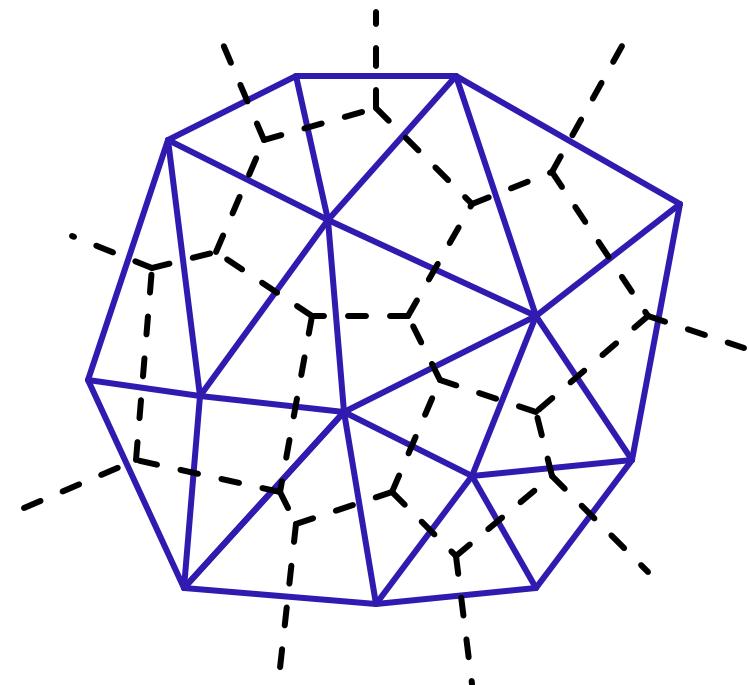
Implicit surfaces -- demo



Voronoi diagrams

The *Voronoi diagram*⁽²⁾ of a set of points P_i divides space into ‘cells’, where each cell C_i contains the points in space closer to P_i than any other P_j .

The *Delaunay triangulation* is the dual of the Voronoi diagram: a graph in which an edge connects every P_i which share a common edge in the Voronoi diagram.



A Voronoi diagram (dotted lines) and its dual Delaunay triangulation (solid).

⁽²⁾ AKA “Voronoi tesselation”, “Dirichelet domain”, “Thiessen polygons”, “plesiohedra”, “fundamental areas”, “domain of action”...

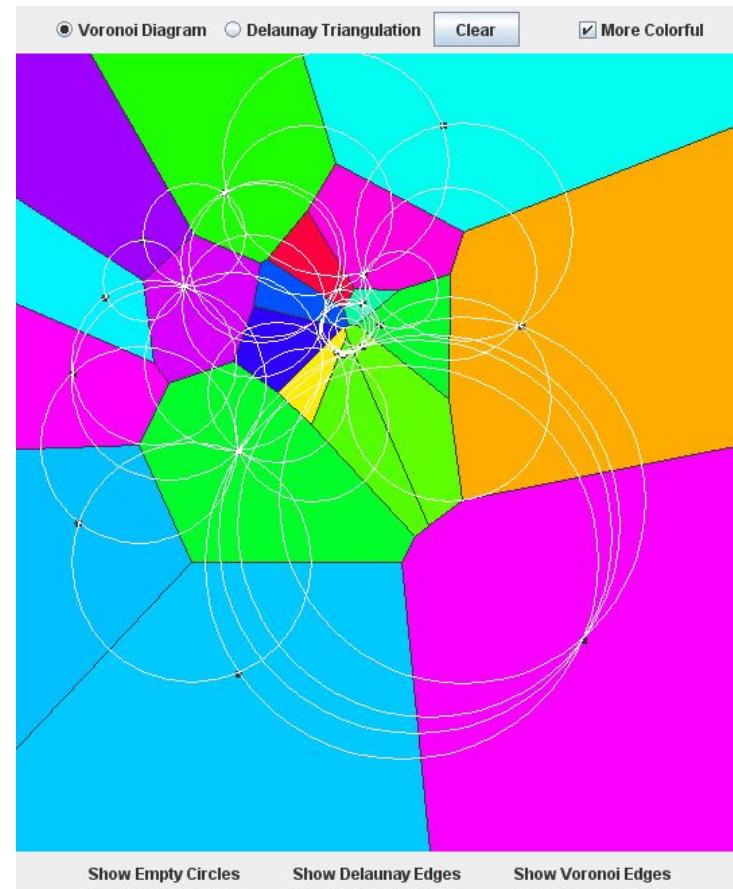
Voronoi diagrams

Given a set $S = \{p_1, p_2, \dots, p_n\}$, the formal definition of a Voronoi cell $C(S, p_i)$ is

$$C(S, p_i) = \{p \in R^d \mid |p - p_i| \leq |p - p_j|, i \neq j\}$$

The p_i are called the *generating points* of the diagram.

Where three or more boundary edges meet is a *Voronoi point*. Each Voronoi point is at the center of a circle (or sphere, or hypersphere...) which passes through the associated generating points and which is guaranteed to be empty of all other generating points.



Delaunay triangulation applet by Paul Chew ©1997–2007
<http://www.cs.cornell.edu/home/chew/Delaunay.html>

Delaunay triangulations and *equi-angularity*

The *equiangularity* of any triangulation of a set of points S is a sorted list of the angles $(\alpha_1 \dots \alpha_{3t})$ of the triangles.

- A triangulation is said to be *equiangular* if it possesses lexicographically largest equiangularity amongst all possible triangulations of S .
- The Delaunay triangulation is equiangular.

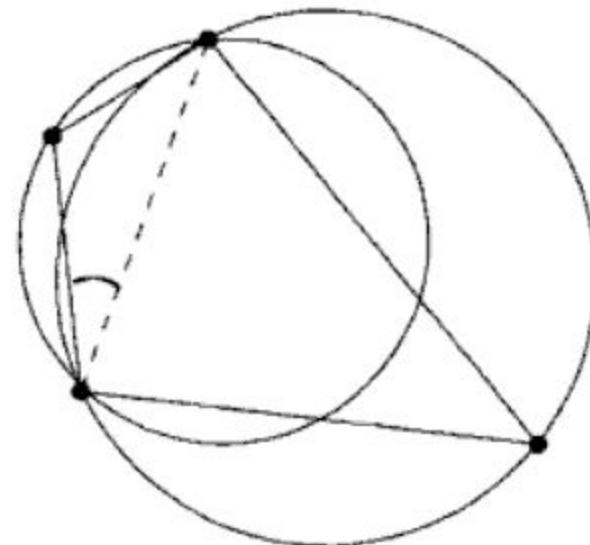


Image from *Handbook of Computational Geometry*
(2000) Jörg-Rüdiger Sack and Jorge Urrutia, p. 227

Delaunay triangulations and *empty circles*

Voronoi triangulations have the *empty circle* property: in any Voronoi triangulation of S , no point of S will lie inside the circle circumscribing any three points sharing a triangle in the Voronoi diagram.

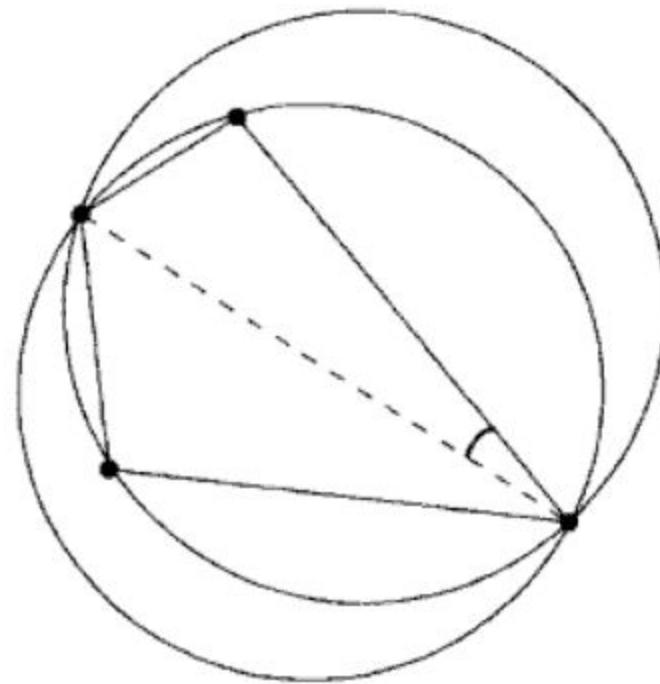


Image from *Handbook of Computational Geometry*
(2000) Jörg-Rüdiger Sack and Jorge Urrutia, p. 227

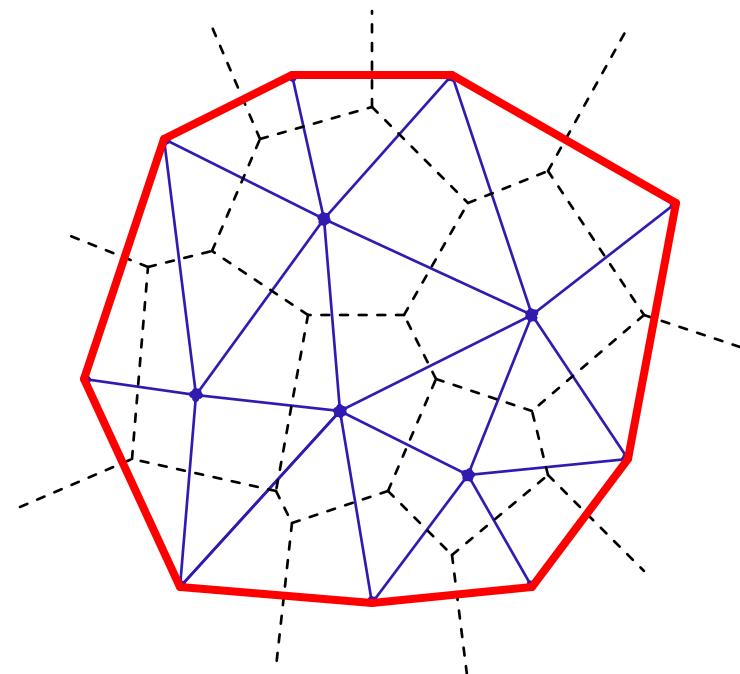
Delaunay triangulations and convex hulls

The border of the Delaunay triangulation of a set of points is always convex.

- This is true in 2D, 3D, 4D...

The Delaunay triangulation of a set of points in R^n is the planar projection of a convex hull in R^{n+1} .

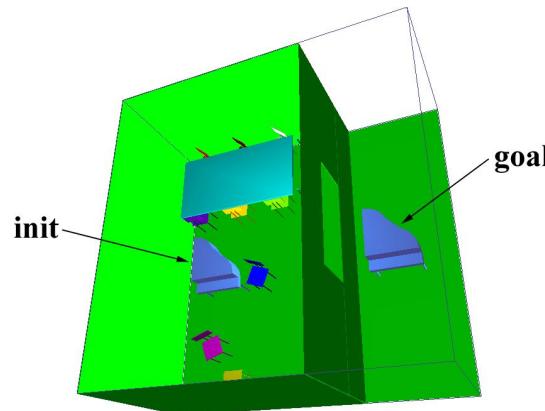
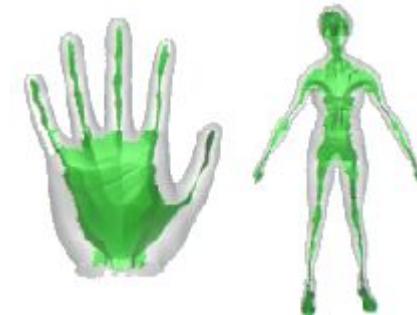
- Ex: from 2D ($P_i = \{x, y\}_i$), loft the points upwards, onto a parabola in 3D ($P'_i = \{x, y, x^2 + y^2\}_i$). The resulting polyhedral mesh will still be convex in 3D.



Voronoi diagrams and the *medial axis*

The *medial axis* of a surface is the set of all points within the surface equidistant to the two or more nearest points on the surface.

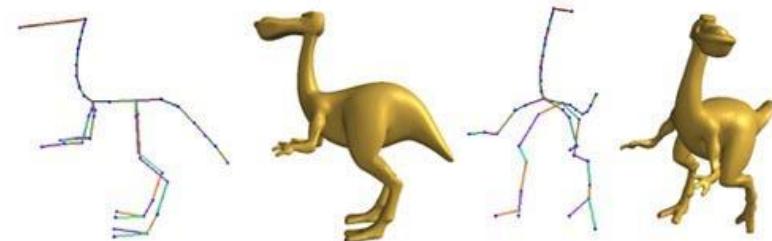
- This can be used to extract a skeleton of the surface, for (for example) path-planning solutions, surface deformation, and animation.



[A Voronoi-Based Hybrid Motion Planner for Rigid Bodies](#)
M Foskey, M Garber, M Lin, DManocha

[Approximating the Medial Axis from the Voronoi Diagram with a Convergence Guarantee](#)

Tamal K. Dey, Wulue Zhao



[Shape Deformation using a Skeleton to Drive Simplex Transformations](#)

IEEE Transaction on Visualization and Computer Graphics, Vol. 14, No. 3, May/June 2008, Page 693-706

Han-Bing Yan, Shi-Min Hu, Ralph R Martin, and Yong-Liang Yang

Fortune's algorithm

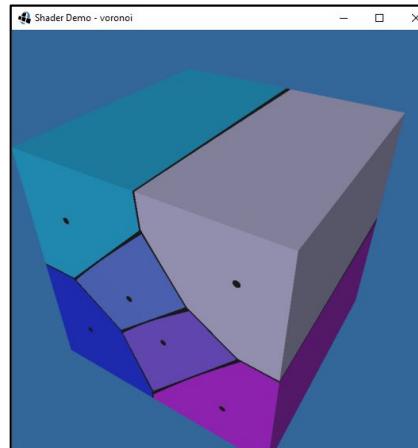
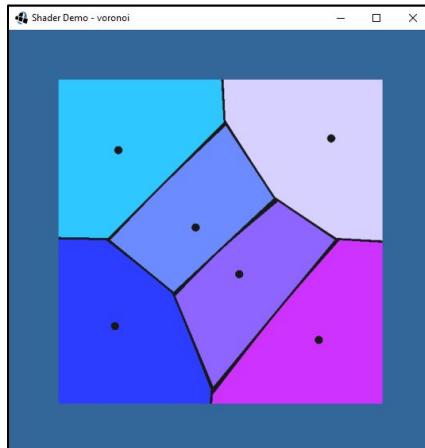
1. The algorithm maintains a sweep line and a “beach line”, a set of parabolas advancing left-to-right from each point. The beach line is the union of these parabolas.
 - a. The intersection of each pair of parabolas is an edge of the voronoi diagram
 - b. All data to the left of the beach line is “known”; nothing to the right can change it
 - c. The beach line is stored in a binary tree
2. Maintain a queue of two classes of event: the addition of, or removal of, a parabola
3. There are $O(n)$ such events, so Fortune's algorithm is $O(n \log n)$



GPU-accelerated Voronoi Diagrams

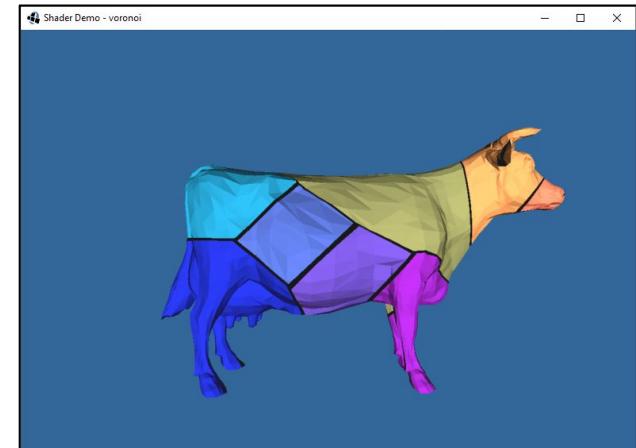
Brute force:

- For each pixel to be rendered on the GPU, search all points for the nearest point

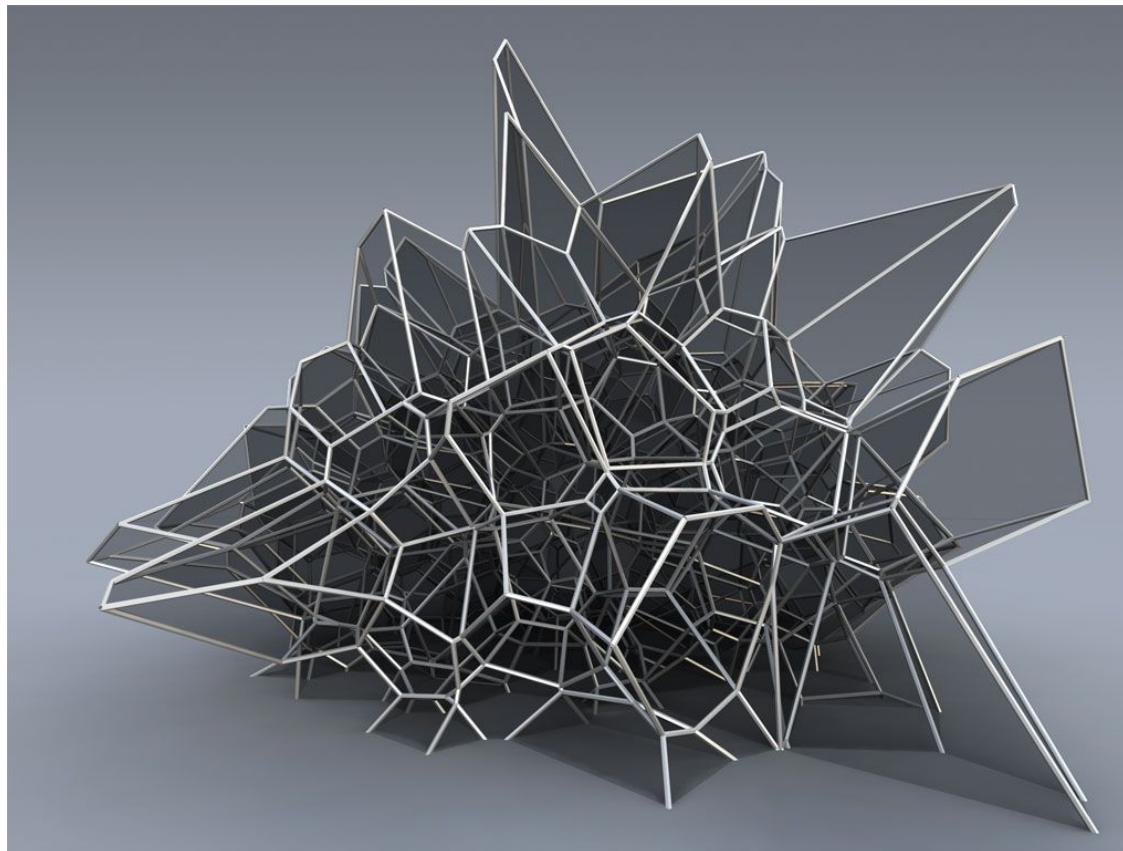


Elegant (and 2D only):

- Render each point as a discrete 3D cone in isometric projection, let z-buffering sort it out



Voronoi cells in 3D



Silvan Oesterle, Michael Knauss

References

Implicit modelling:

- D. Ricci, *A Constructive Geometry for Computer Graphics*, Computer Journal, May 1973
J Bloomenthal, *Polygonization of Implicit Surfaces*, Computer Aided Geometric Design, Issue 5, 1988
B Wyvill, C McPheevers, G Wyvill, *Soft Objects*, Advanced Computer Graphics (Proc. CG Tokyo 1986)
B Wyvill, C McPheevers, G Wyvill, *Animating Soft Objects*, The Visual Computer, Issue 4 1986

<http://astronomy.swin.edu.au/~pbourke/modelling/implicitsurf/>

<http://www.cs.berkeley.edu/~job/Papers/turk-2002-MIS.pdf>

<http://www.unchainedgeometry.com/jbloom/papers/interactive.pdf>

<http://www-courses.cs.uiuc.edu/~cs319/polygonization.pdf>

Voxels:

- J. Wilhelms and A. Van Gelder, *A Coherent Projection Approach for Direct Volume Rendering*, Computer Graphics, 35(4):275-284, July 1991.

Voronoi diagrams

- M. de Berg, O. Cheong, M. van Kreveld, M. Overmars, "Computational Geometry: Algorithms and Applications", Springer-Verlag,
<http://www.cs.uu.nl/geobook/>
<http://www.ics.uci.edu/~eppstein/junkyard/nn.html>
<http://www.iquilezles.org/www/articles/voronoi/voronoilines.htm> // Voronoi on GPU

Gaussian Curvature

- http://en.wikipedia.org/wiki/Gaussian_curvature
<http://mathworld.wolfram.com/GaussianCurvature.html>

The Poincaré Formula

- <http://mathworld.wolfram.com/PoincareFormula.html>

Advanced Graphics



Global Illumination

Alex Benton, University of Cambridge – A.Benton@damtp.cam.ac.uk

What's wrong with raytracing?

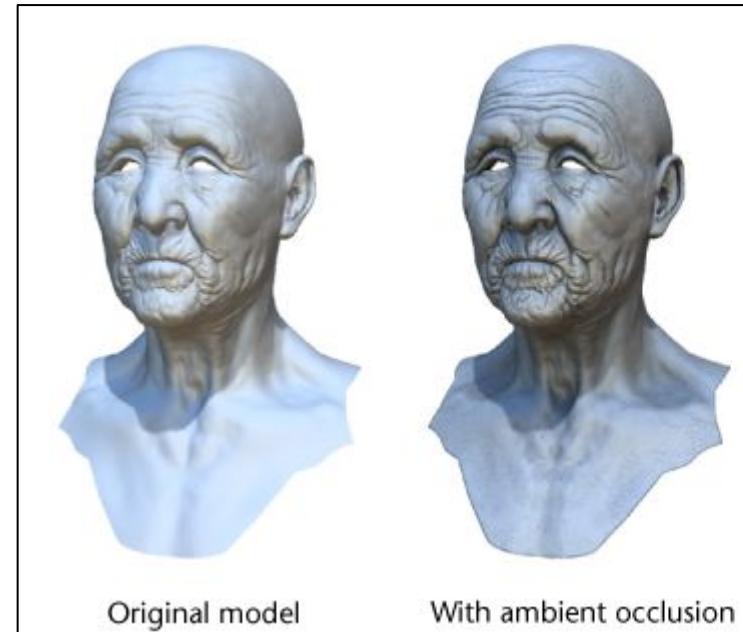
- Soft shadows are expensive
- Shadows of transparent objects require further coding or hacks
- Lighting off reflective objects follows different shadow rules from normal lighting
- Hard to implement diffuse reflection (color bleeding, such as in the Cornell Box—notice how the sides of the inner cubes are shaded red and green.)
- Fundamentally, the ambient term is a hack and the diffuse term is only one step in what should be a recursive, self-reinforcing series.



The *Cornell Box* is a test for rendering Software, developed at Cornell University in 1984 by Don Greenberg. An actual box is built and photographed; an identical scene is then rendered in software and the two images are compared.

Ambient occlusion

- *Ambient illumination* is a blanket constant that we often add to every illuminated element in a scene, to (inaccurately) model the way that light scatters off all surfaces, illuminating areas not in direct lighting.
- *Ambient occlusion* is the technique of adding/removing ambient light when other objects are nearby and scattered light wouldn't reach the surface.
- Computing ambient occlusion is a form of *global illumination*, in which we compute the lighting of scene elements in the context of the scene as a whole.



Ambient occlusion in action



Ambient occlusion in action



Car photos from John Hable's presentation at GDC 2010,
"Uncharted 2: HDR Lighting" (filmicgames.com/archives/6)

Ambient occlusion in action



Car photos from John Hable's presentation at GDC 2010,
"Uncharted 2: HDR Lighting" (filmicgames.com/archives/6)

Ambient occlusion in action



Car photos from John Hable's presentation at GDC 2010,
"Uncharted 2: HDR Lighting" (filmicgames.com/archives/6)

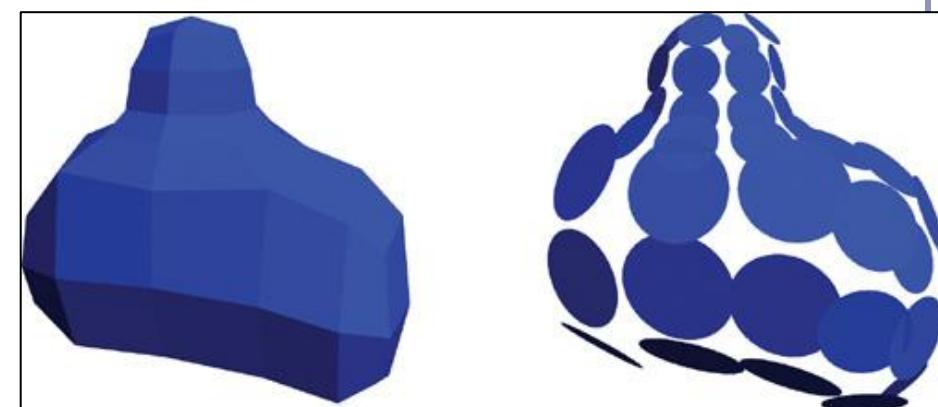
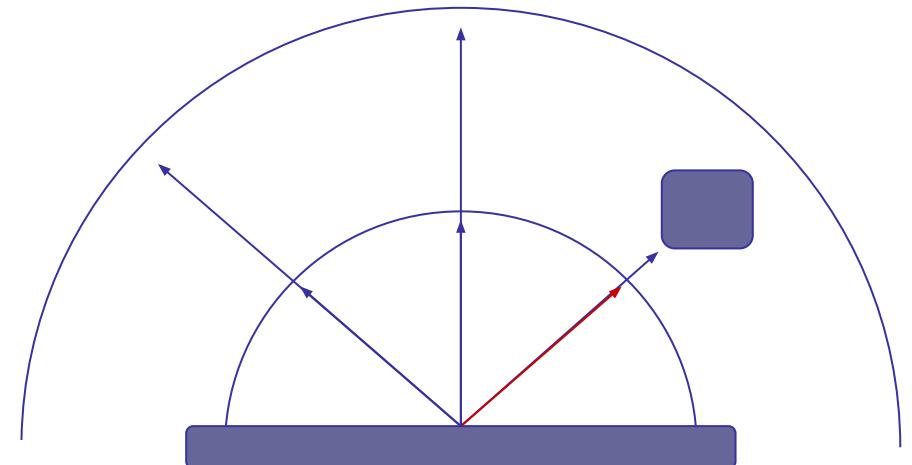
Ambient occlusion - Theory

We can treat the background (the sky) as a vast ambient illumination source.

- For each vertex of a surface, compute how much background illumination reaches the vertex by computing how much sky it can ‘see’
- Integrate occlusion A_p over the hemisphere around the normal at the vertex:

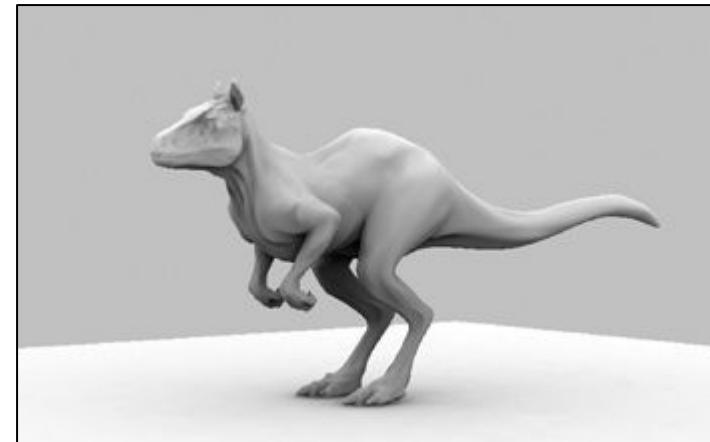
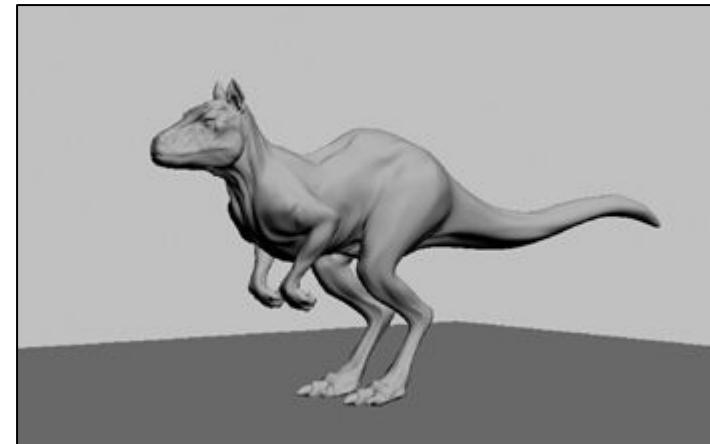
$$A_{\bar{p}} = \frac{1}{\pi} \int_{\Omega} V_{\bar{p}, \hat{\omega}} (\hat{n} \cdot \hat{\omega}) d\omega$$

- A_p occlusion at point p
- n normal at point p
- $V_{p, \omega}$ visibility from p in direction ω
- Ω integrate over area (hemisphere)



Ambient occlusion - Theory

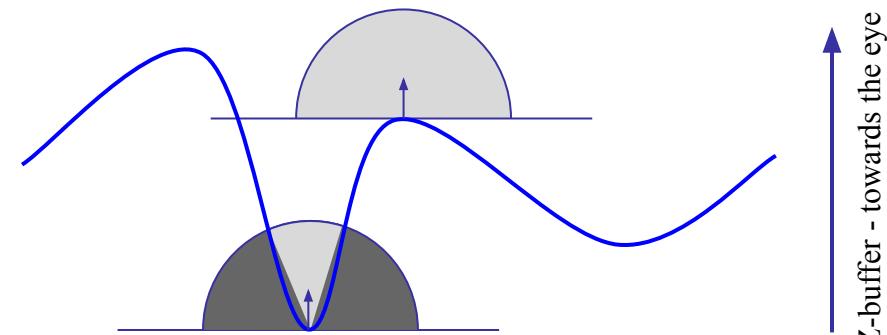
- This approach is very flexible
- Also very expensive!
- To speed up computation, randomly sample rays cast out from each polygon or vertex (this is a *Monte-Carlo* method)
- Alternatively, render the scene from the point of view of each vertex and count the background pixels in the render
- Best used to pre-compute per-object “*occlusion maps*”, texture maps of shadow to overlay onto each object
- But pre-computed maps fare poorly on animated models...



Screen Space Ambient Occlusion (“SSAO”)

“True ambient occlusion is hard,
let’s go hacking.”

- Approximate ambient occlusion by comparing z-buffer values in screen space!
- Open plane = unoccluded
- Closed ‘valley’ in depth buffer = shadowed by nearby geometry
- Multi-pass algorithm
- Runs entirely on the GPU



Screen Space Ambient Occlusion

1. For each visible point on a surface in the scene (ie., each pixel), take multiple samples (typically between 8 and 32) from nearby and map these samples back to screen space
2. Check if the depth sampled at each neighbor is nearer to, or further from, the scene sample point
3. If the neighbor is nearer than the scene sample point then there is some degree of occlusion
 - a. Care must be taken not to occlude if the nearer neighbor is too much nearer than the scene sample point; this implies a separate object, much closer to the camera
4. Sum retained occlusions, weighting with an occlusion function

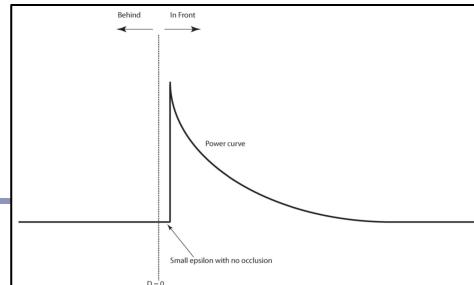
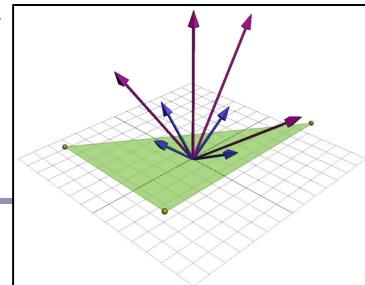


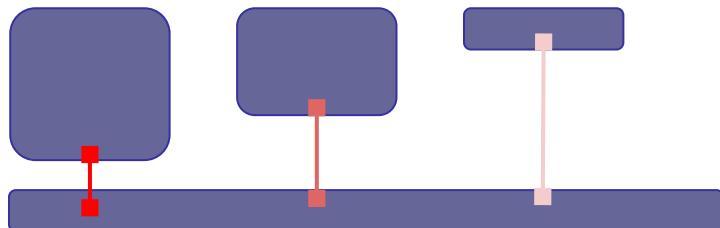
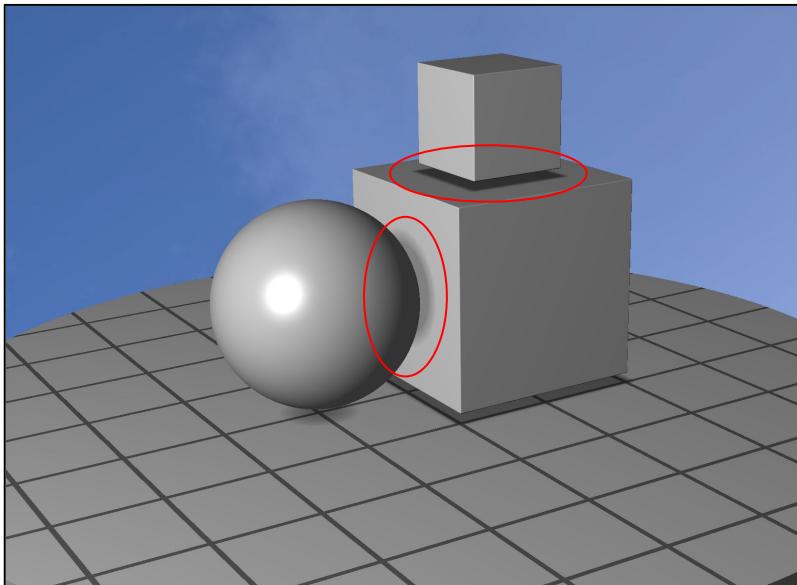
Image: StarCraft II. Advances in Real-Time Rendering in 3D Graphics and Games - Course notes, SIGGRAPH 2008

SSAO example- Uncharted 2



4) Low Pass Filter (significant blurring)

Ambient occlusion and Signed Distance Fields



In a nutshell, SSAO tries to estimate occlusion by asking, “how far is it to the nearest neighboring geometry?”

With signed distance fields, this question is almost trivial to answer.

```
float ambient(vec3 pt, vec3 normal) {  
    return abs(getSdf(pt + 0.1 * normal)) / 0.1;  
}
```

```
float ambient(vec3 pt, vec3 normal) {  
    float a = 1;  
    int step = 0;  
  
    for (float t = 0.01; t <= 0.1; ) {  
        float d = abs(getSdf(pt + t * normal));  
        a = min(a, d / t);  
        t += max(d, 0.01);  
    }  
    return a;  
}
```

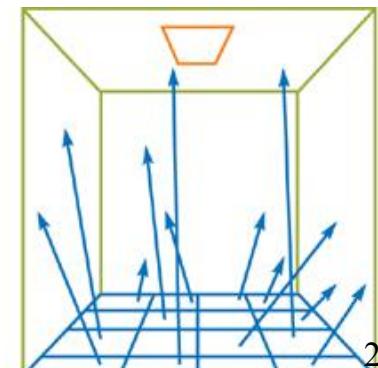
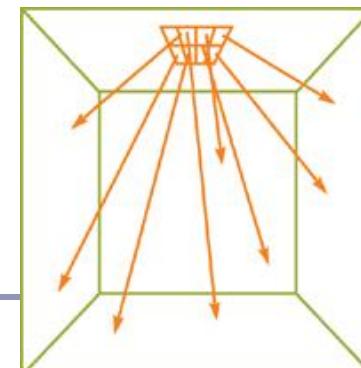
Radiosity

- *Radiosity* is an illumination method which simulates the global dispersion and reflection of diffuse light.
 - First developed for describing spectral heat transfer (1950s)
 - Adapted to graphics in the 1980s at Cornell University
- Radiosity is a finite-element approach to global illumination: it breaks the scene into many small elements ('*patches*') and calculates the energy transfer between them.



Radiosity—algorithm

- Surfaces in the scene are divided into *patches*, small subsections of each polygon or object.
- For every pair of patches A, B, compute a *view factor* (also called a *form factor*) describing how much energy from patch A reaches patch B.
 - The further apart two patches are in space or orientation, the less light they shed on each other, giving lower view factors.
- Calculate the lighting of all directly-lit patches.
- Bounce the light from all lit patches to all those they light, carrying more light to patches with higher relative view factors. Repeating this step will distribute the total light across the scene, producing a global diffuse illumination model.



Radiosity—mathematical support

The ‘radiosity’ of a single patch is the amount of energy leaving the patch per discrete time interval.

This energy is the total light being emitted directly from the patch combined with the total light being reflected by the patch:

$$B_i = E_i + R_i \sum_{j=1}^n B_j F_{ij}$$

This forms a system of linear equations, where...

B_i is the radiosity of patch i ;

B_j is the radiosity of each of the other patches ($j \neq i$)

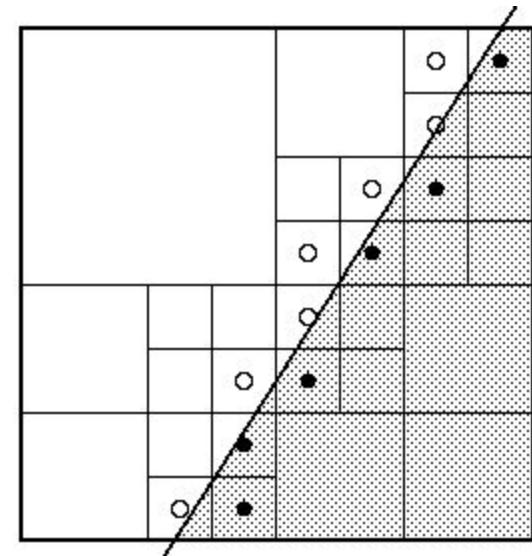
E_i is the emitted energy of the patch

R_i is the reflectivity of the patch

F_{ij} is the view factor of energy from patch i to patch j .

Radiosity—form factors

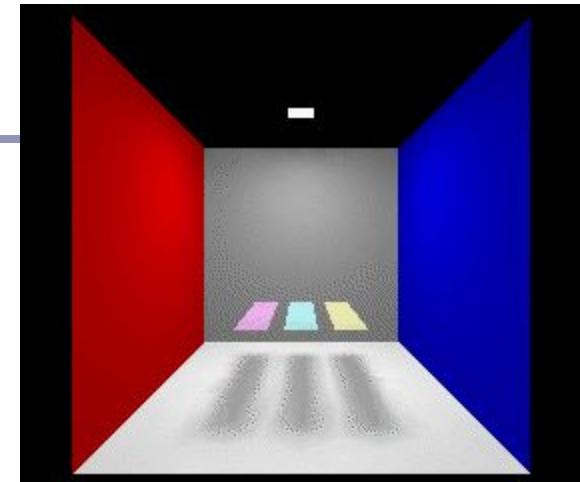
- Finding form factors can be done procedurally or dynamically
 - Can subdivide every surface into small patches of similar size
 - Can dynamically subdivide wherever the 1st derivative of calculated intensity rises above some threshold.
- Computing cost for a general radiosity solution goes up as the square of the number of patches, so try to keep patches down.
 - Subdividing a large flat white wall could be a waste.
- Patches should ideally closely align with lines of shadow.



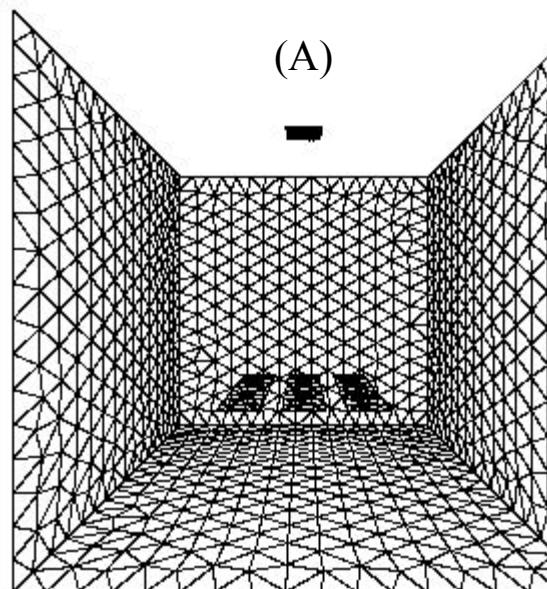
Radiosity—implementation

(A) Simple patch triangulation

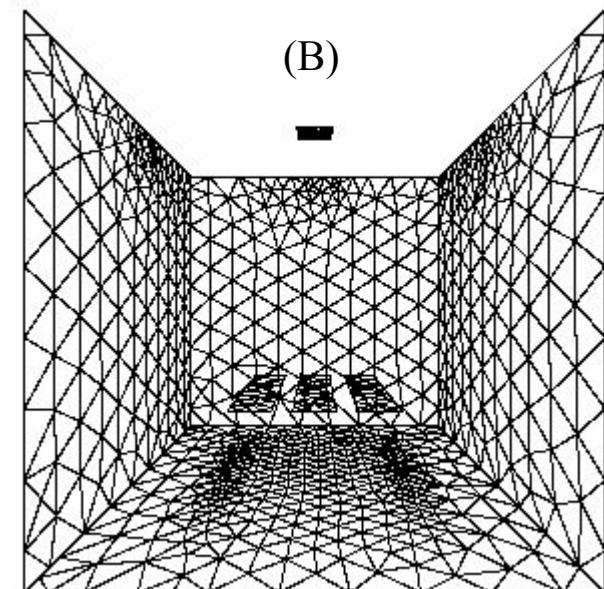
(B) Adaptive patch generation: the floor and walls of the room are dynamically subdivided to produce more patches where shadow detail is higher.



Images from “Automatic generation of node spacing function”, IBM (1998)
<http://www.trl.ibm.com/projects/meshing/nsp/nspE.htm>



(A)



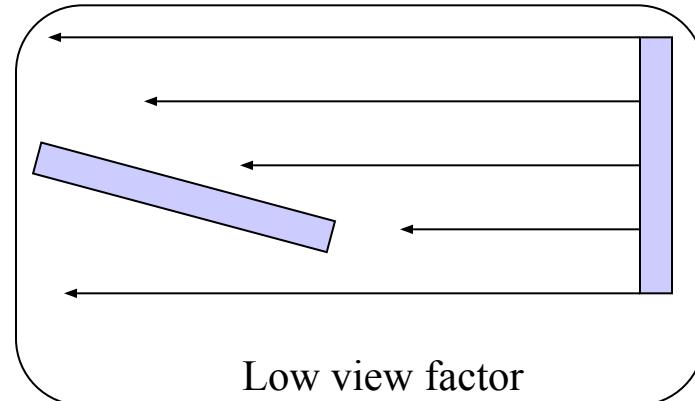
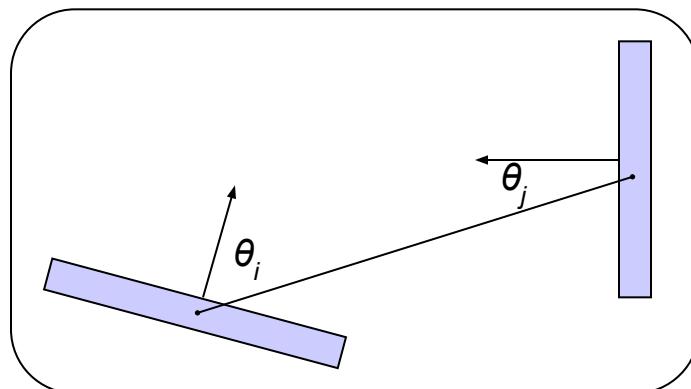
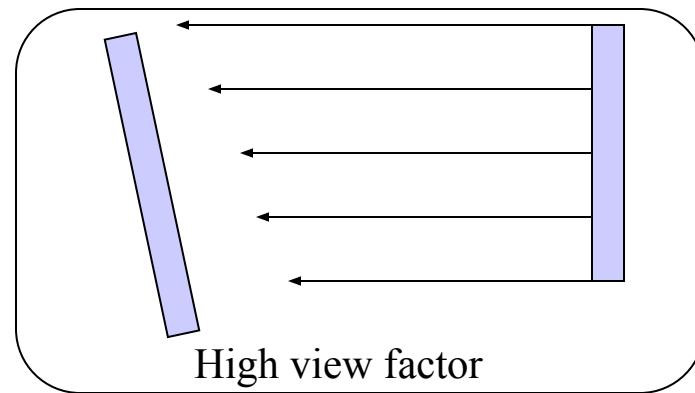
(B)

Radiosity—view factors

One equation for the view factor between patches i, j is:

$$Fi \rightarrow j = \frac{\cos \theta_i \cos \theta_j}{\pi r^2} V(i, j)$$

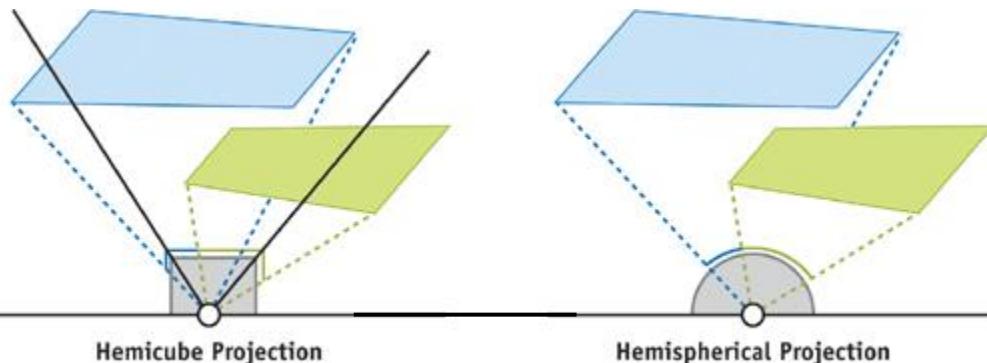
...where θ is the angle between the normal of patch i and the line to patch j , r is the distance and $V(i, j)$ is the visibility from i to j (0 for occluded, 1 for clear line of sight.)



Low view factor

Radiosity—calculating visibility

- Calculating $V(i,j)$ can be slow.
- One method is the *hemicube*, in which each form factor is encased in a half-cube. The scene is then ‘rendered’ from the point of view of the patch, through the walls of the hemicube; $V(i,j)$ is computed for each patch based on which patches it can see (and at what percentage) in its hemicube.
- A purer method, but more computationally expensive, uses hemispheres.



Note: This method can be accelerated using modern graphics hardware to render the scene. The scene is ‘rendered’ with flat lighting, setting the ‘color’ of each object to be a pointer to the object in memory.

Radiosity gallery



Image from *A Two Pass Solution to the Rendering Equation: a Synthesis of Ray Tracing and Radiosity Methods*,
John R. Wallace, Michael F. Cohen and Donald P. Greenberg
(Cornell University, 1987)

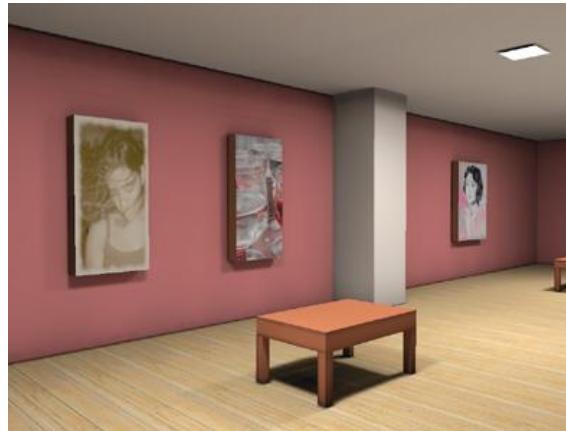


Image from
GPU Gems II, nVidia



Teapot (wikipedia)

Shadows, refraction and caustics

- Problem: shadow ray strikes transparent, refractive object.
 - Refracted shadow ray will now miss the light.
 - This destroys the validity of the boolean shadow test.
- Problem: light passing through a refractive object will sometimes form *caustics* (right), artifacts where the envelope of a collection of rays falling on the surface is bright enough to be visible.



This is a photo of a real pepper-shaker. Note the caustics to the left of the shaker, in and outside of its shadow.

Photo credit: Jan Zankowski

Shadows, refraction and caustics

- Solutions for shadows of transparent objects:
 - Backwards ray tracing (Arvo)
 - *Very* computationally heavy
 - Improved by stencil mapping (Shenya et al)
 - Shadow attenuation (Pierce)
 - Low refraction, no caustics
- More general solution:
 - *Photon mapping* (Jensen)→



Image from <http://graphics.ucsd.edu/~henrik/>
Generated with photon mapping

Photon mapping

Photon mapping is the process of emitting photons into a scene and tracing their paths probabilistically to build a *photon map*, a data structure which describes the illumination of the scene independently of its geometry.

This data is then combined with ray tracing to compute the global illumination of the scene.

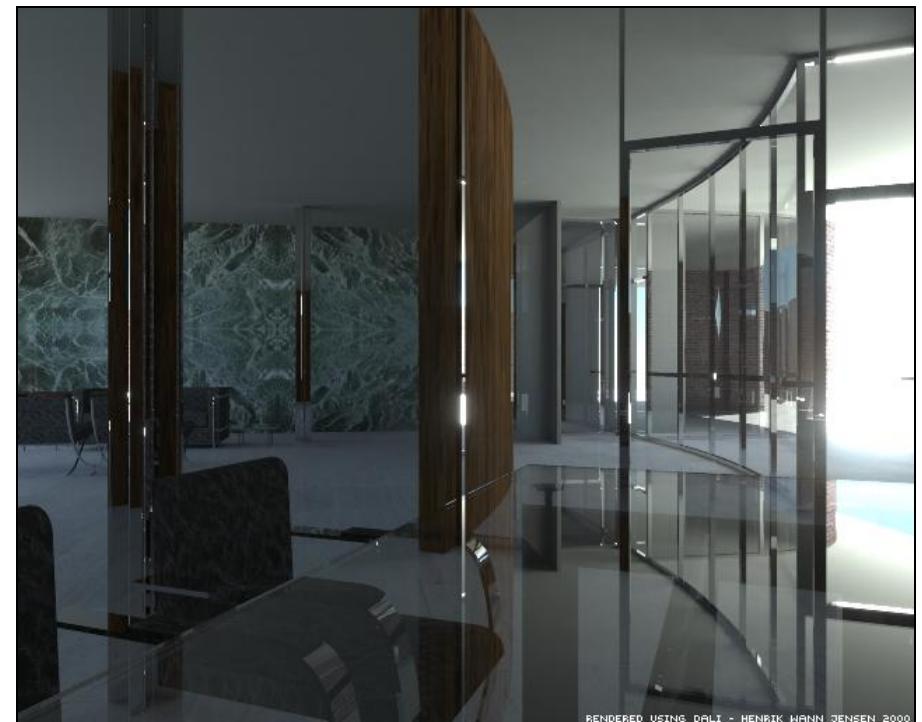


Image by Henrik Jensen (2000)

Photon mapping—algorithm (1/2)

Photon mapping is a two-pass algorithm:

1. Photon scattering

- A. Photons are fired from each light source, scattered in randomly-chosen directions. The number of photons per light is a function of its surface area and brightness.
- B. Photons fire through the scene (re-use that raytracer, folks.) Where they strike a surface they are either absorbed, reflected or refracted.
- C. Wherever energy is absorbed, cache the location, direction and energy of the photon in the *photon map*. The photon map data structure must support fast insertion and fast nearest-neighbor lookup; a *kd-tree*¹ is often used.

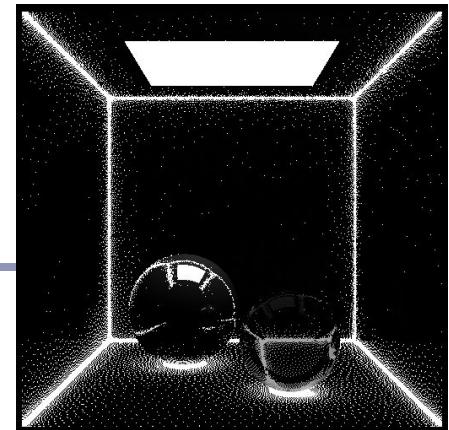


Image by Zack Waters

Photon mapping—algorithm (2/2)

Photon mapping is a two-pass algorithm:

2. Rendering

- A. Ray trace the scene from the point of view of the camera.
- B. For each first contact point P use the ray tracer for specular but compute diffuse from the photon map and do away with ambient completely.
- C. Compute radiant illumination by summing the contribution along the eye ray of all photons within a sphere of radius r of P .
- D. Caustics can be calculated directly here from the photon map. For speed, the caustic map is usually distinct from the radiance map.

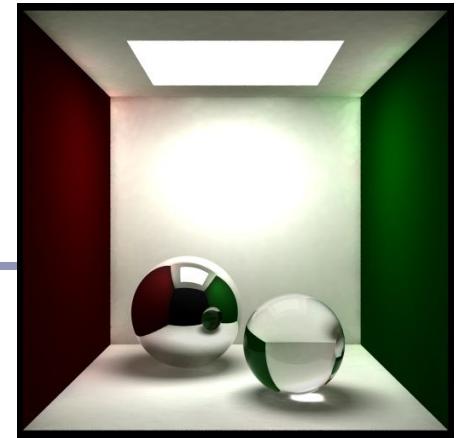
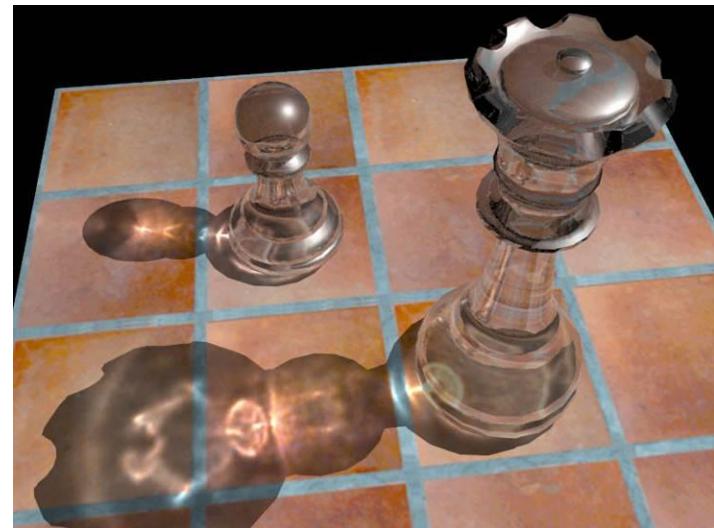


Image by Zack Waters

Photon mapping is probabilistic

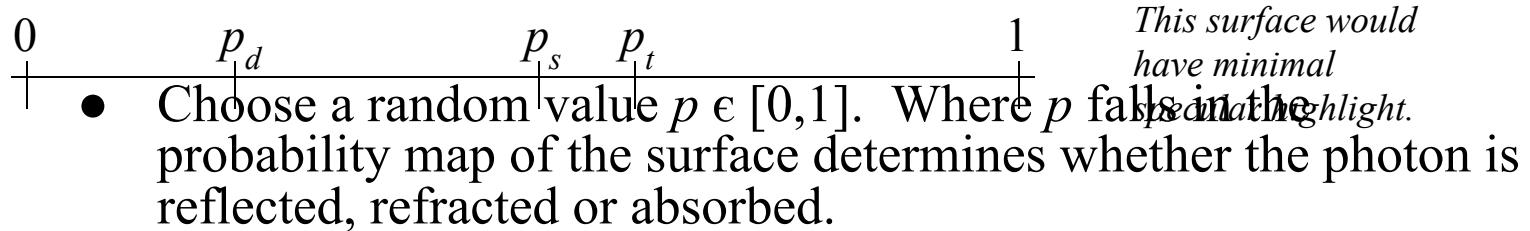
This method is a great example of *Monte Carlo integration*, in which a difficult integral (the lighting equation) is simulated by randomly sampling values from within the integral's domain until enough samples average out to about the right answer.

- This means that you're going to be firing *millions* of photons. Your data structure is going to have to be very space-efficient.

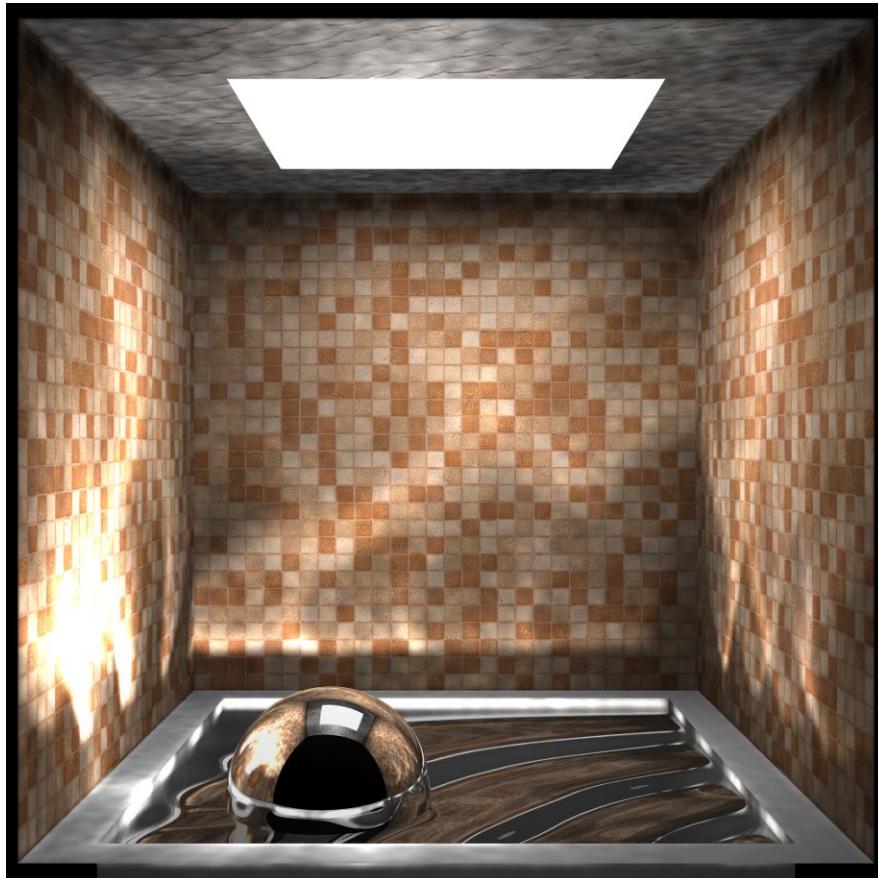


Photon mapping is probabilistic

- Initial photon direction is random. Constrained by light shape, but random.
- What exactly happens each time a photon hits a solid also has a random component:
 - Based on the diffuse reflectance, specular reflectance and transparency of the surface, compute probabilities p_d , p_s and p_t where $(p_d + p_s + p_t) \leq 1$. This gives a probability map:



Photon mapping gallery



http://web.cs.wpi.edu/~emmanuel/courses/cs563/write_ups/zackw/photon_mapping/PhotonMapping.html



<http://graphics.ucsd.edu/~henrik/images/global.html>



<http://www.pbrt.org/gallery.php>

References

Shirley and Marschner, “Fundamentals of Computer Graphics”, Chapter 24 (2009)

Ambient occlusion and SSAO

- “GPU Gems 2”, nVidia, 2005. Vertices mapped to illumination.
http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter14.html
- MITTRING, M. 2007. Finding Next Gen – CryEngine 2.0, Chapter 8, SIGGRAPH 2007 Course 28 – Advanced Real-Time Rendering in 3D Graphics and Games, Siggraph 2007, San Diego, CA, August 2007.
http://developer.amd.com/wordpress/media/2012/10/Chapter8-Mittring-Finding_NextGen_CryEngine2.pdf
- John Hable’s presentation at GDC 2010, “Uncharted 2: HDR Lighting” (filmicgames.com/archives/6)

Radiosity

- nVidia: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter39.html
- Cornell: <http://www.graphics.cornell.edu/online/research/>
- Wallace, J. R., K. A. Elmquist, and E. A. Haines. 1989, “A Ray Tracing Algorithm for Progressive Radiosity.” In *Computer Graphics (Proceedings of SIGGRAPH 89)* 23(4), pp. 315–324.
- Buss, “3-D Computer Graphics: A Mathematical Introduction with OpenGL” (Chapter XI), Cambridge University Press (2003)

Photon mapping

- Henrik Jensen, “Global Illumination using Photon Maps”: <http://graphics.ucsd.edu/~henrik/>
- Henrik Jensen, “Realistic Image Synthesis Using Photon Mapping”
- Zack Waters, “Photon Mapping”:
http://web.cs.wpi.edu/~emmanuel/courses/cs563/write_ups/zackw/photon_mapping/PhotonMapping.html