Performance Graphics Programming

Mike Bailey

mjb@cs.oregonstate.edu

Oregon State University

Motivation

Why are we covering this?
There reaches a point where the amount of scientific data you are trying to display overwhelms the graphics card's abilities. At that point, you start to wonder if there are ways you can speed-up your display without buying a new graphics card. This puts you in the same league as game developers. ☺

Two Approaches

There are two major approaches to performance graphics programming:

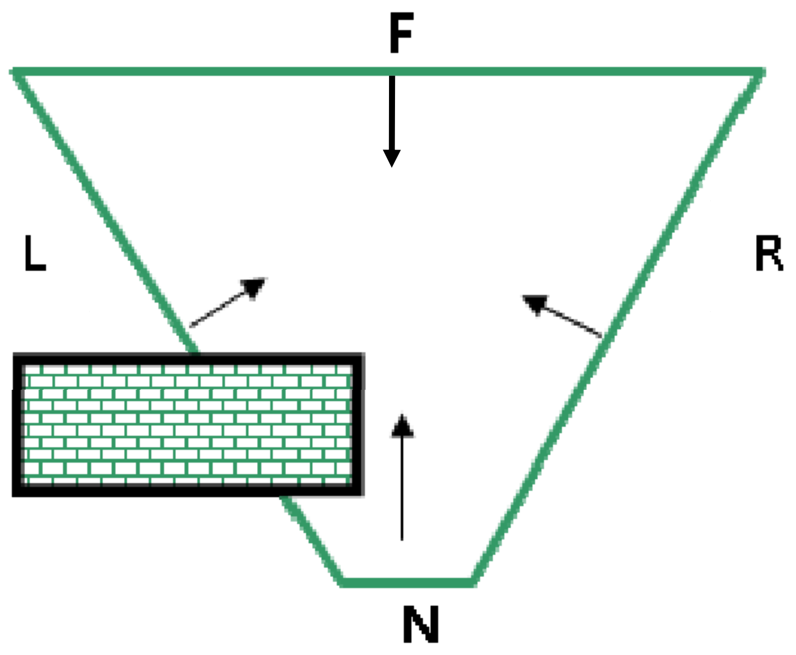
1. Eliminate large portions of the scene so that the graphics card never sees them

2. Draw the scene faster

Eliminating Scene Detail: Bounding Volumes

Quickly pre-eliminate as much scenery as possible using the CPU.

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



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 as-is and let the graphics hardware decide what's visible and what's not.

The Cohen-Sutherland Algorithm

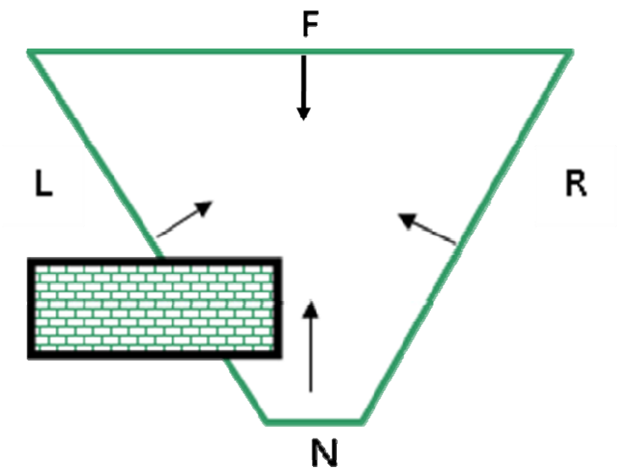
1. Determine a clip code for each of the 8 vertices of the bounding box.

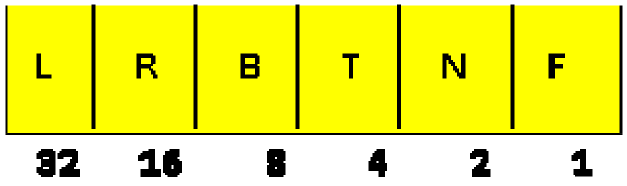
2. If you 'inclusive-or' (|) all 8 clip codes together and get **0**, then the bounding box is completely inside the viewing volume

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

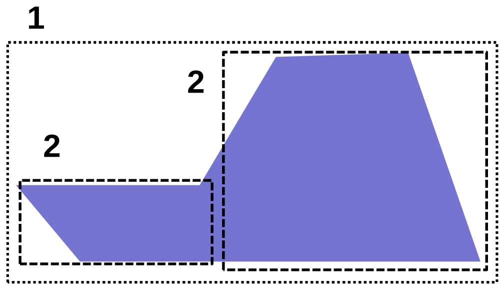
4. Sometimes a bounding sphere or bounding cylinder makes more sense. It depends on what bounding geometry most tightly fits your scene.

5. **Hierarchical Culling** -- break big things into smaller things so that more can be cleanly culled.

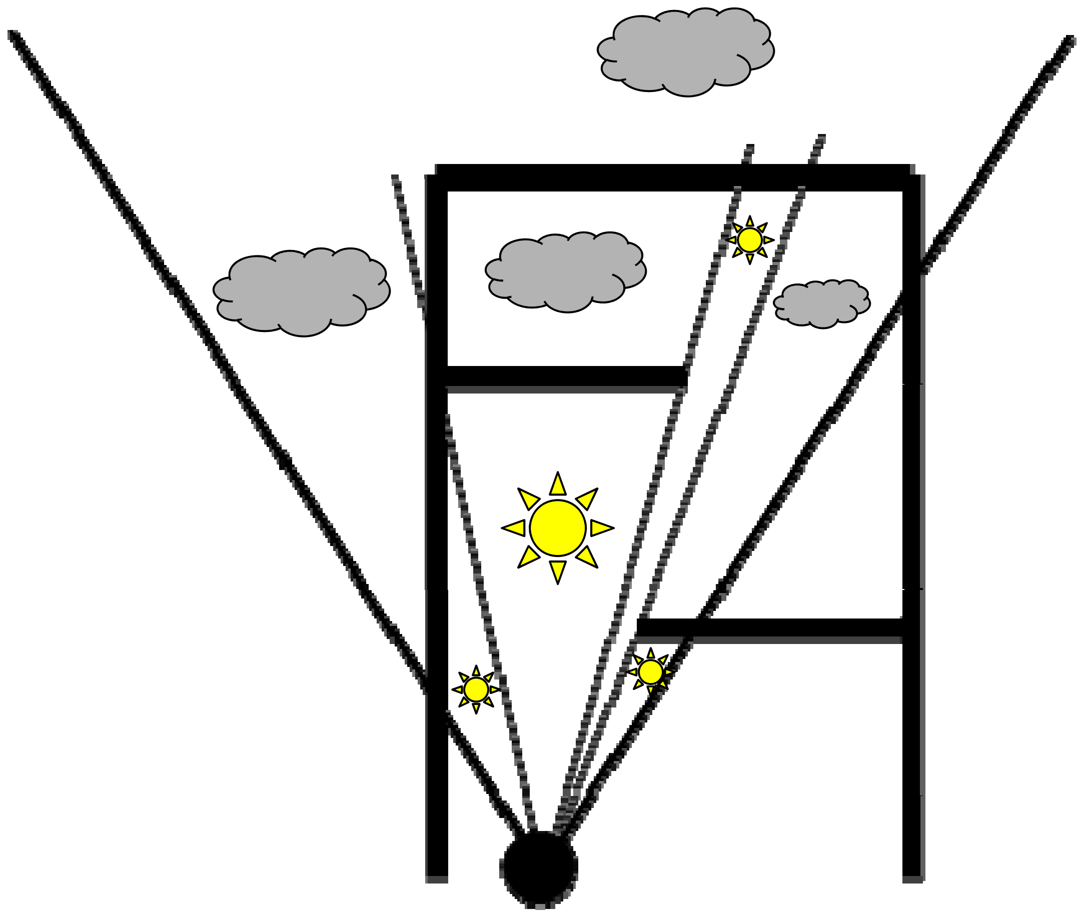




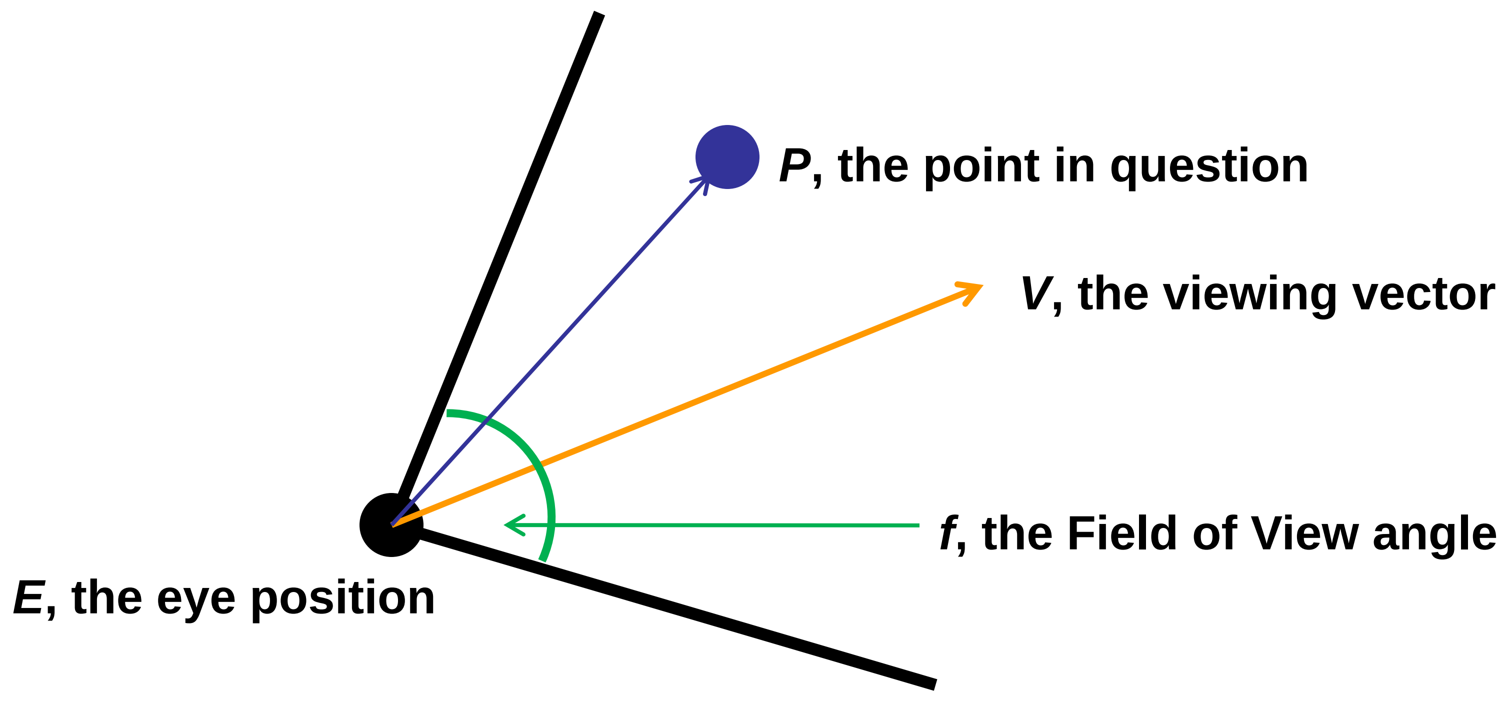
L	R	B	T	N	F
32	16	8	4	2	1



Remove Parts of the Scene
While Making the Player Think it is Part of the Game



Test if Something is within a Certain Angle of the Viewing Direction



P, is *in* the viewing volume if: $\dfrac{\hat{V} \bullet (P - E)}{\lVert P - E \rVert} > \cos\left(\dfrac{f}{2}\right)$

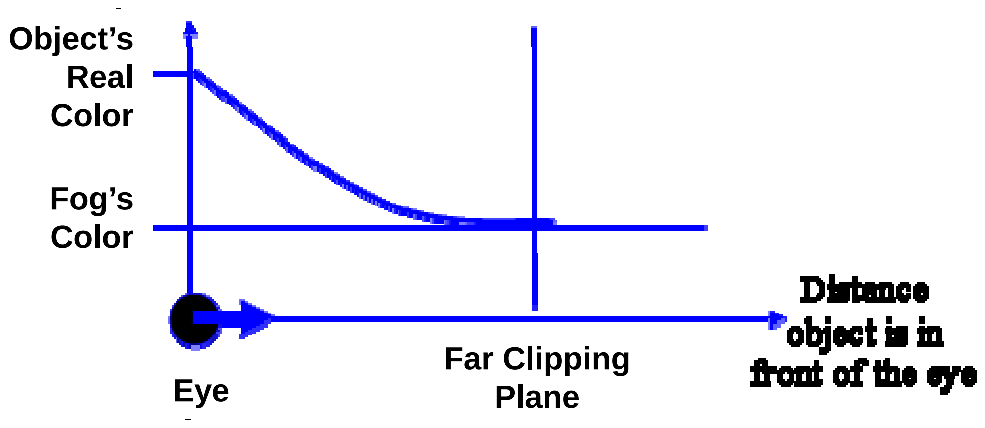
This can be pre-computed

Eliminating Scene Detail: 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 partly 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).



University of Michigan

Minimize How Much Data Needs to be Sent Across the Bus

• 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…)

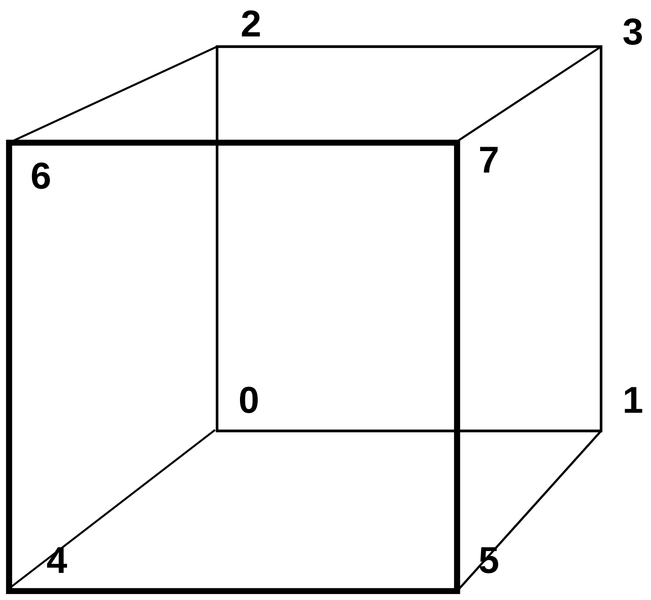
• Better: use **vertex arrays** – only transform each vertex once

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

• Maximize the size of vertex array/vertex buffer object blocks

    Small batches of geometry can kill performance
    100 triangles/batch should be a minimum
    >= 500 would be better
    Some say they try to use a size of 10,000 vertices or more

Vertex Buffers Store Arrays in GPU Memory



```
GLfloat CubeVertices[ ][3] =
{
	{ -1., -1., -1. },
	{  1., -1., -1. },
	{ -1.,  1., -1. },
	{  1.,  1., -1. },
	{ -1., -1.,  1. },
	{  1., -1.,  1. },
	{ -1.,  1.,  1. },
	{  1.,  1.,  1. }
};
```

```
GLfloat CubeColors[ ][3] =
{
	{ 0., 0., 0. },
	{ 1., 0., 0. },
	{ 0., 1., 0. },
	{ 1., 1., 0. },
	{ 0., 0., 1. },
	{ 1., 0., 1. },
	{ 0., 1., 1. },
	{ 1., 1., 1. },
};
```

```
GLuint CubeIndices[ ][4] =
{
	{ 0, 2, 3, 1 },
	{ 4, 5, 7, 6 },
	{ 1, 3, 7, 5 },
	{ 0, 4, 6, 2 },
	{ 2, 6, 7, 3 },
	{ 0, 1, 5, 4 }
};
```

Using a Vertex Buffer Object Class

Setting Up:

```
VertexBufferObject   Blob( );
Blob.CollapseCommonVertices( true );
```

Filling:

```
Blob.glBegin( GL_TRIANGLES  );    // can be any of the OpenGL topologies
            Blob.glColor3f(   r0, g0, b0 );
            Blob.glVertex3f( x0, y0, z0 );
            . . .
Blob.glEnd(  );
```

Drawing:

```
Blob.Draw( );
```

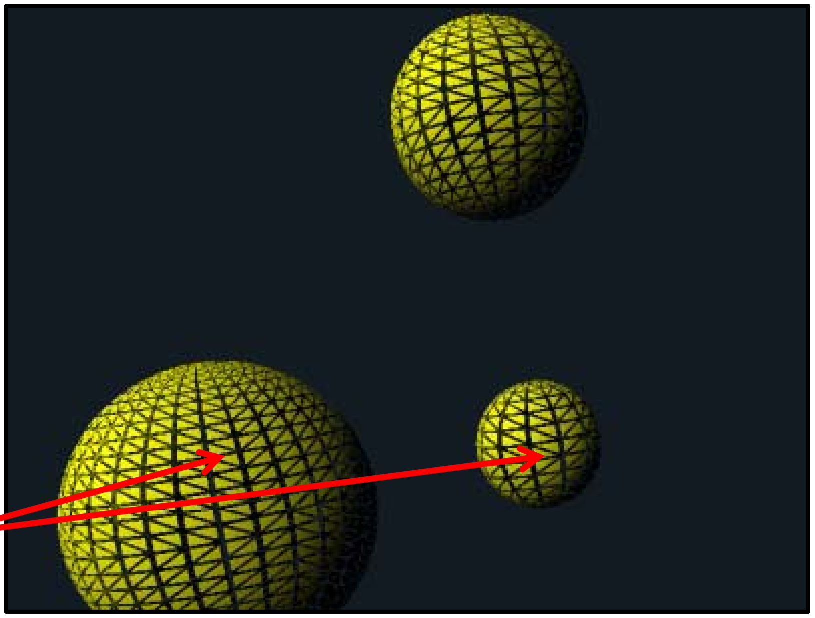
Eliminating Scene Detail: Level of Detail

- 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.)

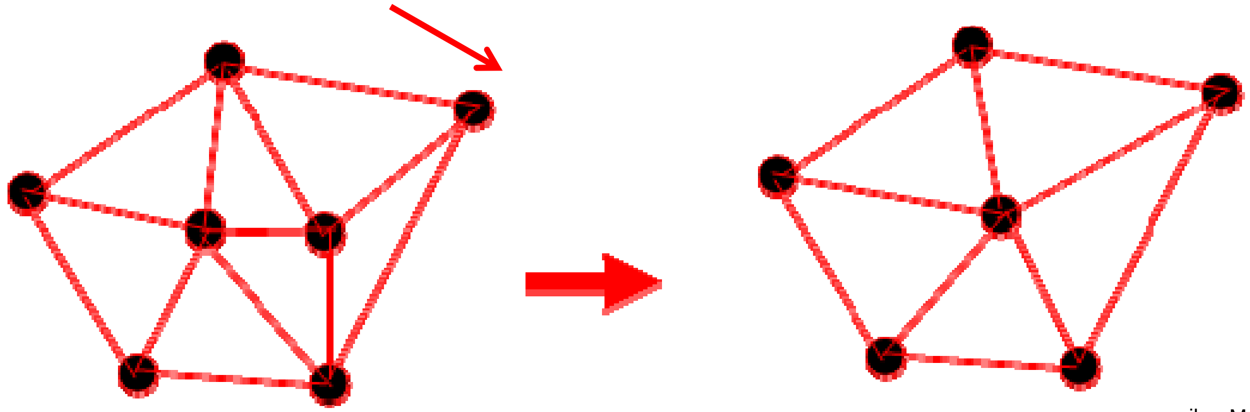
- Don't use up your "polygon budget" if you don't need to

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

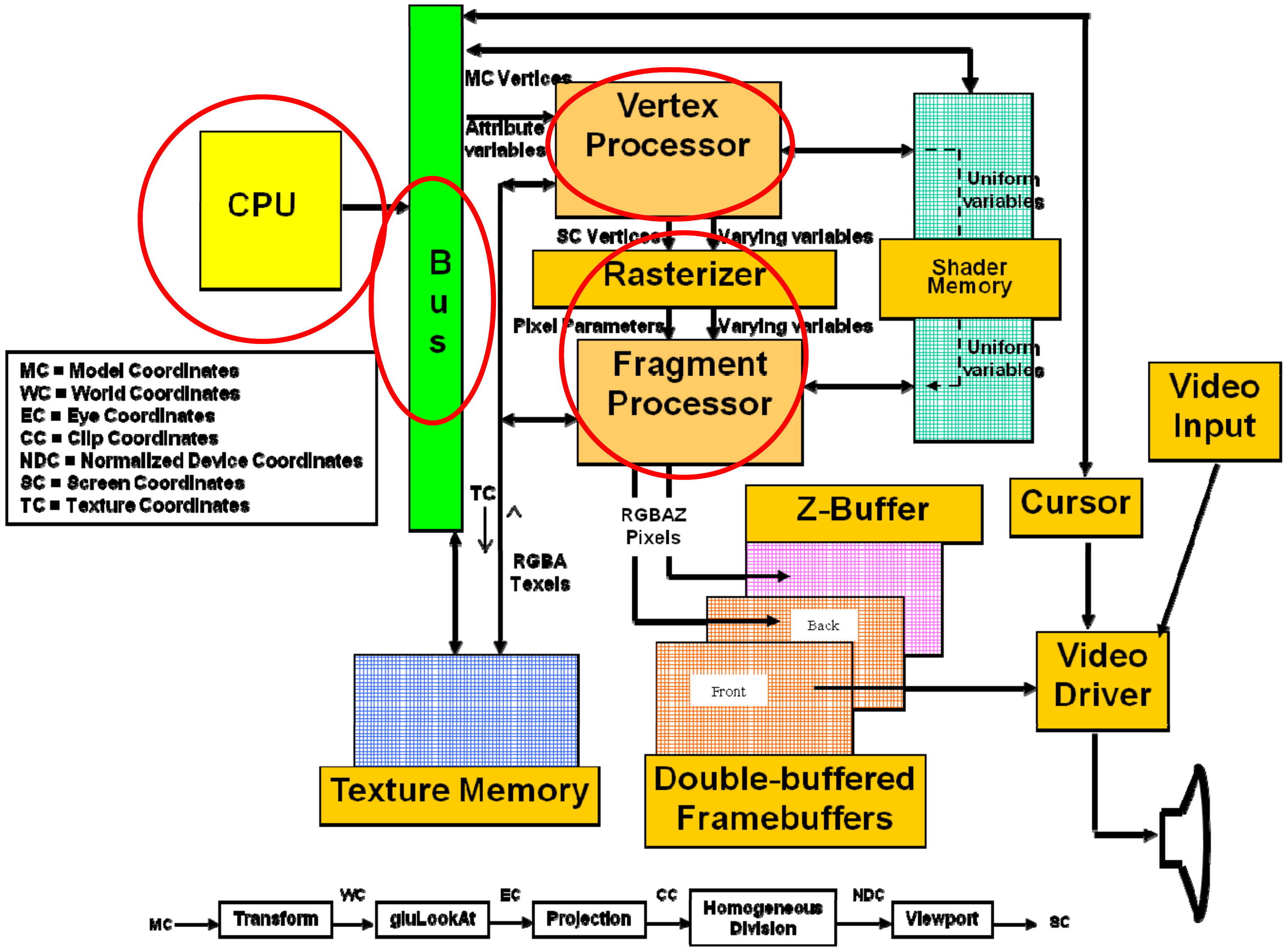
- Example #1: Different resolutions of spheres



- Example #2: Polygon mesh decimation



Drawing the Scene Faster: Four Major Performance Bottleneck Locations



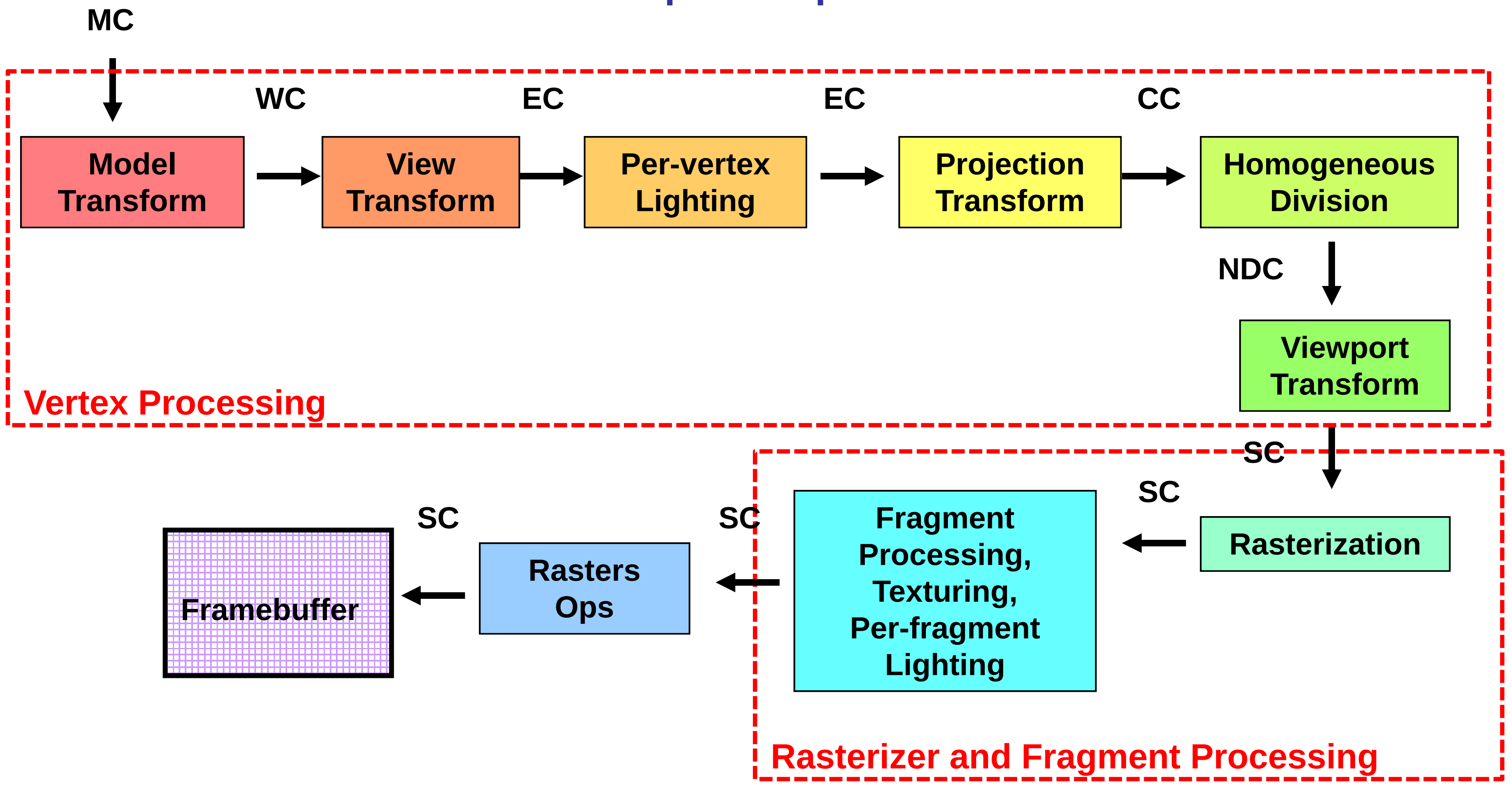
Drawing the Scene Faster: Four Major Performance Bottleneck Locations

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.

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

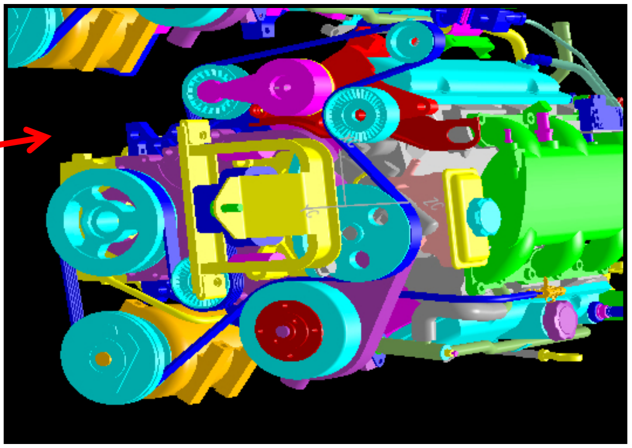
The Graphics Pipeline



MC = Model Coordinates
WC = World Coordinates
EC = Eye Coordinates
CC = Clip Coordinates
NDC = Normalized Device Coordinates
SC = Screen Coordinates

Drawing the Scene Faster: Four Major Performance Bottleneck Locations

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.

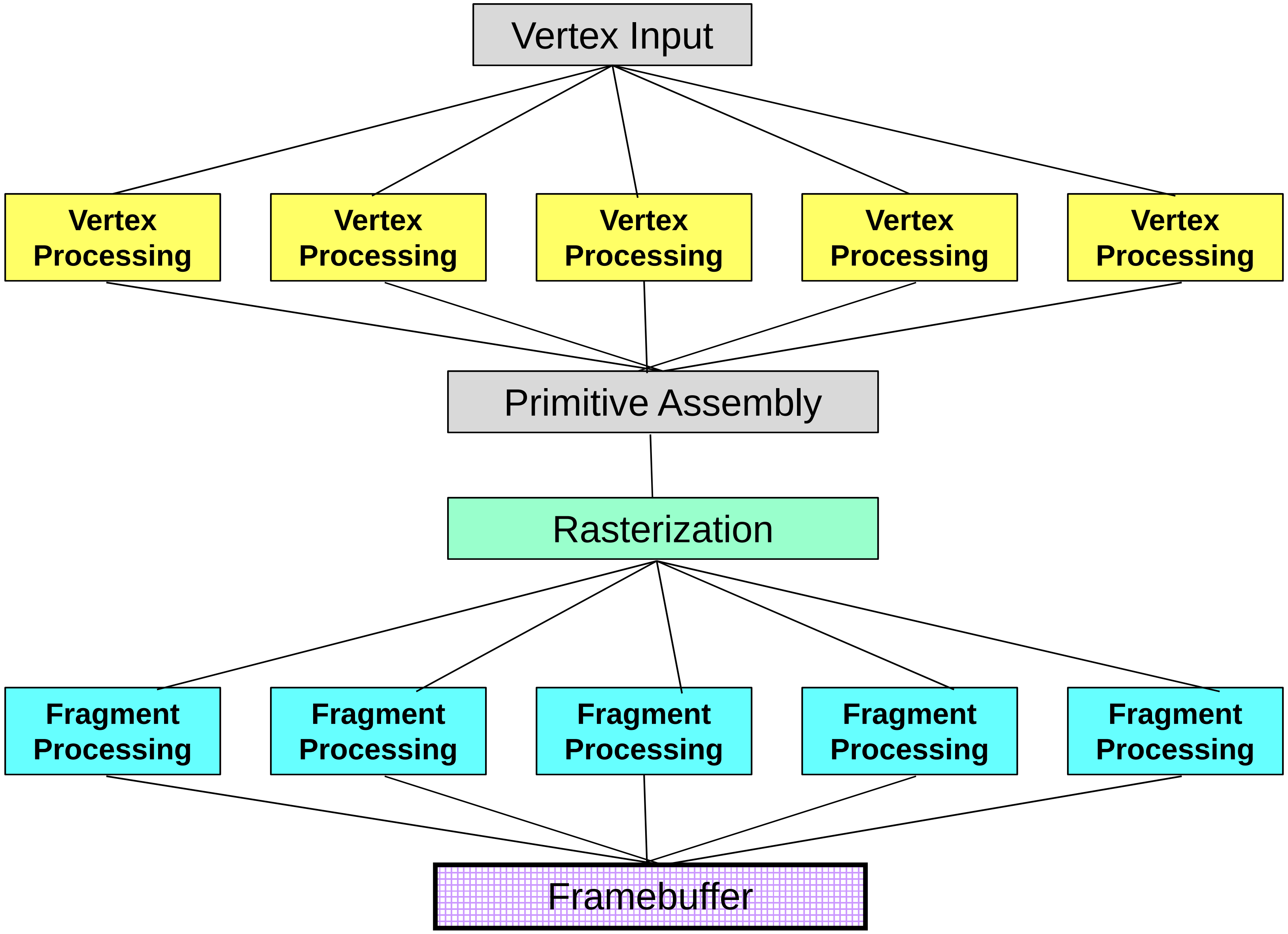


http://www.spec.org

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.

KnifeEdge

There is a Lot of Parallelism on a Graphics Card



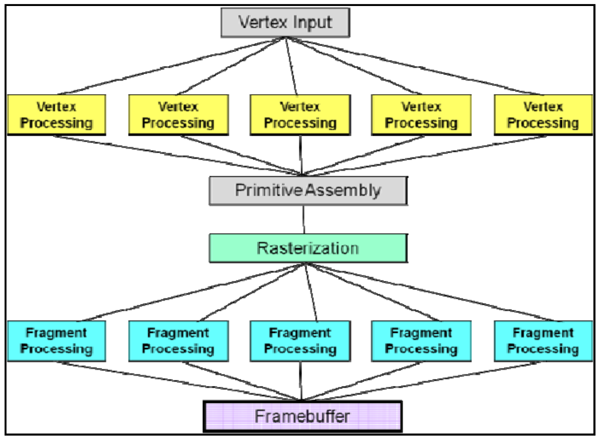
Drawing the Scene Faster: Minimize State Changes

- *Minimize all state changes.* Group similar-attribute primitives (e.g., color) together.

- ***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 often causes the pipeline to completely clear out before any new geometry can enter, which kills performance.***



Note: this sometimes contradicts the goals of object-oriented programming.

Drawing the Scene Faster: Vertex Processing

• Group like primitives in the *same* **glBegin( )** – **glEnd( )** or the same vertex buffer

• Disable lighting, or simulate it in a texture

• Use as few light sources as you can

• Use directional lights, not positional or spot

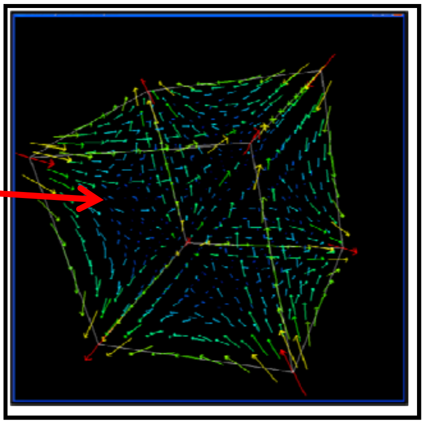
• Don’t attenuate light sources

• Use pre-unitized normals, call glDisable(GL_NORMALIZE), and don’t use **glScalef( )**

• 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.

Drawing the Scene Faster: Vertex Processing

• Pre-transform objects which are undergoing constant (“static”) transformations. Don’t use a pile of **glPushMatrix( ) - glRotatef( ) – glCallList( ) – glPopMatrix( )** calls. E.g., your vector cloud.



Arrow( ) is setup correctly for this.

• Pre-compute as much geometry as you can, especially objects involving trigonometry (e.g., spheres)

• 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( )**.

sample.cpp is setup correctly for this.

Drawing the Scene Faster: 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

Drawing the Scene Faster: Fragment Processing

• The fragment processing step in the pipeline takes all the information about a specific pixel, and computes the final color of that pixel. Given the sheer number of pixels in many scenes, this is a place that usually needs some speed-up.

• Use GL_FLAT shading

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

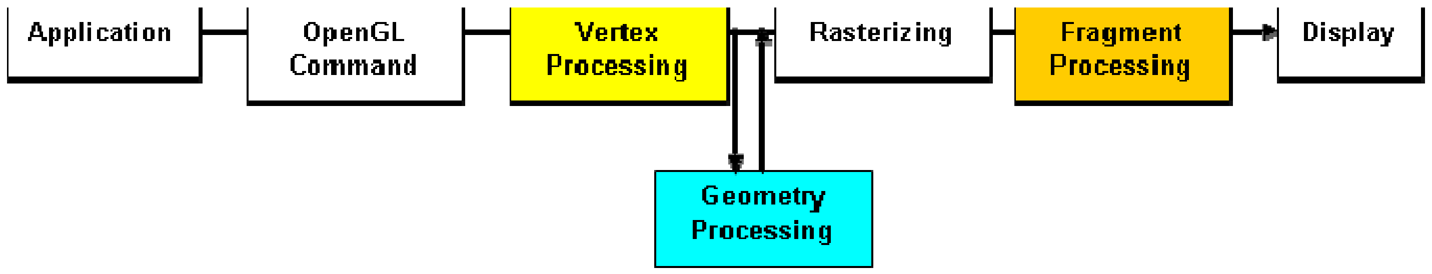
• Use **Texture Objects** (this pre-loads textures into graphics card memory)

• For fog and textures, use a **glHint( )** of GL_FASTEST.

• Use fragment shaders

Drawing the Scene Faster: Shaders

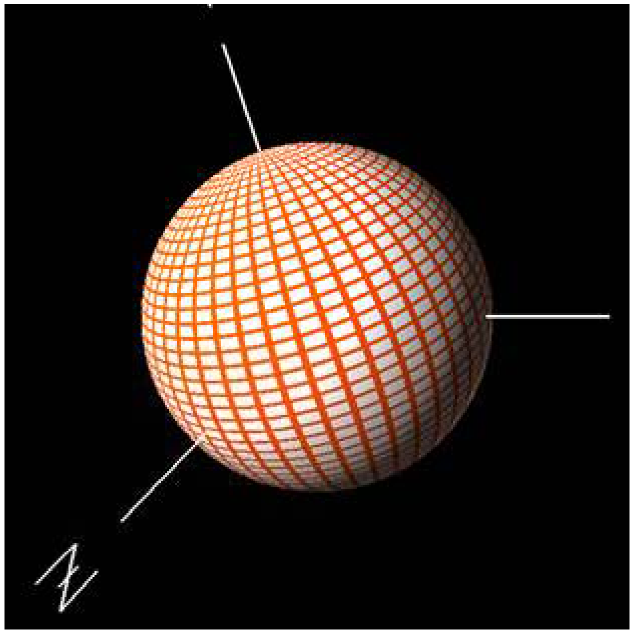
Be aware that you can place your own code in the per-vertex, geometry-creating, and per-fragment parts of the graphics hardware. This can allow you to move operations to the graphics card that used to need to be on the CPU.

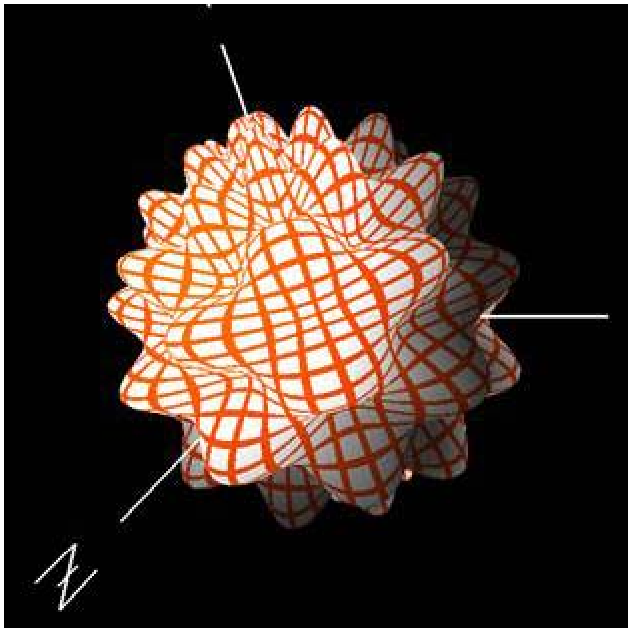


If want to learn more about this, take the Shaders class!

Drawing the Scene Faster: Vertex Shaders

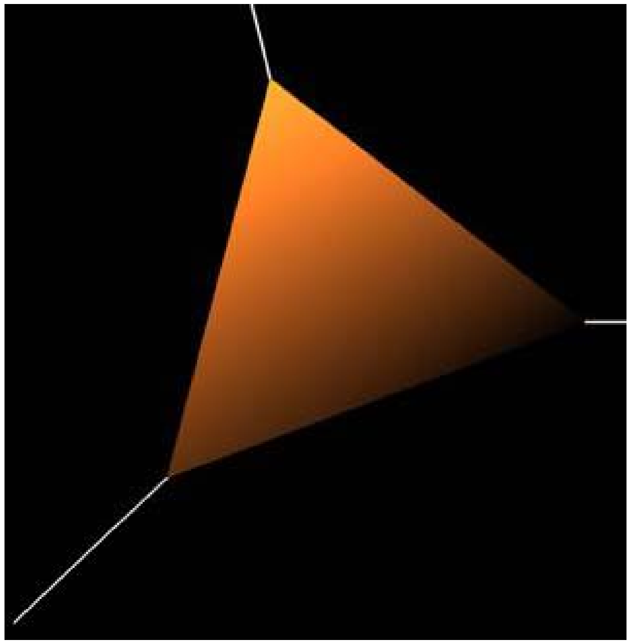
Create geometry from an equation. That is, start with a mesh of points and displace each one according to a displacement equation.

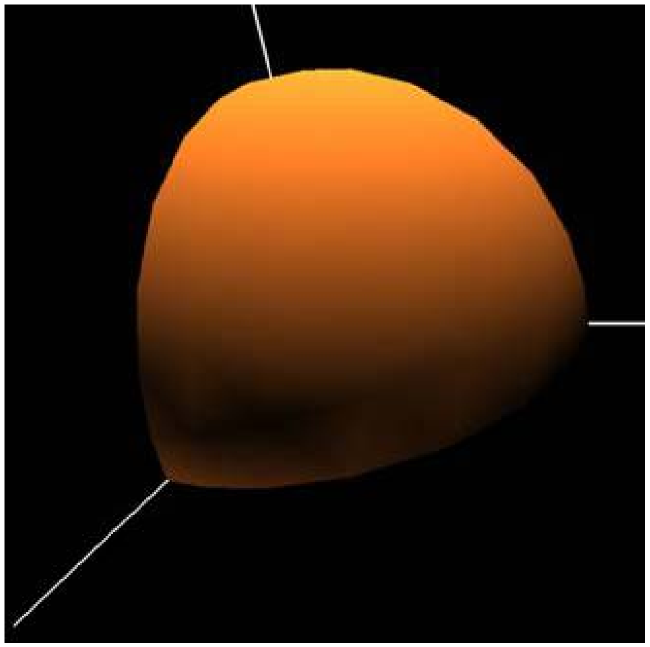




Drawing the Scene Faster: Geometry and Tessellation Shaders

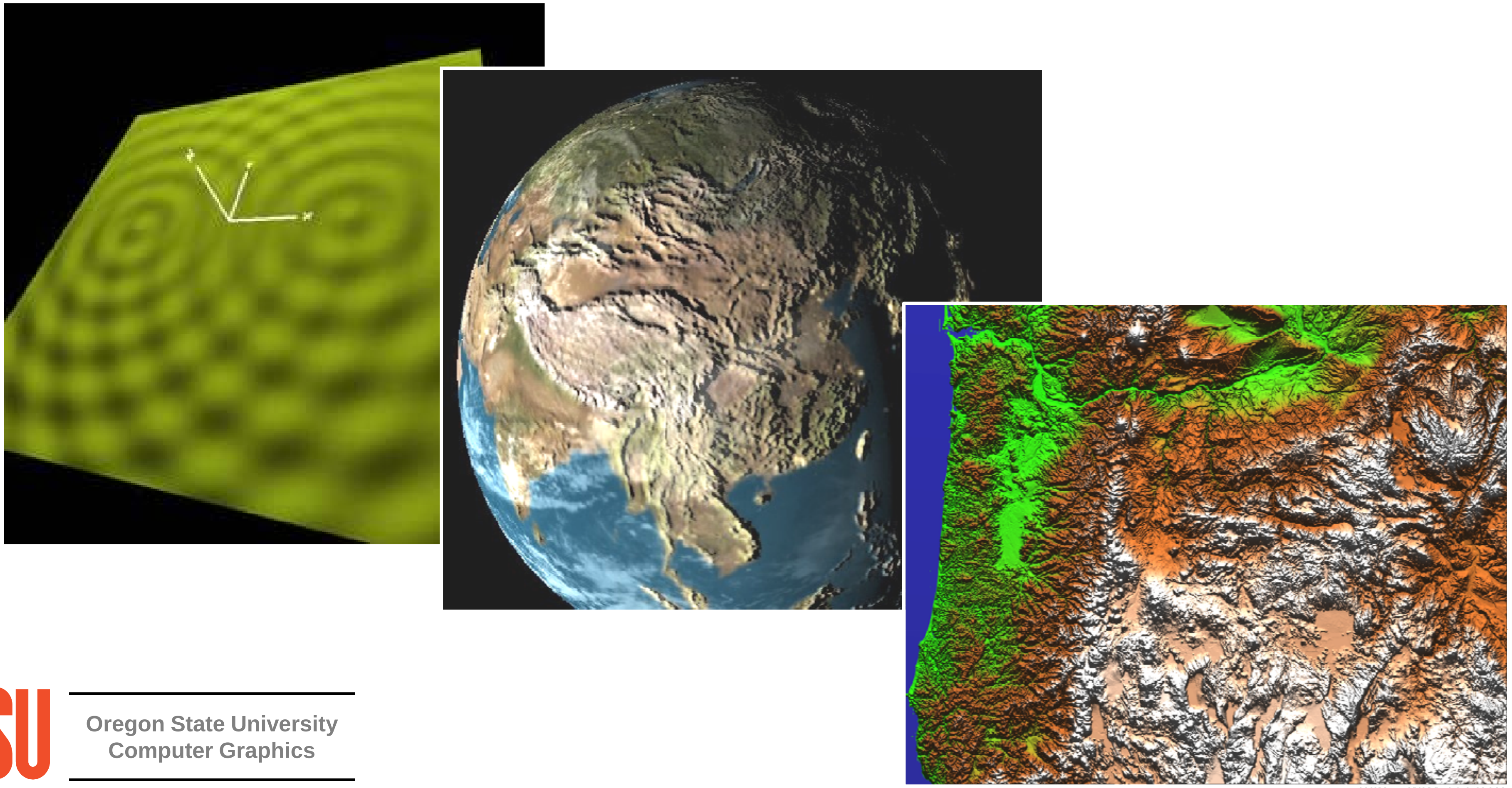
Add detail to geometry. Start with a simple object and add detail depending on what is needed for how far you are zoomed in.





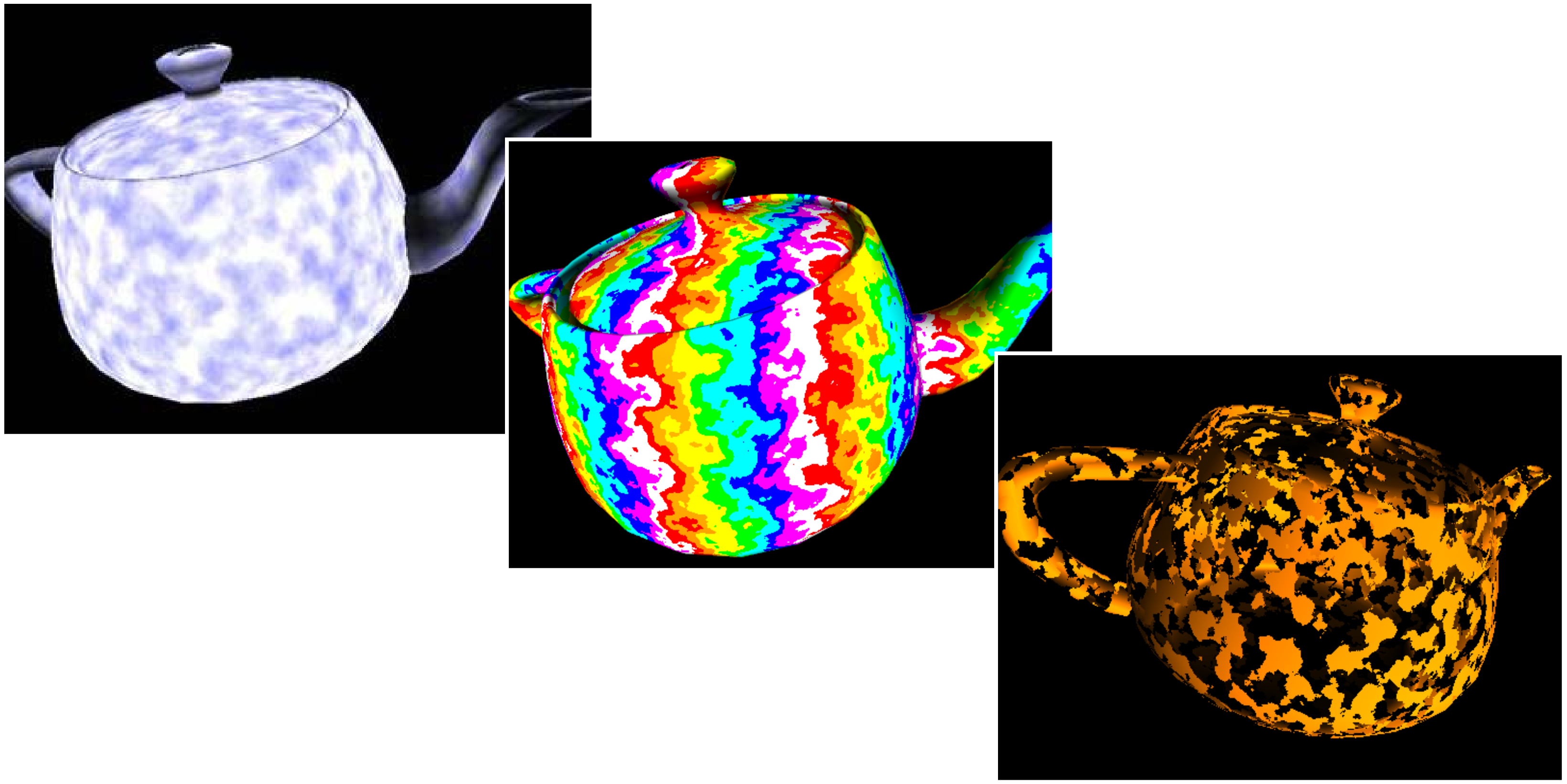
Drawing the Scene Faster: Fragment Shaders

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.



Drawing the Scene Faster: Fragment Shaders

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



Drawing the Scene Faster: Fragment Shaders

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

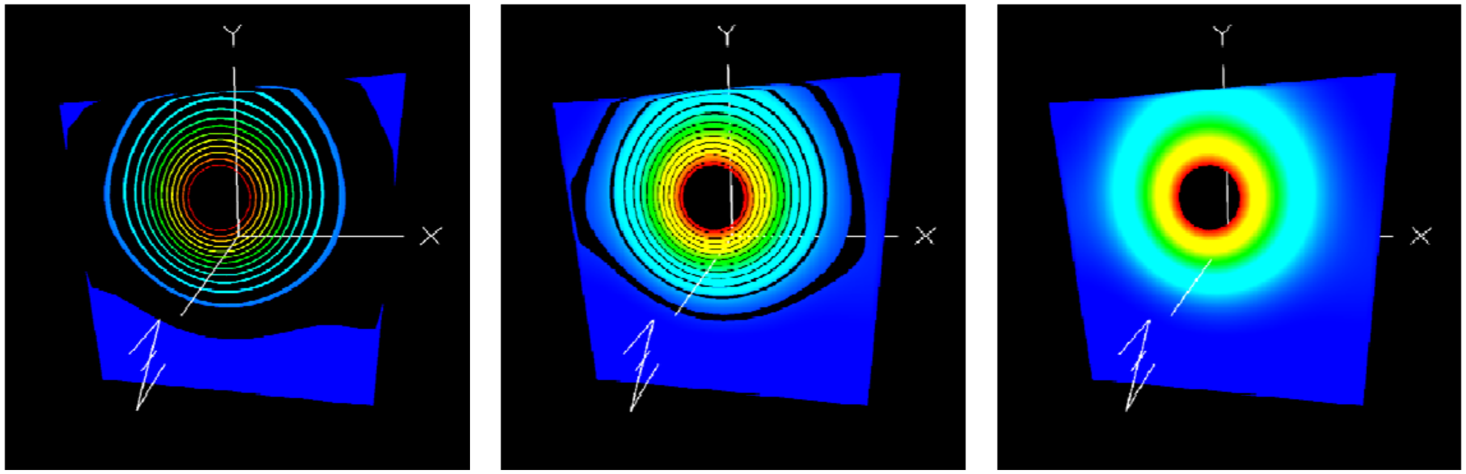
Edge detection for biological visualization

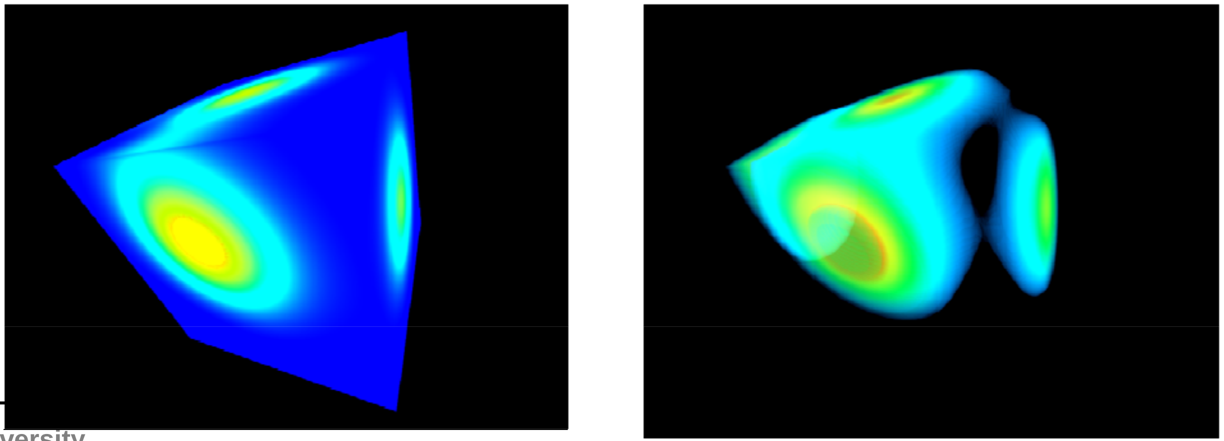
Toon rendering for architectural visualization

Drawing the Scene Faster: Fragment Shaders

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

Mash-up of interpolated colors and contour lines

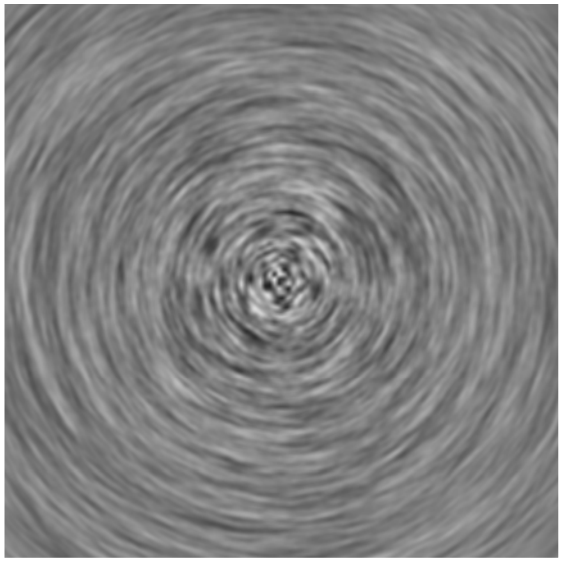


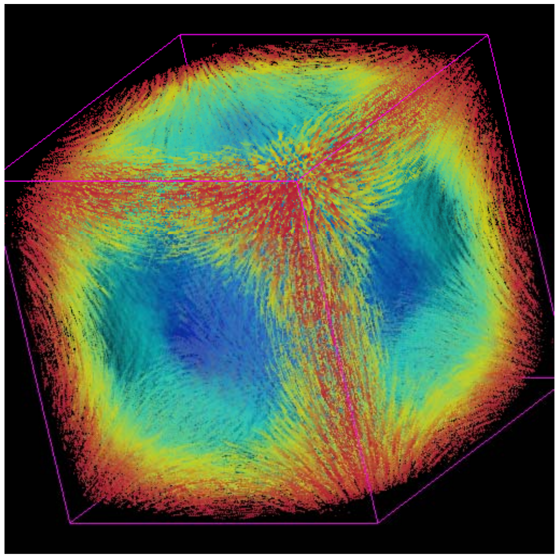


Volume rendering

Drawing the Scene Faster: Fragment Shaders

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


2D Line Integral Convolution


3D Line Integral Convolution

Streamtube

Drawing the Scene Faster: Benchmarking

• Sometimes, you just have to run your own timing tests

• Run timing tests by calling *glFinish( )* before the drawing you wish to benchmark to completely clear out the pipeline, and again right after the drawing you wish to benchmark (and before the call to *glutSwapBuffers( )*) to wait until all graphics have been processed.

• Definitely remove this for production runs !!

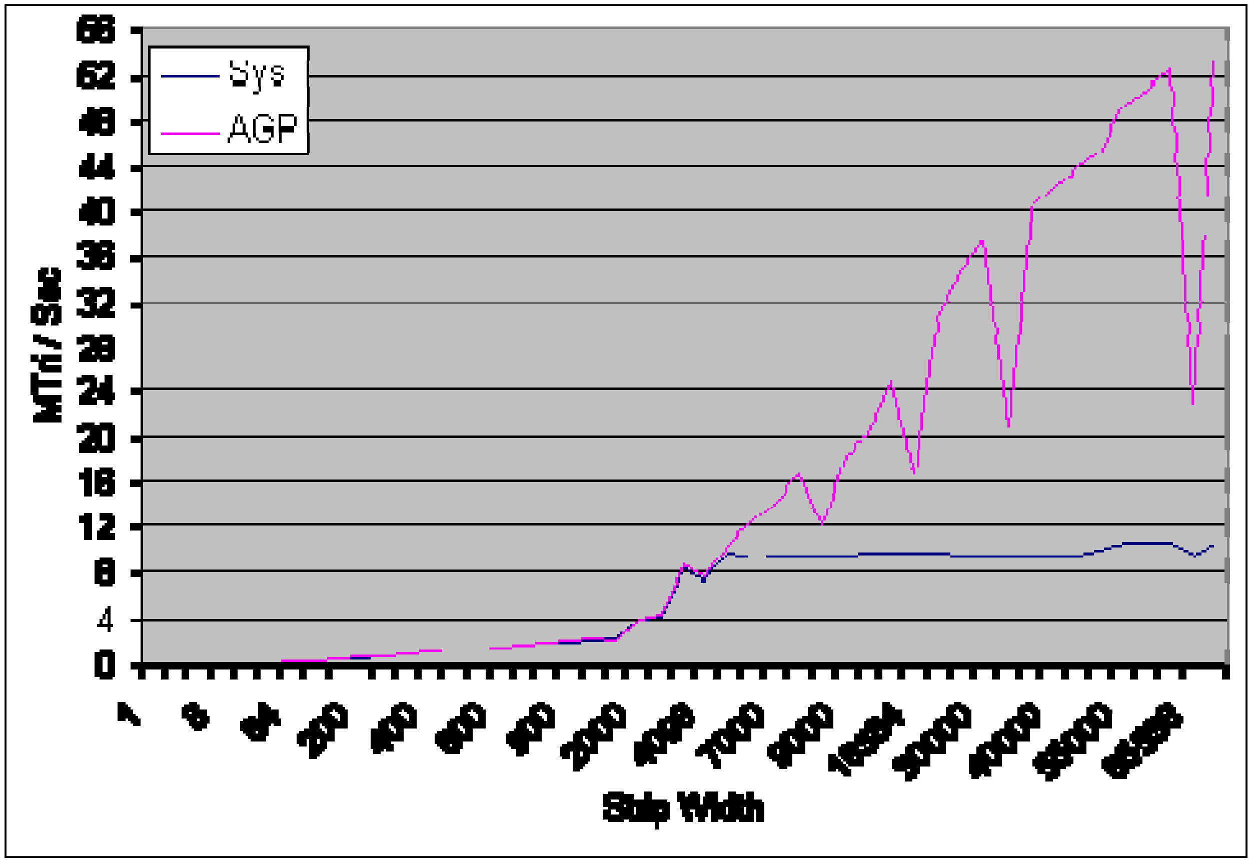
• Graphics cards today are *really* fast. Make your test size big so that the precision of the system clock is not an issue.

```
glClear( . . . );
. . .
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 );
```

• 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.

Drawing the Scene Faster: Benchmarking

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



Daniel New

Be sure to hold everything constant except for the one thing you are testing!