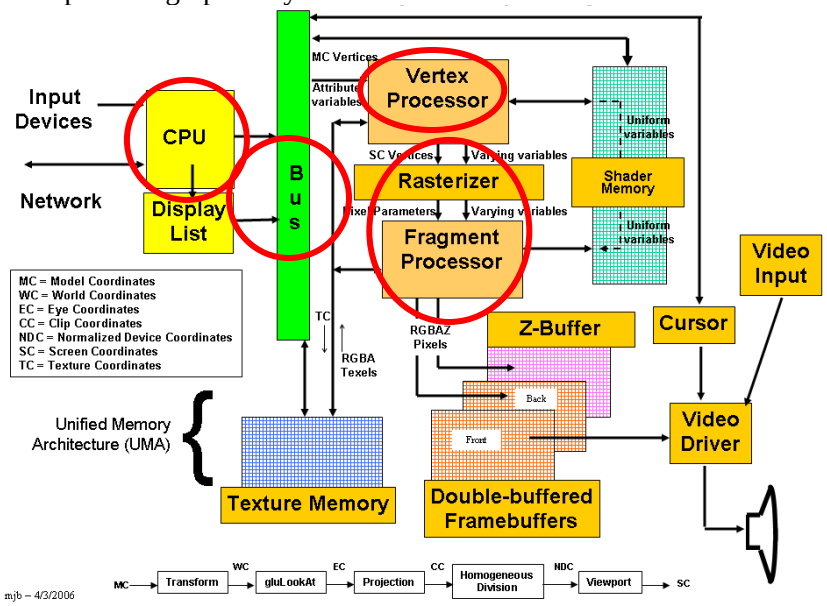
Performance OpenGL Programming
(for whatever reason)

Mike Bailey
Oregon State University

Performance Bottlenecks

In general there are four places a graphics system can become bottlenecked:



1. The computer – if the CPU cannot compute the data, read the data, uncompress the data, or call the graphics routines fast enough, then it doesn't matter how fast your graphics card is.

2. The bus – a slow bus will choke down transmission of graphics from the CPU to the graphics card. AGP is dead. Long live PCI Express! :-)

Type of Board	Speed to the Board	Speed from the Board
PCI	132 Mb/sec	132 Mb/sec
AGP 8X	2 Gb/sec	264 Mb/sec
PCI Express	4 Gb/sec	4 Gb/sec

3. The vertex processing – a graphics scene will bottleneck here if it has lots of small primitives. This is often how CAD-type applications are characterized.

4. The rasterizer and fragment processing (i.e., per-pixel operations) – a graphics scene will bottleneck here if it has a small number of large primitives. Games and flight simulators generally work this way.

Part of the trick is to achieve a balance of the load between these four elements.

General Performance Principles

`glFinish()` blocks until the graphics card is finished with what has been sent to it so far. Use it to generate benchmark timing information. Take it out for production. *Never, ever, use it in production code.*

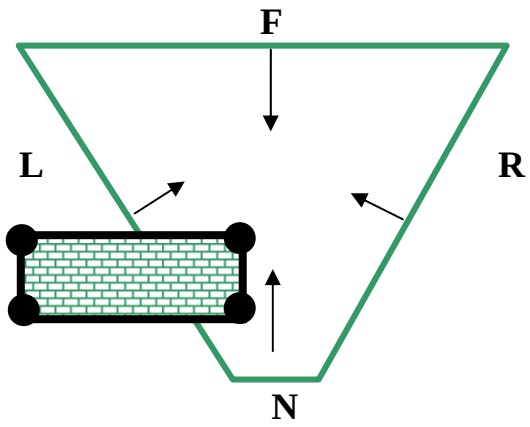
Understand the nature of your scene against the four places a graphics scene can become bottlenecked.

Benchmark if you are not sure.

Vertex Processing Principles

Quickly pre-eliminate as much scenery as possible. (There is a dividing line here. If it takes too much computation time to eliminate a section of the scene, it might be better to display it and let the graphics hardware do it.)

Cull based on comparing bounding volumes with the viewing frustum -- try to achieve *quick trivial rejections*



The bounding volume is out-of-bounds if all 8 points are out-of-bounds *for the same reason*
The Cohen-Sutherland algorithm:

Clip Code

L	R	B	T	N	F
32	16	8	4	2	1

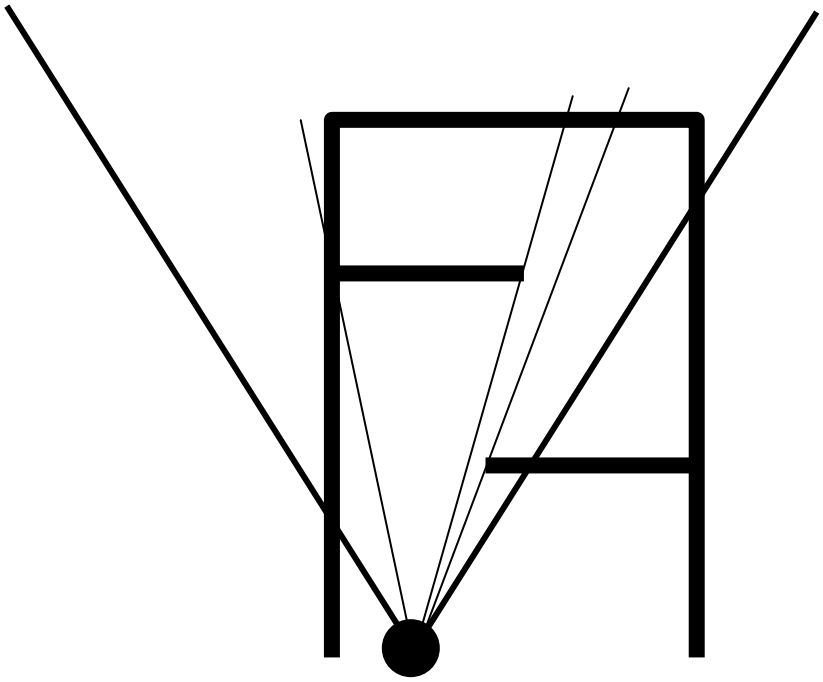
If you ‘inclusive-or’ (|) all 8 clip codes together and get **0**, then the bounding box is completely inside the viewing volume

If you ‘and’ (&) all 8 clip codes together and get **! 0**, then the bounding box is completely outside the viewing volume.

Sometimes a bounding sphere or bounding cylinder makes more sense. It depends on what bounding geometry most tightly fits your scene.

Break big things into smaller things so that more can be cleanly culled – hierarchical culling.

Remove part of the scene while making the player think it is part of the game :-)



Minimize all state changes. Group similar-attribute primitives (e.g., color) together. Note: this sometimes flies in the face of object-oriented programming.

State: current setting of all OpenGL attributes, such as color, lighting, texture, transformation, etc.

Pipeline: graphics hardware is a multi-step process. It is good for performance if you can keep all steps busy.

Changing state usually causes the pipeline to completely clear out before any new geometry can enter, which kills performance.

Group similar primitives in the *same* glBegin() – glEnd().

Disable lighting, or simulate it in a texture

Use directional lights, not positional

Use quad strips instead of quads – transform less points

Use triangle strips instead of triangles, and instead of quad anythings. (The vendors say this, but I am not sure it is true…)

Better: use **vertex arrays** – only transform each vertex once

Best: use **vertex objects** – vertex data gets stored on the graphics card.

Maximize the size of vertex array/object blocks
- Small batches of geometry can kill performance
- 100 triangles/batch should be a minimum
- >= 500 would be better
- Some game development companies say they try to use a size of 10,000 vertices or more

Use multiple levels of detail, depending on how much of the screen a set of primitives occupies. Use *occlusion testing* to determine if this is necessary.

Pre-transform objects which are undergoing constant ("static") transformations. E.g., your vector cloud. (I.e., don't use a pile of **glPushMatrix( ) - glRotatef( ) – glCallList( ) – glPopMatrix( )** calls.)

Pre-compute as much geometry as you can
- Use display lists (some drivers implement them in host memory, some in graphics card memory)

Let one frame finish drawing while setting up to draw the next one, i.e., don't use **glFinish( )**.

Level of Detail

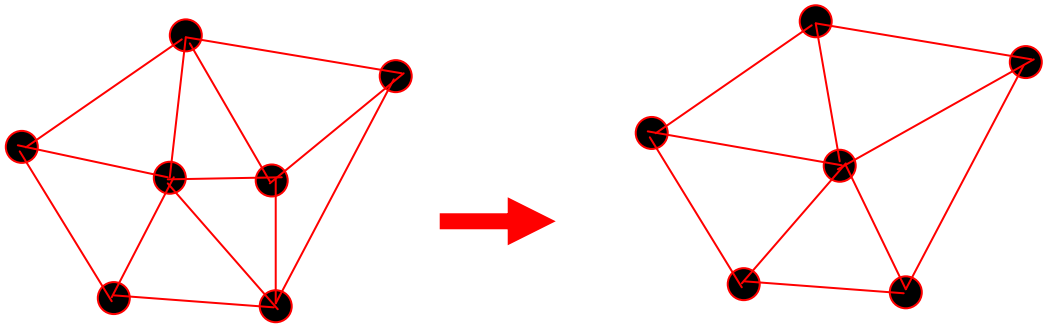
How far away is the object from the viewer? If it's far away, you don't need to use as much geometric detail to display it. Occlusion querying can help. (Occlusion query lets you pretend-render a bounding box, and the graphics processor will tell you how many pixels it would have occupied.)

Don't use up your "polygon budget" if you don't need to

Keep several representations for different components of the scene and switch between them.

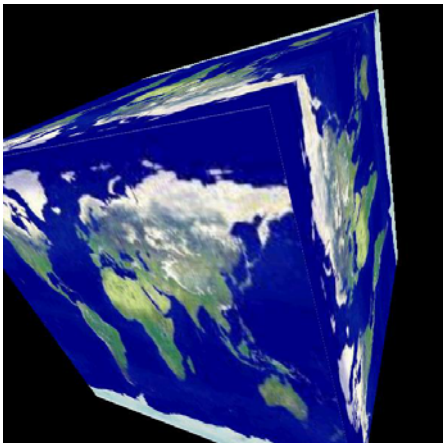
Example #1: Different resolutions of GLUT spheres

Example #2: Polygon mesh decimation



Level-of-detailing can also be done as a function of the speed with which an object is moving in the scene.

Creative Texturing



Reduces polygon count without apparent loss of detail

Textures can be used in Level of Detail analysis

Make "plywood sets" instead of 3D scene detail
E.g., a distant forest or mountains

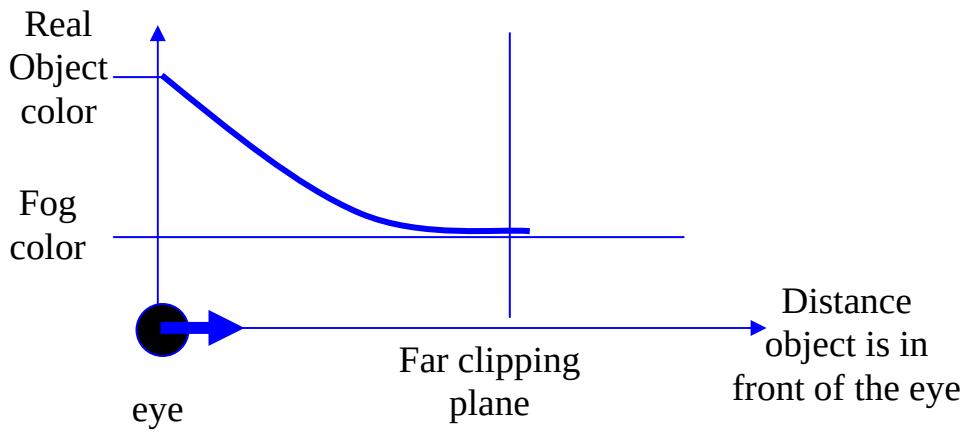
Can bring in full 3D detail as you get closer

Build lighting into the texture image, and then display it as a GL_REPLACE texture

Render 3D scenes into an image and use it as a texture
This is referred to as "render-to-texture".
E.g., creating the distant forest or mountains

Fog Tricks

Use fog to cover up a close-in far clipping plane – fog and haze can hide the fact that you are far-clipping away a lot of scene detail. And you thought it was just part of the game… **:-)**



Also, an object in the haze can be displayed with less scene detail

Another way to get away with a close-in far clipping plane: bring distant detail in with alpha blending (transparency).

Fragment Processing Principles

Quickly pre-eliminate as much scenery as possible. (There is a dividing line here. If it takes too much computation to eliminate a section of the scene, it might be better to display it and let the graphics hardware do it.)

Use unsigned bytes for pixel formats (not floating point, even though you can do more with floating point pixel formats).

Cull backfaces

Use **Texture Objects** (load all textures into the graphics card memory, then query to see how they fit)

Assign intelligent priorities to textures

Use alpha testing to reject transparent pixels

Depth sort primitives front-to-back so maximum number of pixels will get depth-rejected
Use texture objects.

Benchmarking

Run timing tests by using `glFinish()` in `Display()`to completely clear out the pipeline.

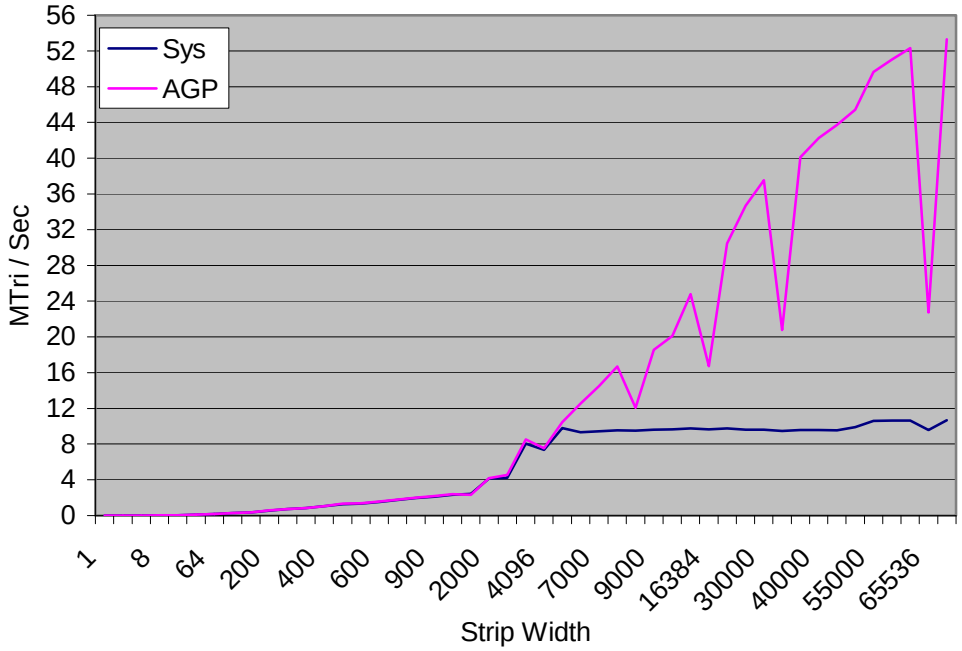
```
glFinish();
int t0 = glutGet( GLUT_ELAPSED_TIME );

<< All Graphics Calls except Swapping the Double Buffers>>

glFinish();
int t1 = glutGet( GLUT_ELAPSED_TIME );

glutSwapBuffers();
fprintf( stderr, "One display frame took %d milliseconds\n", t1 - t0 );
```

Definitely remove this for production runs !!

Graphics cards today are really fast. Make your test size *really big* so that the precision of the system clock is not an issue.

Be careful about just putting a **for( )** loop around a set of code. Some compilers will optimize that down to just one loop. If the time seems to be independent of the number of loops, fool the compiler by translating the scene based on the loop index.

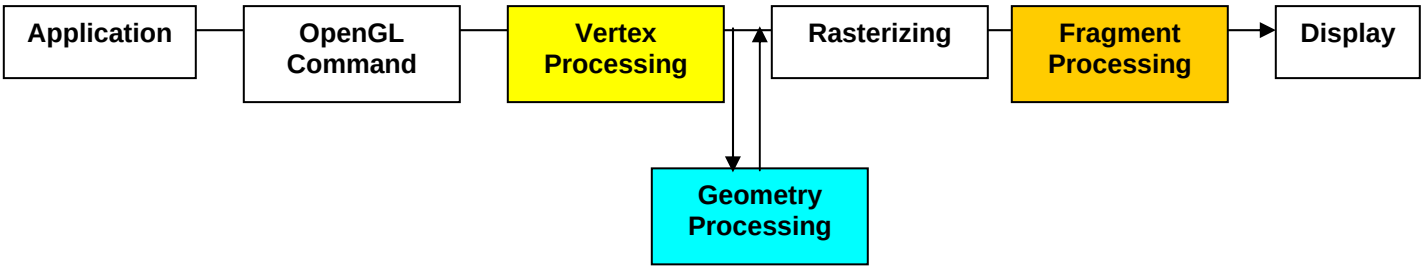
Benchmarking is worthwhile to do. You sometimes get exactly what you thought you'd get. But, sometimes you get wildly unexpected results, such as this



Credit: Daniel New

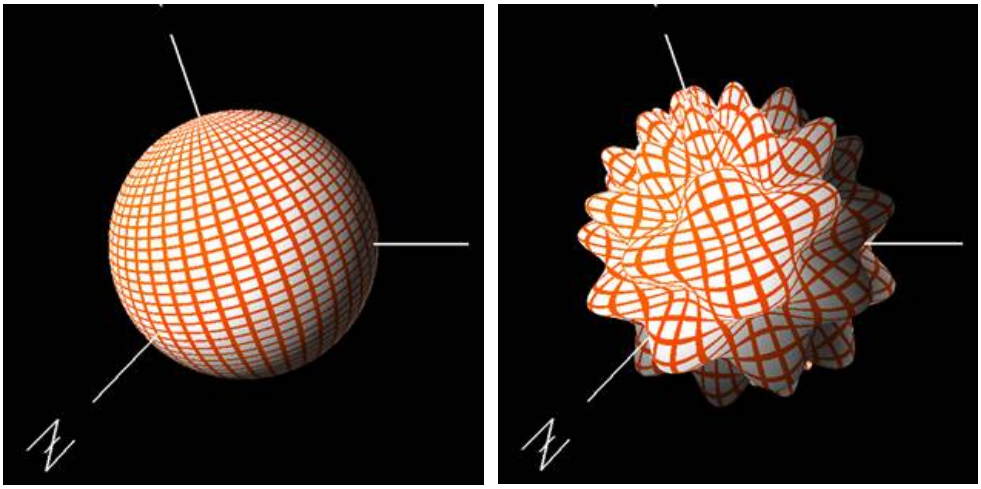
OpenGL Shaders

Be aware that you can now place your own code in the per-vertex and per-fragment parts of the graphics hardware. This can allow you to move operations to the graphics card that used to need to be on the CPU.

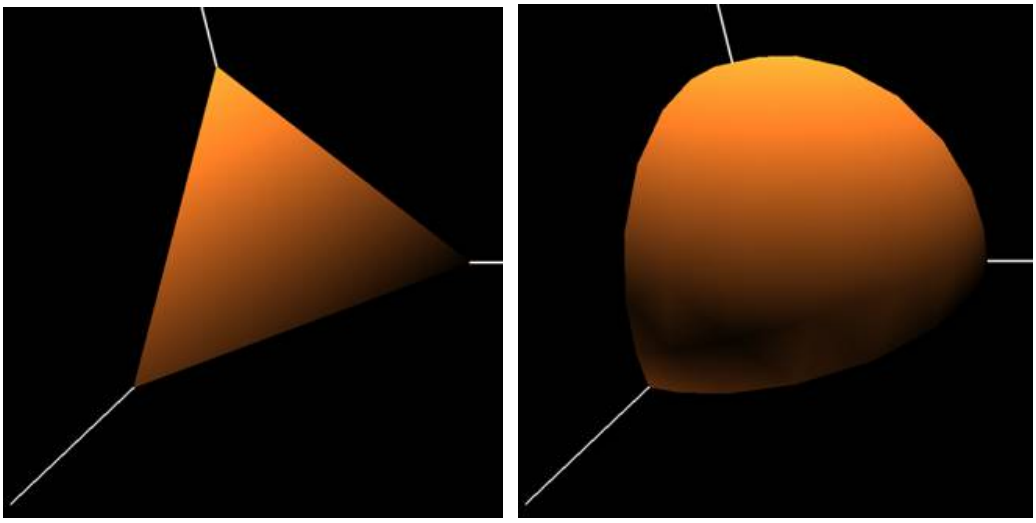


This can be used, for example, to:

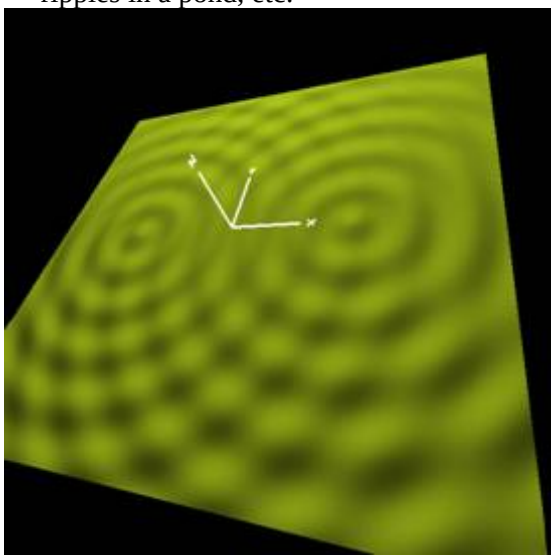
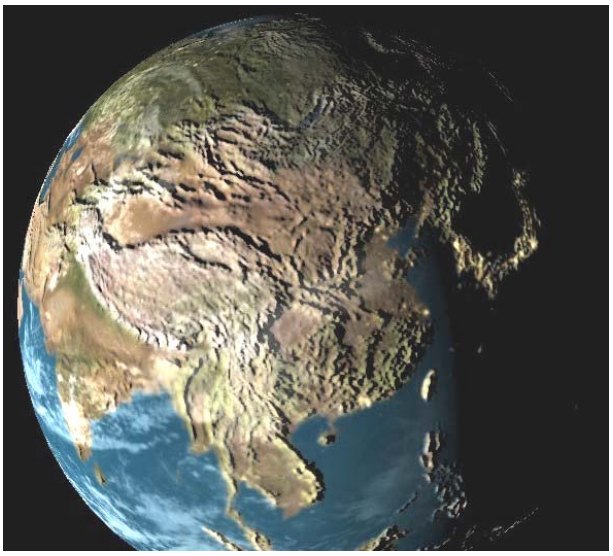
- Create geometry from an equation. That is, start with a mesh of points and displace each one according to a displacement equation.



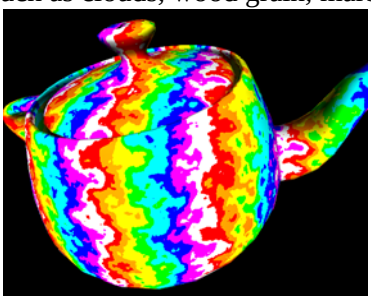
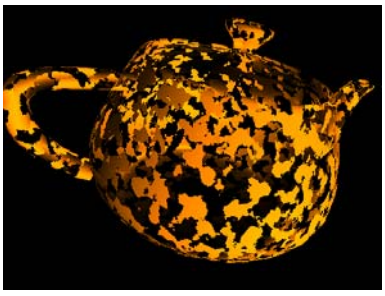
- Add detail to geometry. Start with a simple object and add detail depending on what is needed for how far you are zoomed in.



- Create apparent geometry from an equation. Perturb the normals so that the lighting makes it look like there is bumpy geometry (= “bump mapping”). This has been used for mountains on a globe, ripples in a pond, etc.

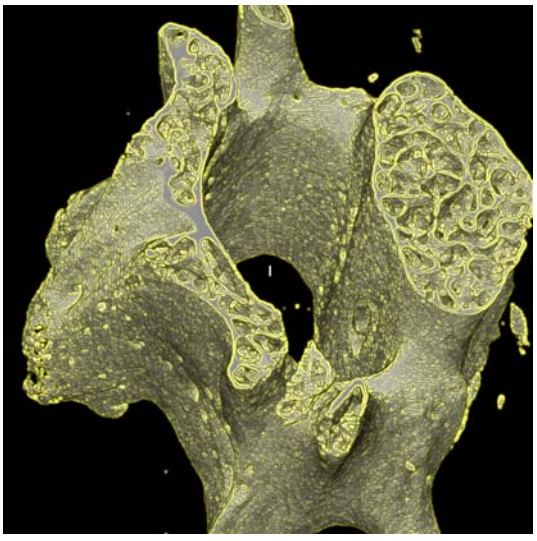



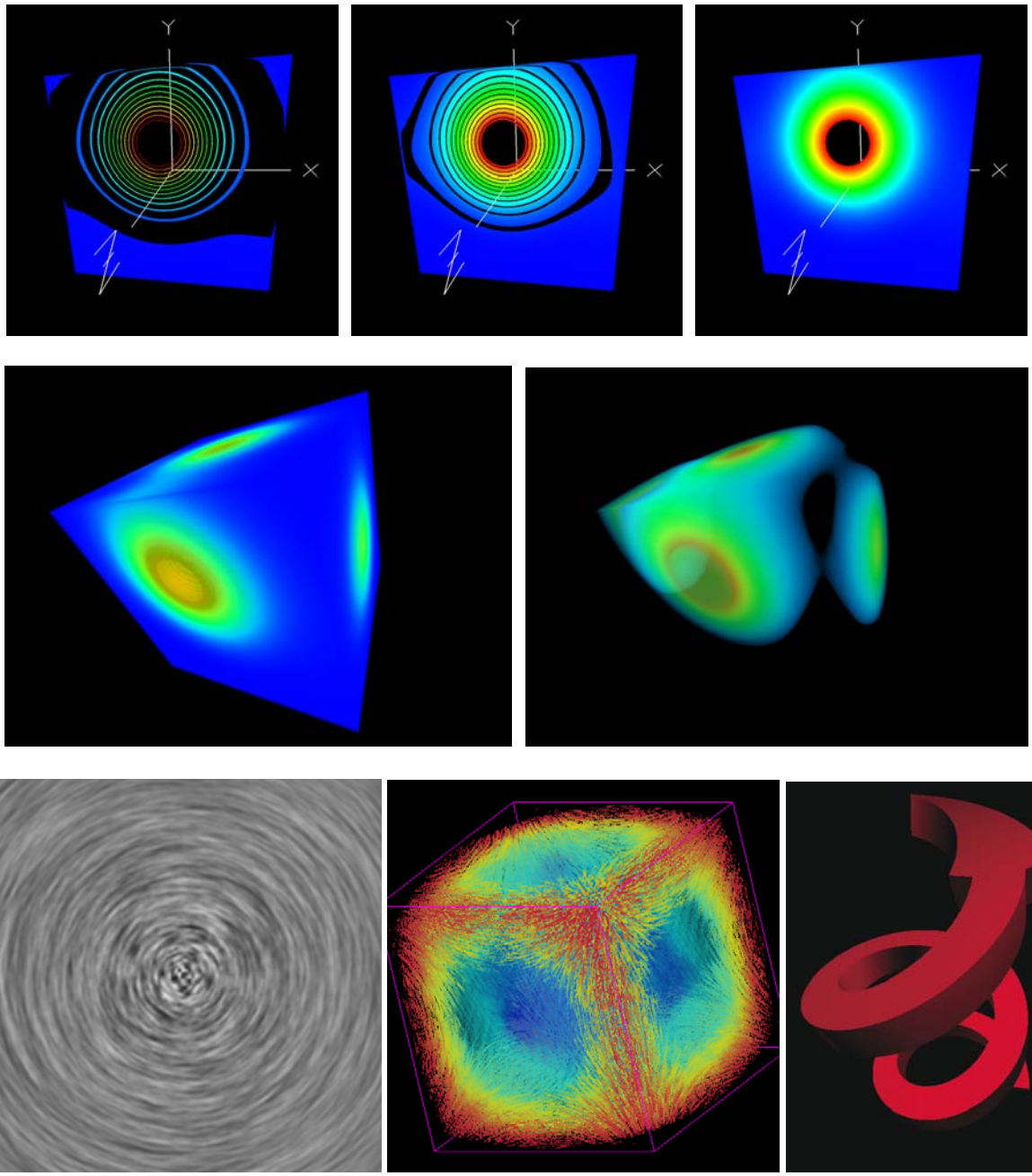
- Create special surface effects, such as clouds, wood grain, marble, a screen, or corrosion.

- You can also do interesting visualization things in shaders and take advantage of GPU speed-ups:

.



- Etc, etc. We will talk *lots* more about this in CS 519 next quarter.