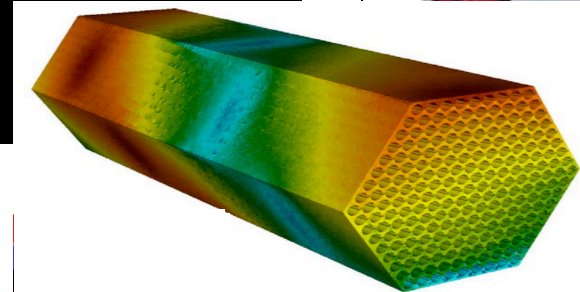
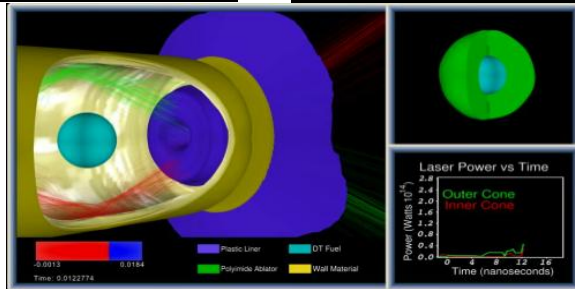
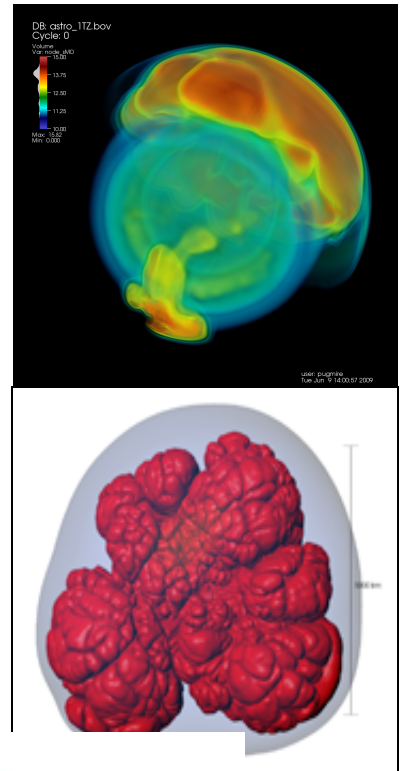
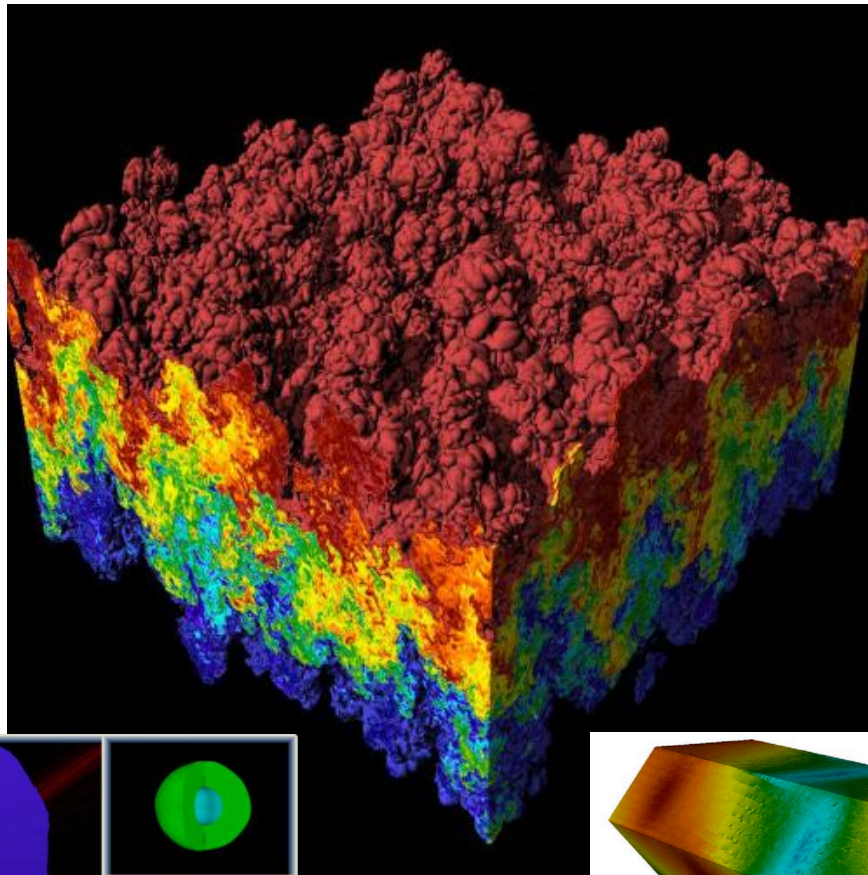
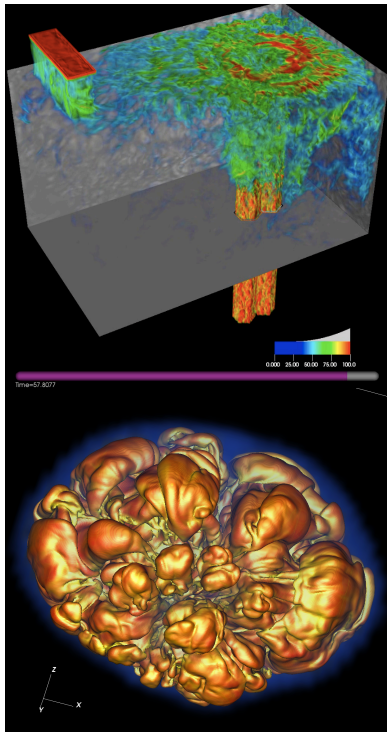


CIS 441/541: Intro to Computer Graphics

Lecture 9: More OpenGL!





Office Hours: Week 5

- Monday: 1-2 (Roscoe)
- Tuesday: 1-2 (Roscoe)
- Wednesday: 1-3 (Roscoe)
- Thursday: 1130-1230 (Hank)
- Friday: 1130-1230 (**Roscoe**)



Timeline (1/2)

- 1F: assigned Feb 7th (Feb 1), due Feb 19th
– → not as tough as 1E
- 2A: posted now, due Feb 21st
- → you need to work on both 1F and 2A during Week 6 (Feb 11-15)
- 2B: posted now, due Feb 27th
- YouTube lectures for Feb 12th and 14th



Timeline (2/2)

Sun	Mon	Tues	Weds	Thurs	Fri	Sat
	Feb 4	Feb 5 Lec 8	Feb 6 1E due	Feb 7 Begin 1F, begin 2A	Feb 8	Feb 9
Feb 10	Feb 11	YouTube	Feb 13	YouTube??	Feb 15	Feb 16
						
Feb 17	Feb 18	Feb 19 1F due	Feb 20	Feb 21 2A due, begin 2B	Feb 22	Feb 23



Comparing to previous terms (1/2)

- Way ahead on lecture
 - If I complete today's lecture, we will be 1.5 lectures ahead of the pace from previous term
 - Why?:
 - YouTube videos saving on material repeat
 - Bad materials in previous terms, and then have to waste class time fixing things
 - May only need 1 YouTube lecture from Japan



Comparing to previous terms (2/2)

- A little behind on project pace

Project	Due date (F16)	Due date (W19)
1E	Monday of Week 5	Weds of Week 5
1F	Monday of Week 6	Tuesday of Week 7
2A	Monday of Week 7	Thursday of Week 7
2B	Monday of Week 8	Wednesday of Week 8

- *The W19 plan only works if you pursue *both* 1F and 2A during Week 6!*



Midterm

- Date still not set
- Considering different plan: 25 & 5



First OpenGL programs

The University of New Mexico

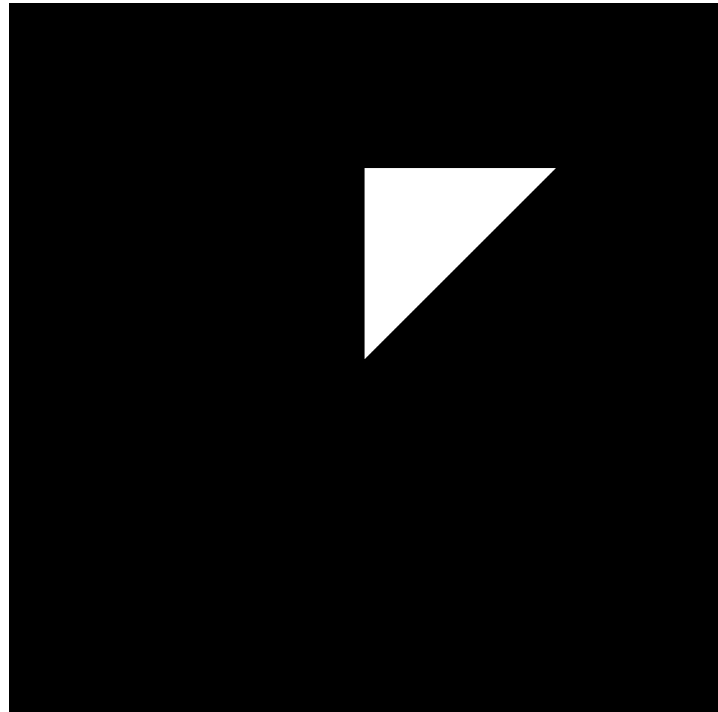
-
- Remember: none of these programs have windowing or events
 - They contain just the code to put primitives on the screen, with lighting and colors.



First OpenGL programs

The University of New Mexico

```
class vtk441PolyDataMapper : public vtkOpenGLPolyDataMapper
{
public:
    static vtk441PolyDataMapper *New();
    virtual void RenderPiece(vtkRenderer *ren, vtkActor *act)
    {
        float ambient[3] = { 1, 1, 1 };
        glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ambient);
        glBegin(GL_TRIANGLES);
        glVertex3f(0,0,0);
        glVertex3f(0,1,0);
        glVertex3f(1,1,0);
        glEnd();
    }
};
```

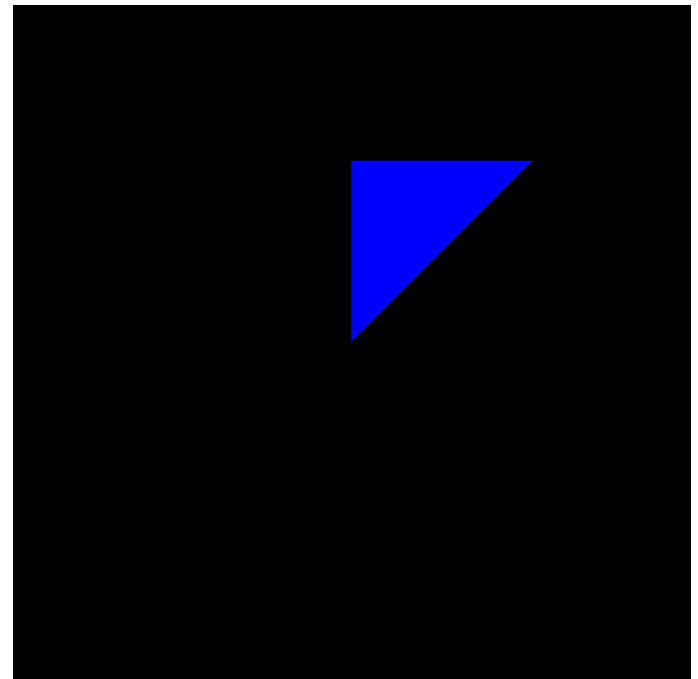




First OpenGL programs

The University of New Mexico

```
class vtk441PolyDataMapper : public vtkOpenGLPolyDataMapper
{
public:
    static vtk441PolyDataMapper *New();
    virtual void RenderPiece(vtkRenderer *ren, vtkActor *act)
    {
        → glEnable(GL_COLOR_MATERIAL);
        float ambient[3] = { 1, 1, 1 };
        glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ambient);
        glBegin(GL_TRIANGLES);
        → glColor3ub(0, 0, 255);
        glVertex3f(0,0,0);
        glVertex3f(0,1,0);
        glVertex3f(1,1,0);
        glEnd();
    }
};
```





The University of New Mexico

glEnable/glDisable: important functions

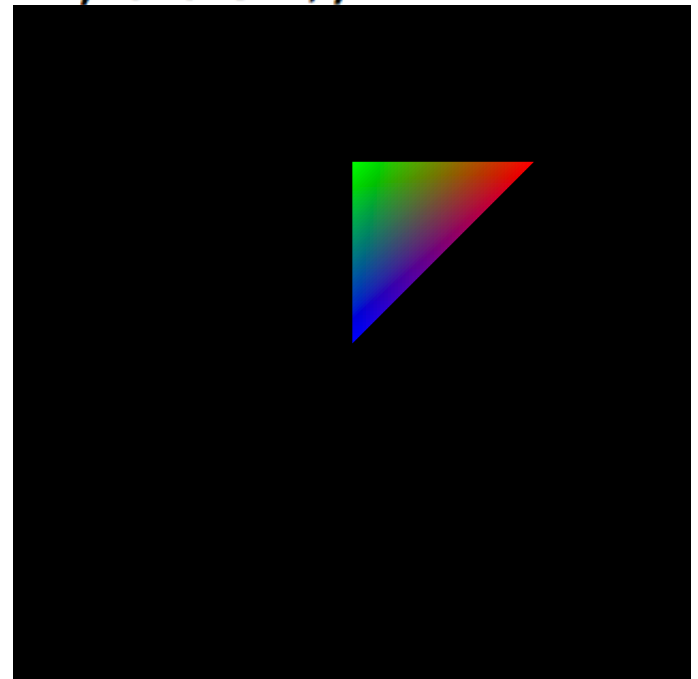
<p>← → ↻ www.opengl.org/sdk/docs/man/xhtml/glEnable.xml</p>	<p>Both <code>glEnable</code> and <code>glDisable</code> take a single argument, <code>cap</code>, which can assume one of the following values:</p>
<h2>Name</h2> <p><code>glEnable</code> — enable or disable server-side GL capabilities</p>	<p><code>GL_BLEND</code></p> <p>If enabled, blend the computed fragment color values with the values in the color buffers. See glBlendFunc.</p> <p><code>GL_CULL_FACE</code></p> <p>If enabled, cull polygons based on their winding in window coordinates. See glCullFace.</p> <p><code>GL_DEPTH_TEST</code></p> <p>If enabled, do depth comparisons and update the depth buffer. Note that even if the depth buffer exists and the depth mask is non-zero, the depth buffer is not updated if the depth test is disabled. See glDepthFunc and glDepthRange.</p> <p><code>GL_DITHER</code></p> <p>If enabled, dither color components or indices before they are written to the color buffer.</p> <p><code>GL_POLYGON_OFFSET_FILL</code></p> <p>If enabled, an offset is added to depth values of a polygon's fragments produced by rasterization. See glPolygonOffset.</p> <p><code>GL_SAMPLE_ALPHA_TO_COVERAGE</code></p> <p>If enabled, compute a temporary coverage value where each bit is determined by the alpha value at the corresponding sample location. The temporary coverage value is then ANDed with the fragment coverage value.</p> <p><code>GL_SAMPLE_COVERAGE</code></p> <p>If enabled, the fragment's coverage is ANDed with the temporary coverage value. If <code>GL_SAMPLE_COVERAGE_INVERT</code> is set to <code>GL_TRUE</code>, invert the coverage value. See glSampleCoverage.</p> <p><code>GL_SCISSOR_TEST</code></p> <p>If enabled, discard fragments that are outside the scissor rectangle. See glScissor.</p> <p><code>GL_STENCIL_TEST</code></p> <p>If enabled, do stencil testing and update the stencil buffer. See glStencilFunc and glStencilOp.</p>
<h2>C Specification</h2> <pre>void glEnable(GLenum cap);</pre>	
<h2>Parameters</h2> <p>cap</p> <p>Specifies a symbolic constant indicating a GL capability.</p>	
<h2>C Specification</h2> <pre>void glDisable(GLenum cap);</pre>	
<h2>Parameters</h2> <p>cap</p> <p>Specifies a symbolic constant indicating a GL capability.</p>	



First OpenGL programs

The University of New Mexico

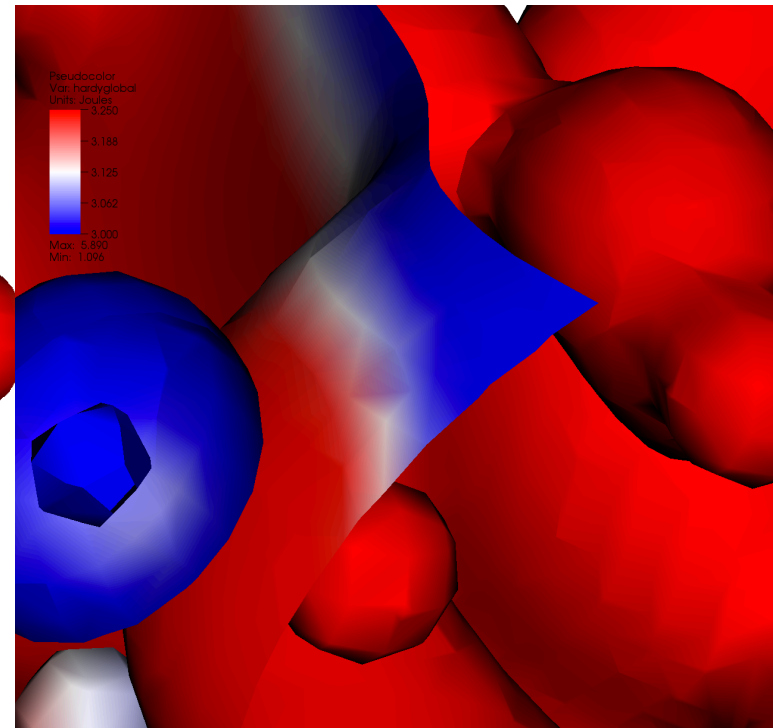
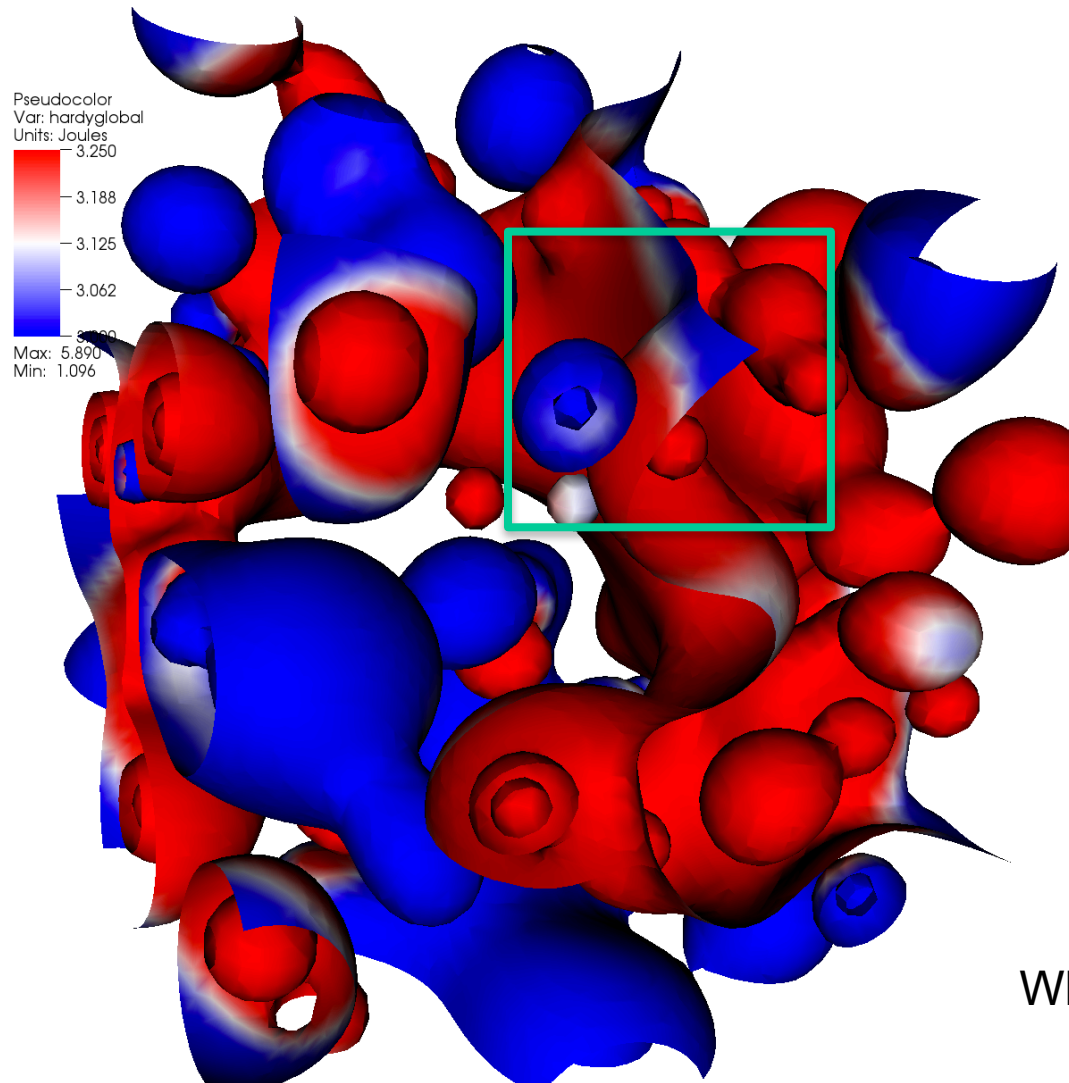
```
class vtk441PolyDataMapper : public vtkOpenGLPolyDataMapper
{
public:
    static vtk441PolyDataMapper *New();
    virtual void RenderPiece(vtkRenderer *ren, vtkActor *act)
    {
        glEnable(GL_COLOR_MATERIAL);
        float ambient[3] = { 1, 1, 1 };
        glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ambient);
        glBegin(GL_TRIANGLES);
        glColor3ub(0, 0, 255);
        glVertex3f(0,0,0);
        → glColor3ub(0, 255, 0);
        glVertex3f(0,1,0);
        → glColor3ub(255, 0, 0);
        glVertex3f(1,1,0);
        glEnd();
    }
};
```





The University of New Mexico

Visualization use case



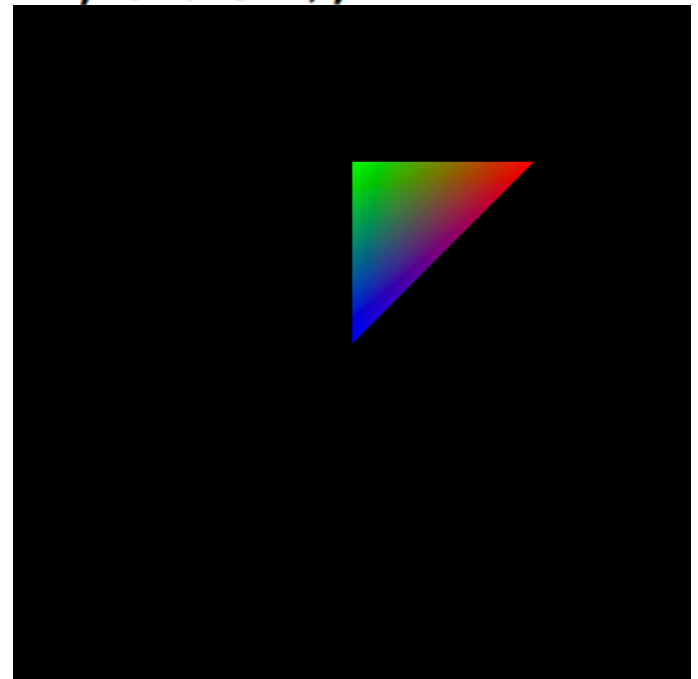
Why is there purple in this picture?



First OpenGL programs

The University of New Mexico

```
class vtk441PolyDataMapper : public vtkOpenGLPolyDataMapper
{
public:
    static vtk441PolyDataMapper *New();
    virtual void RenderPiece(vtkRenderer *ren, vtkActor *act)
    {
        glEnable(GL_COLOR_MATERIAL);
        float ambient[3] = { 1, 1, 1 };
        glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ambient);
        glBegin(GL_TRIANGLES);
        glColor3ub(0, 0, 255);
        glVertex3f(0,0,0);
        → glColor3ub(0, 255, 0);
        glVertex3f(0,1,0);
        → glColor3ub(255, 0, 0);
        glVertex3f(1,1,0);
        glEnd();
    }
};
```



Textures



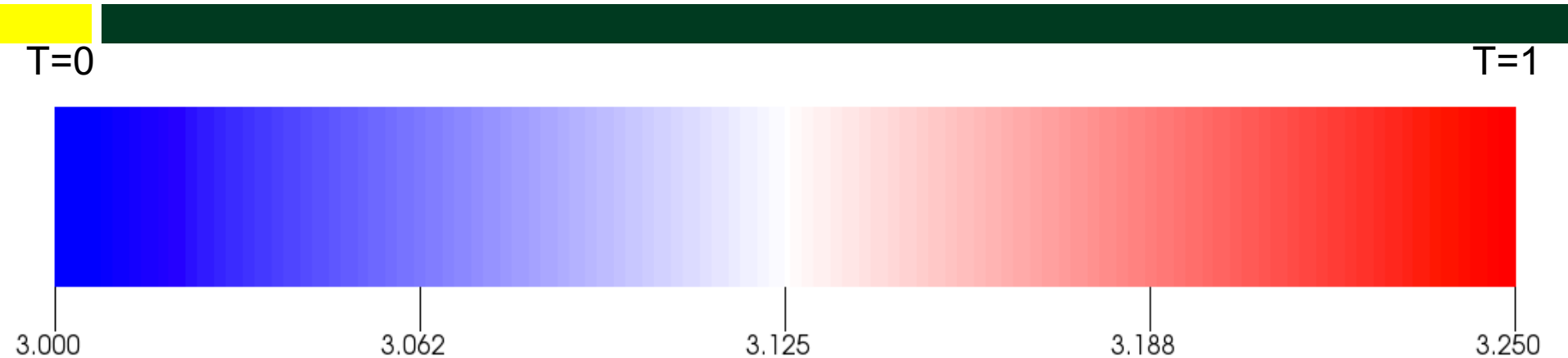
- “Textures” are a mechanism for adding “texture” to surfaces.
 - Think of texture of a cloth being applied to a surface
 - Typically used in 2D form
- We will start with a 1D form, and work our way up to 2D later.

1D textures: basic idea



- Store color map on GPU as a texture
 - An array of colors
- Old color interpolation of fragment on a scanline:
 - For (int i = 0 ; i < 3 ; i++)
 - $RGB[i] = leftRGB[i] + proportion * (rightRGB[i] - leftRGB[i])$
- New color interpolation of fragment on a scanline:
 - $textureVal = leftTextureVal + proportion * (rightTextureVal - leftTextureVal)$
 - $RGB \leftarrow textureLookup[textureVal]$

Example



- Triangle with vertices with scalar values 2.9, 3.3, and 3.1.
- $T \text{ for } 2.9 = (2.9 - 3.0) / (3.25 - 3) = -0.4$
- $T \text{ for } 3.1 = (3.1 - 3.0) / (3.25 - 3) = 0.4$
- $T \text{ for } 3.3 = (3.3 - 3.0) / (3.25 - 3) = 1.2$
- Fragment colors come from interpolating texture coordinates and applying texture

First OpenGL Texture Program

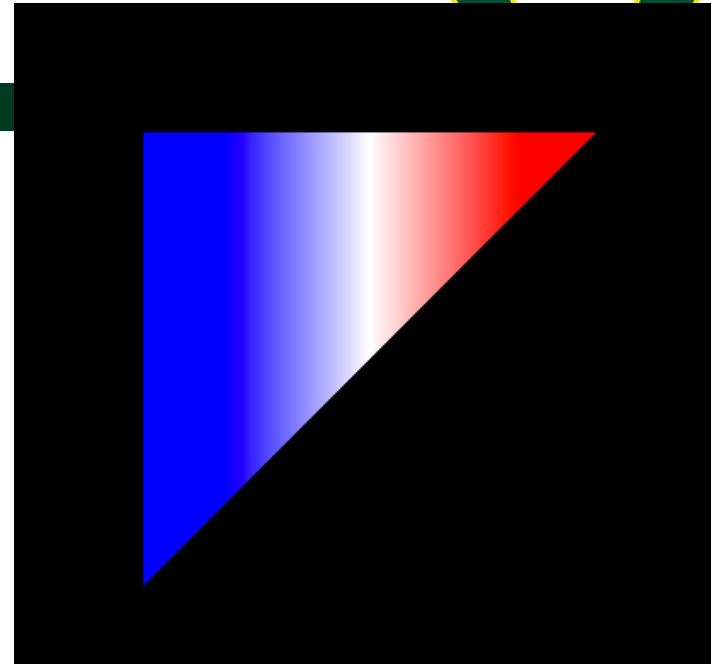


```
class vtk441PolyDataMapper : public vtkOpenGLPolyDataMapper
{
public:
    static vtk441PolyDataMapper *New();

    virtual void RenderPiece(vtkRenderer *ren, vtkActor *act)
    {
        GLubyte Texture3[9] = {
            0, 0, 255,    // blue
            255, 255, 255, // white
            255, 0, 0,    // red
        };

        glTexImage1D(GL_TEXTURE_1D, 0, GL_RGB, 3, 0, GL_RGB,
                     GL_UNSIGNED_BYTE, Texture3);
        glEnable(GL_COLOR_MATERIAL);
        glTexParameterf(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        glTexParameterf(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

        glEnable(GL_TEXTURE_1D);
        float ambient[3] = { 1, 1, 1 };
        glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ambient);
        glBegin(GL_TRIANGLES);
        glTexCoord1f(0);
        glVertex3f(0,0,0);
        glTexCoord1f(0.0);
        glVertex3f(0,1,0);
        glTexCoord1f(1.);
        glVertex3f(1,1,0);
        glEnd();
    }
};
```



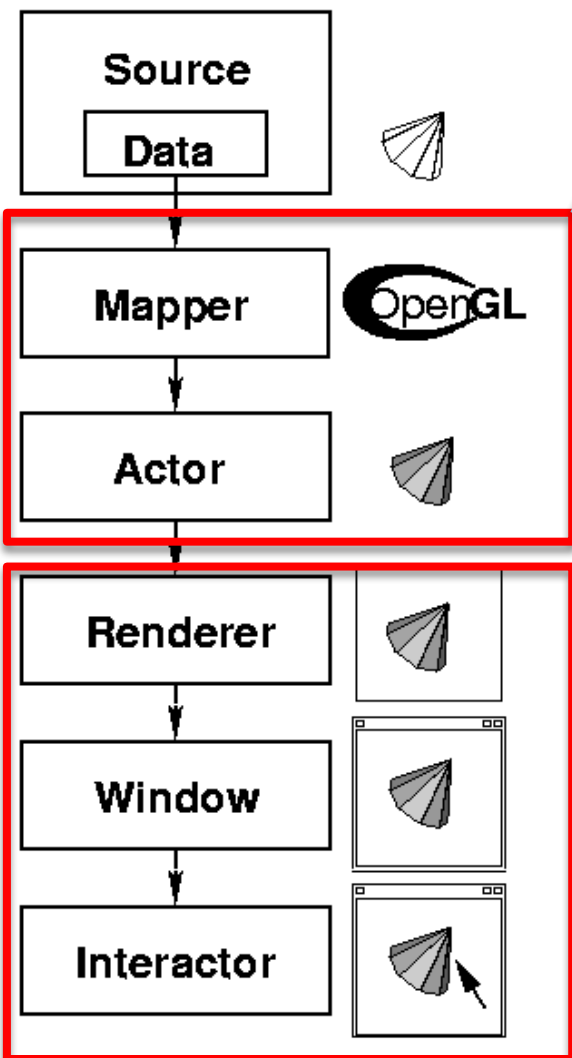
(advanced texture features &
2D textures on Weds)

Project 2A (START RECORDING)



We will replace these and write our own GL calls.

Cone.py Pipeline Diagram (type "python Cone.py" to run)



Either reads the data from a file or creates the data from scratch.

Moves the data from VTK into OpenGL.

For setting colors, surface properties, and the position of the object.

The rectangle of the computer screen that VTK draws into.

The window, including title bar and decorations.

Allows the mouse to be used to interact with the data.

```
from vtkpython import *
```

```
cone = vtkConeSource()  
cone.SetResolution(10)
```

```
coneMapper = vtkPolyDataMapper()  
coneMapper.SetInput(cone.GetOutput())
```

```
coneActor = vtkActor()  
coneActor.SetMapper(coneMapper)
```

```
ren = vtkRenderer()  
ren.AddActor(coneActor)
```

```
renWin = vtkRenderWindow()  
renWin.SetWindowName("Cone")  
renWin.SetSize(300,300)  
renWin.AddRenderer(ren)
```

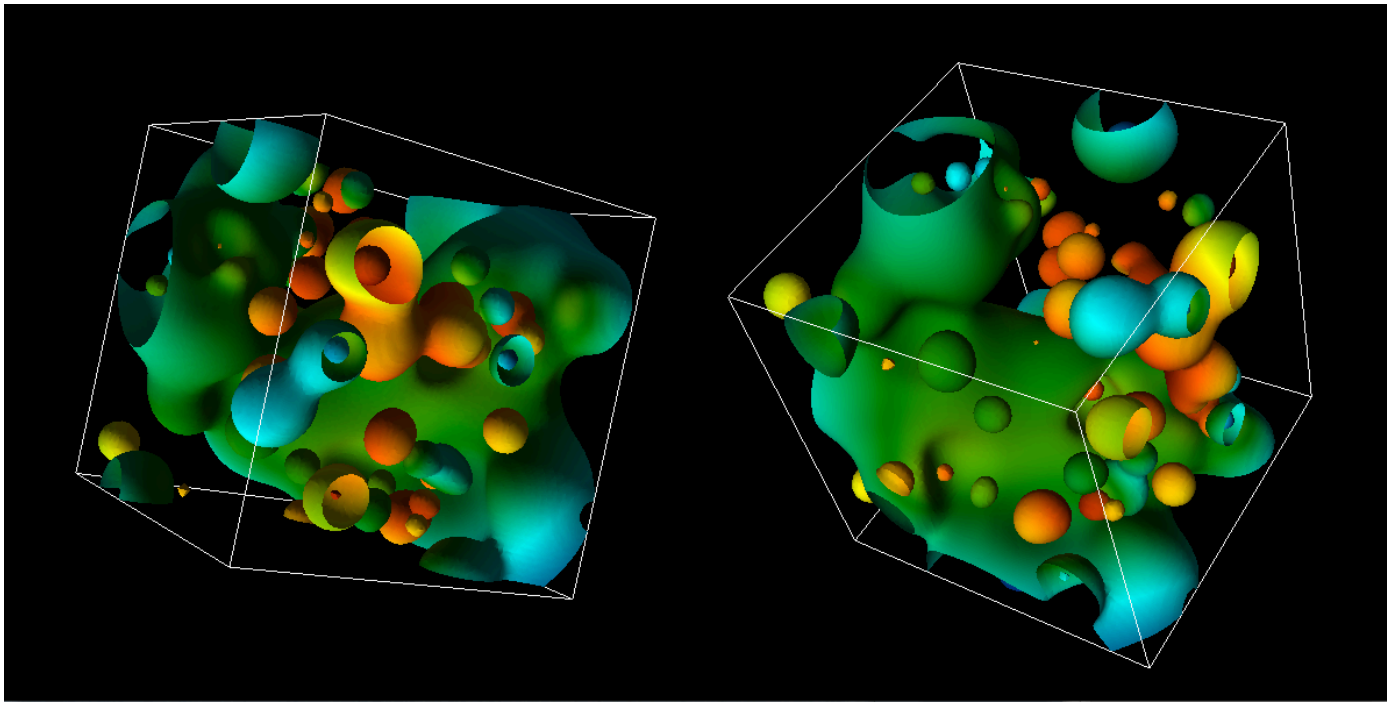
```
iren = vtkRenderWindowInteractor()  
iren.SetRenderWindow(renWin)  
iren.Initialize()  
iren.Start()
```

We will re-use these.

Project #2A (8%), Due Feb. 21st



- Goal: OpenGL program that does regular colors and textures
- New VTK-based `project2A.cxx`
- New `CMakeLists.txt` (but same as old ones)



Hints



- ❑ I recommend you “walk before you run” & “take small bites”. OpenGL can be very punishing. Get a picture up and then improve on it. Make sure you know how to retreat to your previously working version at every step.
- ❑ OpenGL “state thrashing” is common and tricky to debug.
 - ❑ Get one window working perfectly.
 - ❑ Then make the second one work perfectly.
 - ❑ Then try to get them to work together.
 - Things often go wrong, when one program leaves the OpenGL state in a way that doesn’t suit another renderer.

Hints



- ❑ MAKE MANY BACKUPS OF YOUR PROGRAM
- ❑ USE VTK 6
- ❑ If you are having issues on your laptop with a GL program, then use Room 100
 - ❑ (There's only 2 of these projects)

How to do colors (traditional)...



- The Triangle class now has a “fieldValue” data member, which ranges between 0 and 1.
- You will map this to a color using the `GetColorMap` function.
 - `GetColorMap` returns 256 colors.
- Mappings
 - A fieldValue value of 0 should be mapped to the first color
 - A fieldValue value of 1 should be mapped to the 255th color.
 - Each fieldValue in between should be mapped to the closest color of the 256, but interpolation of colors is not required.

How to do colors (texture)...



- Same idea, but use texture infrastructure
- (easier)

END RECORDING



More on Textures



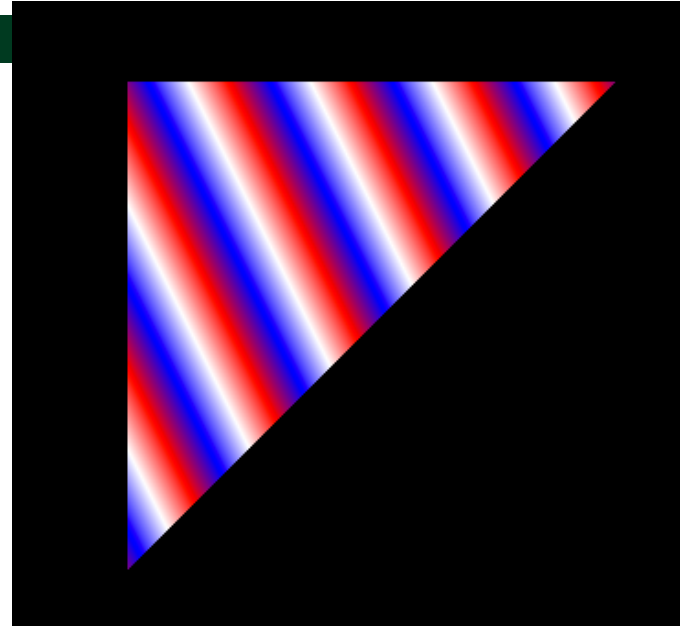
Textures with GL_REPEAT



```
class vtk441PolyDataMapper : public vtkOpenGLPolyDataMapper
{
public:
    static vtk441PolyDataMapper *New();

    virtual void RenderPiece(vtkRenderer *ren, vtkActor *act)
    {
        GLubyte Texture3[9] = {
            0, 0, 255,    // blue
            255, 255, 255, // white
            255, 0, 0,    // red
        };
        glTexImage1D(GL_TEXTURE_1D, 0, GL_RGB, 3, 0, GL_RGB,
                     GL_UNSIGNED_BYTE, Texture3);
        glEnable(GL_COLOR_MATERIAL);
        → glTexParameterf(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_REPEAT);
        glTexParameterf(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

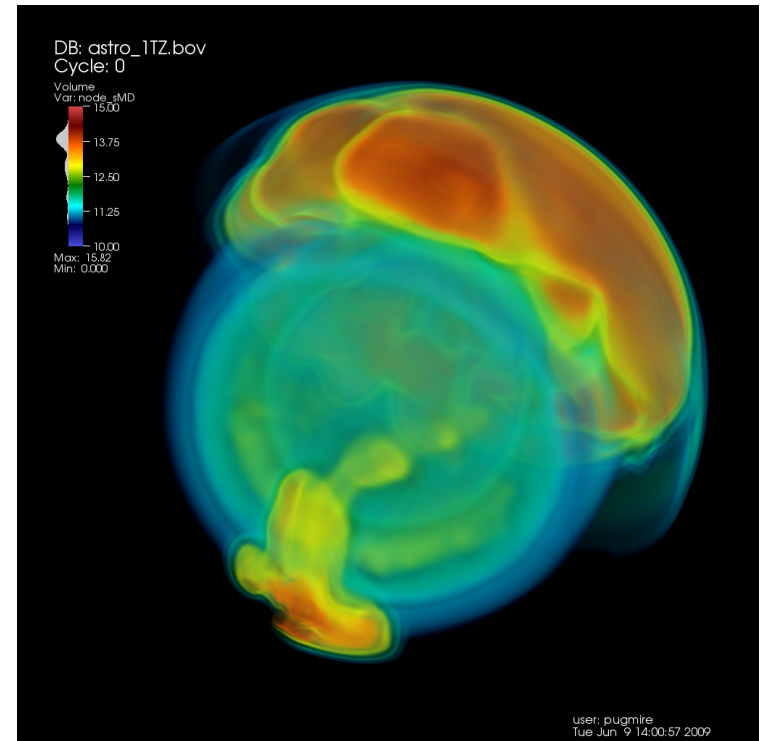
        glEnable(GL_TEXTURE_1D);
        float ambient[3] = { 1, 1, 1 };
        glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ambient);
        glBegin(GL_TRIANGLES);
        → glTexCoord1f(-2);
        glVertex3f(0,0,0);
        → glTexCoord1f(0);
        glVertex3f(0,1,0);
        → glTexCoord1f(4.);
        glVertex3f(1,1,0);
        glEnd();
    }
};
```



1D, 2D, 3D textures



- 2D textures most common
- 1D textures: color maps (e.g., what we just did)
- 3D textures: “volume rendering”
 - Use combination of opacity and color (i.e., RGBA)



2D Textures



- Pre-rendered images painted onto geometry
- `glTexImage1D` → `glTexImage2D`
- `GL_TEXTURE_WRAP_S` → `GL_TEXTURE_WRAP_S`
+ `GL_TEXTURE_WRAP_T`
- `glTexCoord1f` → `glTexCoord2f`

2D Texture Program



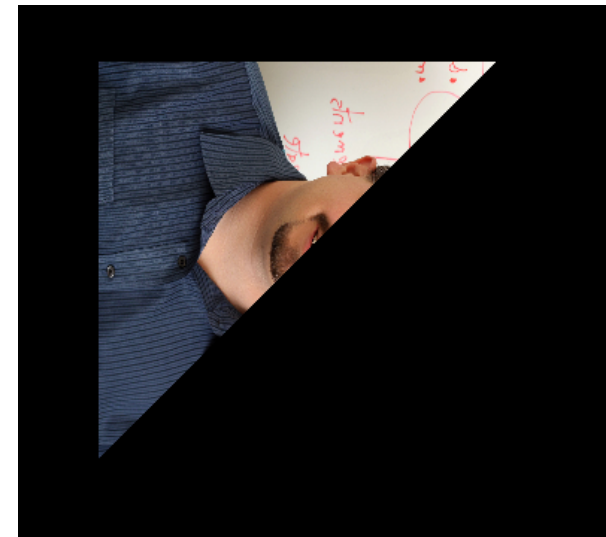
```
class vtk441PolyDataMapper : public vtkOpenGLPolyDataMapper
{
public:
    static vtk441PolyDataMapper *New();

    virtual void RenderPiece(vtkRenderer *ren, vtkActor *act)
    {
        vtkJPEGReader *rdr = vtkJPEGReader::New();
        rdr->SetFileName("HankChilds_345.jpg");
        rdr->Update();
        vtkImageData *img = rdr->GetOutput();
        int dims[3];
        img->GetDimensions(dims);
        unsigned char *buffer = (unsigned char *) img->GetScalarPointer(0,0,0);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, dims[0], dims[1], 0, GL_RGB,
                    GL_UNSIGNED_BYTE, buffer);
        glEnable(GL_COLOR_MATERIAL);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

        glEnable(GL_TEXTURE_2D);
        float ambient[3] = { 1, 1, 1 };
        glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ambient);
        glBegin(GL_TRIANGLES);
        glTexCoord2f(0,0);
        glVertex3f(0,0,0);
        glTexCoord2f(1, 0);
        glVertex3f(0,1,0);
        glTexCoord2f(1., 1.);
        glVertex3f(1,1,0);
        glEnd();
    }
};
```



What do we expect the output to be?



GL_TEXTURE_MIN_FILTER / GL_TEXTURE_MAG_FILTER



- Minifying: texture bigger than triangle.
 - How to map multiple texture elements onto a pixel?
 - GL_NEAREST: pick closest texture
 - GL_LINEAR: average neighboring textures
- Magnifying (GL_TEXTURE_MAG_FILTER): triangle bigger than texture
 - How to map single texture element onto multiple pixels?
 - GL_NEAREST: no interpolation
 - GL_LINEAR: interpolate with neighboring textures

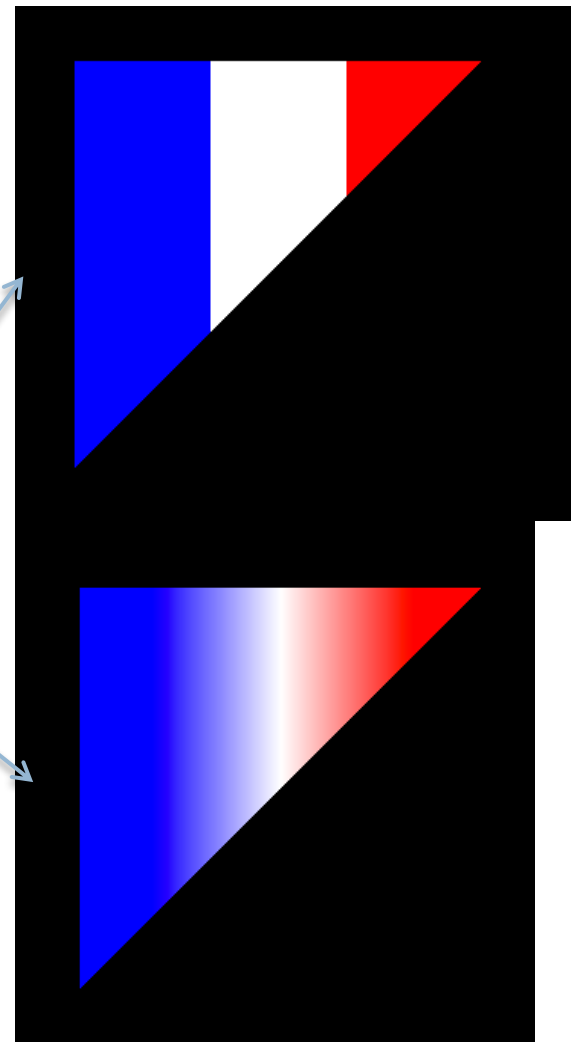
GL_TEXTURE_MAG_FILTER with NEAREST and LINEAR



```
class vtk441PolyDataMapper : public vtkOpenGLPolyDataMapper
{
public:
    static vtk441PolyDataMapper *New();

    virtual void RenderPiece(vtkRenderer *ren, vtkActor *act)
    {
        GLubyte Texture3[9] = {
            0, 0, 255, // blue
            255, 255, 255, // white
            255, 0, 0, // red
        };
        glTexImage1D(GL_TEXTURE_1D, 0, GL_RGB, 3, 0, GL_RGB, GL_UNSIGNED_BYTE,
Texture3);
        glEnable(GL_COLOR_MATERIAL);
        glTexParameterf(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        -----
        glTexParameterf(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
        --- OR ---
        glTexParameterf(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        -----
        glTexParameterf(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

        glEnable(GL_TEXTURE_1D);
        float ambient[3] = { 1, 1, 1 };
        glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ambient);
        glBegin(GL_TRIANGLES);
        glTexCoord1f(0.);
        glVertex3f(0,0,0);
        glTexCoord1f(0.0);
        glVertex3f(0,1,0);
        glTexCoord1f(1.);
        glVertex3f(1,1,0);
        glEnd();
    }
};
```



2D Texture Program



```
virtual void RenderPiece(vtkRenderer *ren, vtkActor *act)
{
    vtkJPEGReader *rdr = vtkJPEGReader::New();
    rdr->SetFileName("HankChilds_345.jpg");
    rdr->Update();
    vtkImageData *img = rdr->GetOutput();
    int dims[3];
    img->GetDimensions(dims);
    unsigned char *buffer = (unsigned char *) img->GetScalarPointer(0,0,0);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, dims[0], dims[1], 0,
                GL_UNSIGNED_BYTE, buffer);
    glEnable(GL_COLOR_MATERIAL);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

    glEnable(GL_TEXTURE_2D);
    float ambient[3] = { 1, 1, 1 };
    glMaterialfv(GL_FRONT, GL_AMBIENT, ambient);
    glBegin(GL_QUADS);
    glTexCoord2f(0, 0);
    glVertex3f(1.1, 0, 0);
    glTexCoord2f(1, 0);
    glVertex3f(0.1, 0, 0);
    glTexCoord2f(1, 1);
    glVertex3f(0.1, 1, 0);
    glTexCoord2f(0, 1);
    glVertex3f(1.1, 1, 0);
    glEnd();
}
```

Texture is not 1:1, should probably scale geometry.

This is a terrible program ... why?



glBindTexture: tell the GPU about the texture once and re-use it!



```
class vtk441PolyDataMapper : public vtkOpenGLPolyDataMapper
{
public:
    static vtk441PolyDataMapper *New();

    GLuint texture;
    bool initialized;

    vtk441PolyDataMapper()
    {
        initialized = false;
    }
    void SetUpTexture()
    {
        glGenTextures(1, &texture);
        glBindTexture(GL_TEXTURE_2D, texture);

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

        vtkJPEGReader *rdr = vtkJPEGReader::New();
        rdr->SetFileName("HankChilds_345.jpg");
        rdr->Update();
        vtkImageData *img = rdr->GetOutput();
        int dims[3];
        img->GetDimensions(dims);
        unsigned char *buffer = (unsigned char *) img->GetScalarPointer(0,0,0);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, dims[0], dims[1], 0, GL_RGB,
                     GL_UNSIGNED_BYTE, buffer);
        initialized = true;
    }
};
```

```
virtual void RenderPiece(vtkRenderer *ren, vtkActor *act)
{
    if (!initialized)
        SetUpTexture();
    glEnable(GL_COLOR_MATERIAL);

    glBindTexture(GL_TEXTURE_2D, texture);
    glEnable(GL_TEXTURE_2D);
    float ambient[3] = { 1, 1, 1 };
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ambient);
    glBegin(GL_TRIANGLES);
    glTexCoord2f(0,0);
    glVertex3f(0,0,0);
    glTexCoord2f(1, 0);
    glVertex3f(0,1,0);
    glTexCoord2f(1., 1.);
    glVertex3f(1,1,0);

    glTexCoord2f(1., 1.);
    glVertex3f(1.1,1,0);
    glTexCoord2f(0,1);
    glVertex3f(1.1,0,0);
    glTexCoord2f(0, 0);
    glVertex3f(0.1,0,0);

    glEnd();
};
```

More on Geometric Primitives



Geometric Specification: glBegin



Name

`glBegin` — delimit the vertices of a primitive or a group of like primitives

C Specification

```
void glBegin(GLenum mode);
```

Parameters

mode

Specifies the primitive or primitives that will be created from vertices presented between `glBegin` and the subsequent [glEnd](#). Ten symbolic constants are accepted: `GL_POINTS`, `GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_QUADS`, `GL_QUAD_STRIP`, and `GL_POLYGON`.

C Specification

```
void glEnd( void);
```

Description

`glBegin` and [glEnd](#) delimit the vertices that define a primitive or a group of like primitives. `glBegin` accepts a single argument that specifies in which of ten ways the vertices are interpreted. Taking `n` as an integer count starting at one, and `N` as the total number of vertices specified, the interpretations are as follows:

Geometric Primitives



GL_POINTS

Treats each vertex as a single point. Vertex n defines point n . N points are drawn.

GL_LINES

Treats each pair of vertices as an independent line segment. Vertices $2 \times n - 1$ and $2 \times n$ define line n . $N/2$ lines are drawn.

GL_LINE_STRIP

Draws a connected group of line segments from the first vertex to the last. Vertices n and $n + 1$ define line n . $N - 1$ lines are drawn.

GL_LINE_LOOP

Draws a connected group of line segments from the first vertex to the last, then back to the first. Vertices n and $n + 1$ define line n . The last line, however, is defined by vertices N and 1 . N lines are drawn.

Geometric Primitives



GL_TRIANGLES

Treats each triplet of vertices as an independent triangle. Vertices $3 \times n - 2$, $3 \times n - 1$, and $3 \times n$ define triangle n . $N/3$ triangles are drawn.

GL_TRIANGLE_STRIP

Draws a connected group of triangles. One triangle is defined for each vertex presented after the first two vertices. For odd n , vertices n , $n + 1$, and $n + 2$ define triangle n . For even n , vertices $n + 1$, n , and $n + 2$ define triangle n . $N - 2$ triangles are drawn.

GL_TRIANGLE_FAN

Draws a connected group of triangles. One triangle is defined for each vertex presented after the first two vertices. Vertices 1 , $n + 1$, and $n + 2$ define triangle n . $N - 2$ triangles are drawn.

Geometric Primitives



GL_QUADS

Treats each group of four vertices as an independent quadrilateral. Vertices $4 \times n - 3$, $4 \times n - 2$, $4 \times n - 1$, and $4 \times n$ define quadrilateral n . $N / 4$ quadrilaterals are drawn.

GL_QUAD_STRIP

Draws a connected group of quadrilaterals. One quadrilateral is defined for each pair of vertices presented after the first pair. Vertices $2 \times n - 1$, $2 \times n$, $2 \times n + 2$, and $2 \times n + 1$ define quadrilateral n . $N / 2 - 1$ quadrilaterals are drawn. Note that the order in which vertices are used to construct a quadrilateral from strip data is different from that used with independent data.

GL_POLYGON

Draws a single, convex polygon. Vertices 1 through N define this polygon.

What can go inside a glBegin?



Only a subset of GL commands can be used between `glBegin` and `glEnd`. The commands are `glVertex`, `glColor`, `glSecondaryColor`, `glIndex`, `glNormal`, `glFogCoord`, `glTexCoord`, `glMultiTexCoord`, `glVertexAttrib`, `glEvalCoord`, `glEvalPoint`, `glArrayElement`, `glMaterial`, and `glEdgeFlag`. Also, it is acceptable to use `glCallList` or `glCallLists` to execute display lists that include only the preceding commands. If any other GL command is executed between `glBegin` and `glEnd`, the error flag is set and the command is ignored.

Lighting Model



Lighting



- `glEnable(GL_LIGHTING);`
 - Tells OpenGL you want to have lighting.
- Eight lights
 - Enable and disable individually
 - `glEnable(GL_LIGHT0)`
 - `glDisable(GL_LIGHT7)`
 - Set attributes individually
 - `glLightfv(GL_LIGHTi, ARGUMENT, VALUES)`

glLightfv parameters



□ Ten parameters (ones you will use):

GL_AMBIENT

params contains four fixed-point or floating-point values that specify the ambient RGBA intensity of the light. Both fixed-point and floating-point values are mapped directly. Neither fixed-point nor floating-point values are clamped. The initial ambient light intensity is (0, 0, 0, 1).

GL_DIFFUSE

params contains four fixed-point or floating-point values that specify the diffuse RGBA intensity of the light. Both fixed-point and floating-point values are mapped directly. Neither fixed-point nor floating-point values are clamped. The initial value for GL_LIGHT0 is (1, 1, 1, 1). For other lights, the initial value is (0, 0, 0, 0).

GL_SPECULAR

params contains four fixed-point or floating-point values that specify the specular RGBA intensity of the light. Both fixed-point and floating-point values are mapped directly. Neither fixed-point nor floating-point values are clamped. The initial value for GL_LIGHT0 is (1, 1, 1, 1). For other lights, the initial value is (0, 0, 0, 0).

GL_POSITION

params contains four fixed-point or floating-point values that specify the position of the light in homogeneous object coordinates. Both fixed-point and floating-point values are mapped directly. Neither fixed-point nor floating-point values are clamped.

The position is transformed by the modelview matrix when `glLight` is called (just as if it were a point), and it is stored in eye coordinates. If the w component of the position is 0, the light is treated as a directional source. Diffuse and specular lighting calculations take the light's direction, but not its actual position, into account, and attenuation is disabled. Otherwise, diffuse and specular lighting calculations are based on the actual location of the light in eye coordinates, and attenuation is enabled. The initial position is (0, 0, 1, 0); thus, the initial light source is directional, parallel to, and in the direction of the - z axis.

glLightfv in action



For each light source, we can set an RGBA for the diffuse, specular, and ambient components:

```
glEnable(GL_LIGHTING);
```

```
glEnable(GL_LIGHT0);
```

```
GLfloat diffuse0[4] = { 0.7, 0.7, 0.7, 1 };
```

```
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse0);
```

```
... // set ambient, specular, position
```

```
glDisable(GL_LIGHT1); // do we need to do this?
```

```
...
```

```
glDisable(GL_LIGHT7); // do we need to do this?
```

How do we tell OpenGL about the surface normals?



- Flat shading:

```
glNormal3f(0, 0.707, -0.707);
```

```
glVertex3f(0, 0, 0);
```

```
glVertex3f(1, 1, 0);
```

```
glVertex3f(1, 0, 0);
```

- Smooth shading:

```
glNormal3f(0, 0.707, -0.707);
```

```
glVertex3f(0, 0, 0);
```

```
glNormal3f(0, 0.707, +0.707);
```

```
glVertex3f(1, 1, 0);
```

```
glNormal3f(1, 0, 0);
```

```
glVertex3f(1, 0, 0);
```



Distance and Direction

- The source colors are specified in RGBA
- The position is given in homogeneous coordinates
 - If $w = 1.0$, we are specifying a finite location
 - If $w = 0.0$, we are specifying a parallel source with the given direction vector

glLightfv parameters (2)



□ Ten parameters (ones you will never use)

GL_SPOT_DIRECTION

params contains three fixed-point or floating-point values that specify the direction of the light in homogeneous object coordinates. Both fixed-point and floating-point values are mapped directly. Neither fixed-point nor floating-point values are clamped.

The spot direction is transformed by the upper 3x3 of the modelview matrix when `glLight` is called, and it is stored in eye coordinates. It is significant only when `GL_SPOT_CUTOFF` is not 180, which it is initially. The initial direction is (0, 0, -1).

GL_SPOT_EXPONENT

params is a single fixed-point or floating-point value that specifies the intensity distribution of the light. Fixed-point and floating-point values are mapped directly. Only values in the range [0, 128] are accepted.

Effective light intensity is attenuated by the cosine of the angle between the direction of the light and the direction from the light to the vertex being lighted, raised to the power of the spot exponent. Thus, higher spot exponents result in a more focused light source, regardless of the spot cutoff angle (see `GL_SPOT_CUTOFF`, next paragraph). The initial spot exponent is 0, resulting in uniform light distribution.

GL_SPOT_CUTOFF

params is a single fixed-point or floating-point value that specifies the maximum spread angle of a light source. Fixed-point and floating-point values are mapped directly. Only values in the range [0, 90] and the special value 180 are accepted. If the angle between the direction of the light and the direction from the light to the vertex being lighted is greater than the spot cutoff angle, the light is completely masked. Otherwise, its intensity is controlled by the spot exponent and the attenuation factors. The initial spot cutoff is 180, resulting in uniform light distribution.

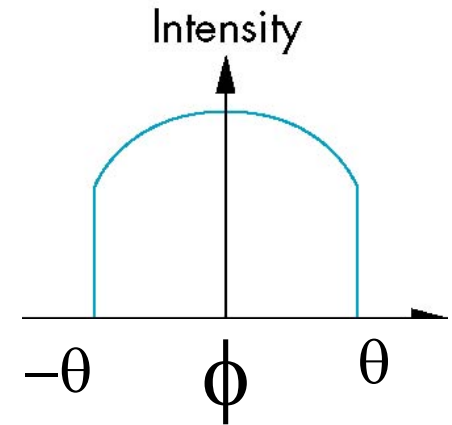
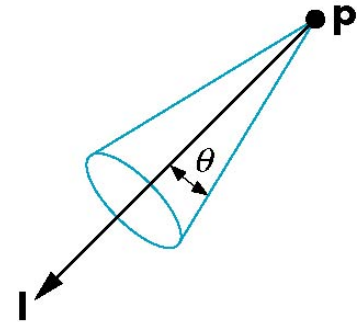
GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION, GL_QUADRATIC_ATTENUATION

params is a single fixed-point or floating-point value that specifies one of the three light attenuation factors. Fixed-point and floating-point values are mapped directly. Only nonnegative values are accepted. If the light is positional, rather than directional, its intensity is attenuated by the reciprocal of the sum of the constant factor, the linear factor times the distance between the light and the vertex being lighted, and the quadratic factor times the square of the same distance. The initial attenuation factors are (1, 0, 0), resulting in no attenuation.



Spotlights

- Use `glLightv` to set
 - Direction `GL_SPOT_DIRECTION`
 - Cutoff `GL_SPOT_CUTOFF`
 - Attenuation `GL_SPOT_EXPONENT`
 - Proportional to $\cos^\alpha \phi$



What happens with multiple lights?



```
glEnable(GL_LIGHT0);
```

```
glEnable(GL_LIGHT1);
```

- → the effects of these lights are additive.
 - Individual shading factors are added and combined
 - Effect is to make objects brighter and brighter
 - Same as handling of high specular factors for 1E



Global Ambient Light

- Ambient light depends on color of light sources
 - A red light in a white room will cause a red ambient term that disappears when the light is turned off
- OpenGL also allows a global ambient term that is often helpful for testing
 - `glLightModelfv(GL_LIGHT_MODEL_AMBIENT, global_ambient)`
 - **VTK turns this on by default!**
 - **Affects lighting of materials colored with `glColor`, but not `glTexCoord1f`!!**

Shading Model





Polygonal Shading

- Shading calculations are done for each vertex
 - Vertex colors become vertex shades
- By default, vertex shades are interpolated across the polygon
 - `glShadeModel (GL_SMOOTH) ;`
- If we use `glShadeModel (GL_FLAT) ;` the color at the first vertex will determine the shade of the whole polygon
 - We will come back to this in a few slides



Polygon Normals

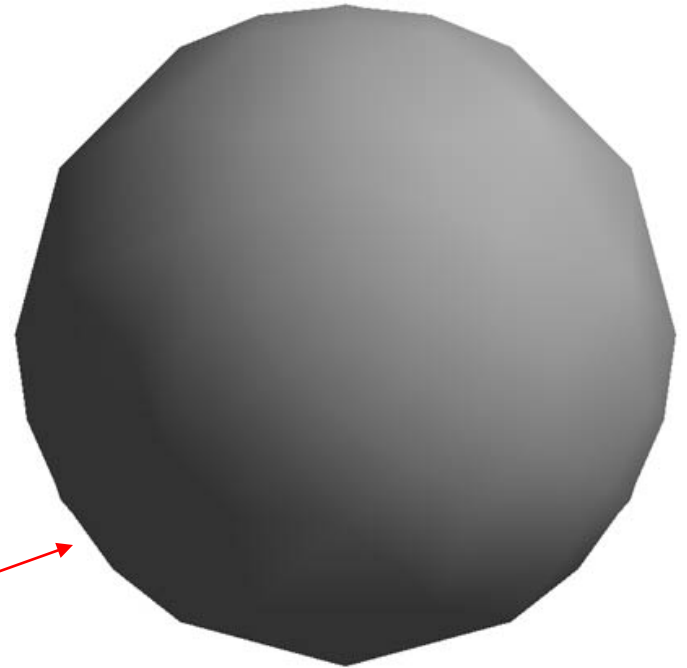
- Polygons have a single normal
 - Shades at the vertices as computed by the Phong model can be almost same
 - Identical for a distant viewer (default) or if there is no specular component
- Consider model of sphere
- Different normals at each vertex, want single normal





Smooth Shading

- We can set a new normal at each vertex
- Easy for sphere model
 - If centered at origin $\mathbf{n} = \mathbf{p}$
- Now smooth shading works
- Note *silhouette edge*

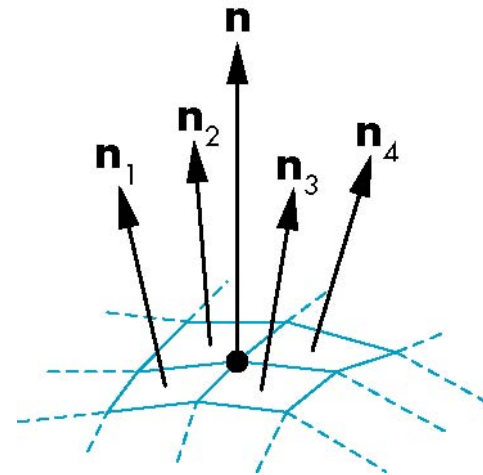




Mesh Shading

- The previous example is not general because we knew the normal at each vertex analytically
- For polygonal models, Gouraud proposed we use the average of the normals around a mesh vertex

$$\mathbf{n} = (\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4) / |\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4|$$



glShadeModel



glShadeModel — select flat or smooth shading

C Specification

```
void glShadeModel(GLenum mode);
```

Parameters

mode

Specifies a symbolic value representing a shading technique. Accepted values are GL_FLAT and GL_SMOOTH. The initial value is GL_SMOOTH.

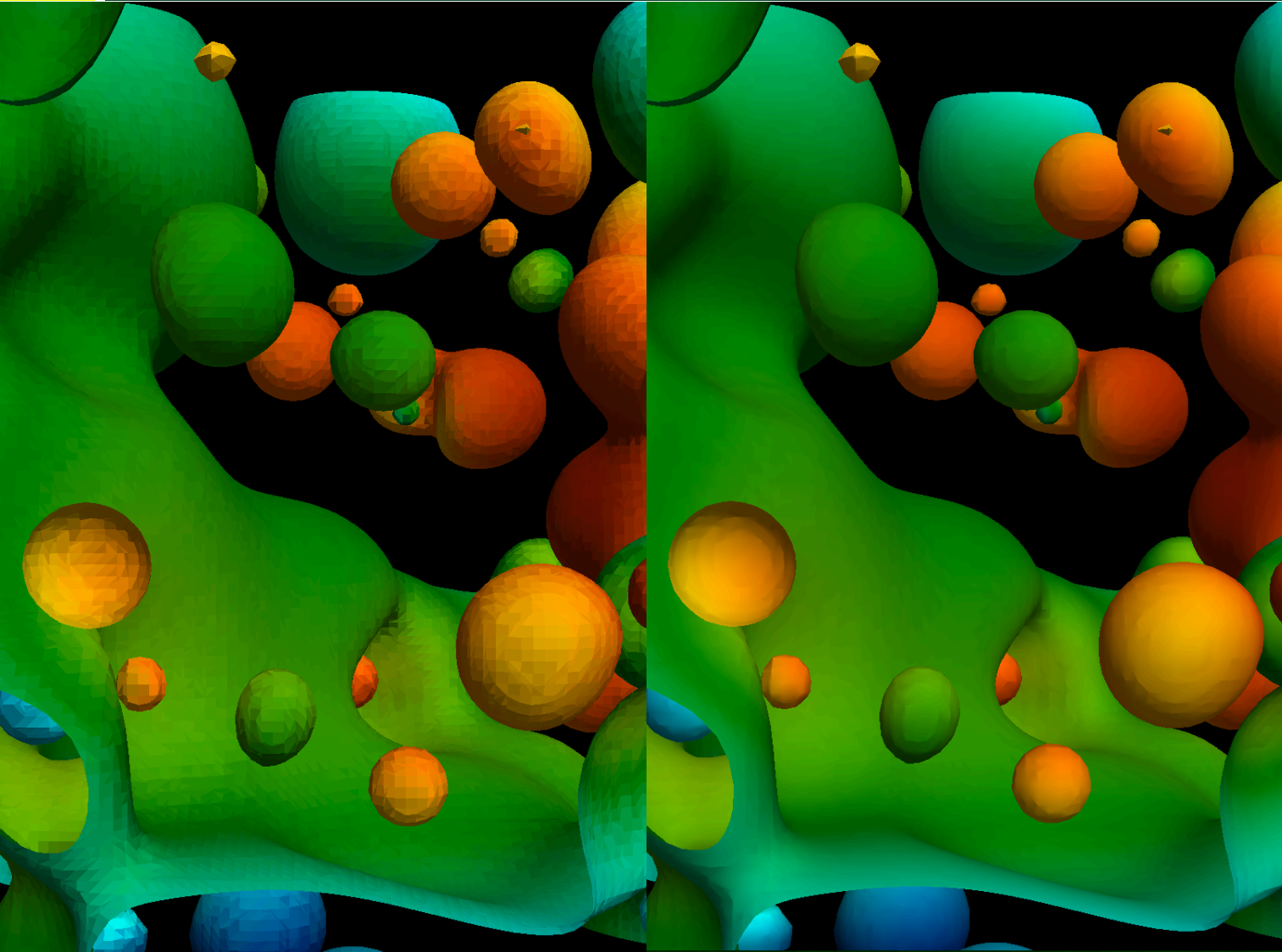
Description

GL primitives can have either flat or smooth shading. Smooth shading, the default, causes the computed colors of vertices to be interpolated as the primitive is rasterized, typically assigning different colors to each resulting pixel fragment. Flat shading selects the computed color of just one vertex and assigns it to all the pixel fragments generated by rasterizing a single primitive. In either case, the computed color of a vertex is the result of lighting if lighting is enabled, or it is the current color at the time the vertex was specified if lighting is disabled.

Flat and smooth shading are indistinguishable for points. Starting when [glBegin](#) is issued and counting vertices and primitives from 1, the GL gives each flat-shaded line segment i the computed color of vertex $i + 1$, its second vertex. Counting similarly from 1, the GL gives each flat-shaded polygon the computed color of the vertex listed in the following table. This is the last vertex to specify the polygon in all cases except single polygons, where the first vertex specifies the flat-shaded color.

Primitive Type of Polygon i	Vertex
Single polygon ($i == 1$)	1
Triangle strip	$i + 2$
Triangle fan	$i + 2$
Independent triangle	$3 \boxtimes i$
Quad strip	$2 \boxtimes i + 2$
Independent quad	$4 \boxtimes i$

glShadeModel



glShadeModel
affects normals
and colors

Material Model



glMaterial: coarser controls for color



- ❑ You specify how much light is reflected for the material type.
- ❑ Command:
`glMaterialfv(FACE_TYPE, PARAMETER, VALUE(S))`
- ❑ `FACE_TYPE =`
 - `GL_FRONT_AND_BACK`
 - ~~`GL_FRONT`~~
 - ~~`GL_BACK`~~

(We will talk about this later)

glMaterialfv Parameters



GL_AMBIENT

params contains four fixed-point or floating-point values that specify the ambient RGBA reflectance of the material. The values are not clamped. The initial ambient reflectance is (0.2, 0.2, 0.2, 1.0).

GL_DIFFUSE

params contains four fixed-point or floating-point values that specify the diffuse RGBA reflectance of the material. The values are not clamped. The initial diffuse reflectance is (0.8, 0.8, 0.8, 1.0).

GL_SPECULAR

params contains four fixed-point or floating-point values that specify the specular RGBA reflectance of the material. The values are not clamped. The initial specular reflectance is (0, 0, 0, 1).

glMaterialfv Parameters



GL_EMISSION

params contains four fixed-point or floating-point values that specify the RGBA emitted light intensity of the material. The values are not clamped. The initial emission intensity is (0, 0, 0, 1).

GL_SHININESS

params is a single fixed-point or floating-point value that specifies the RGBA specular exponent of the material. Description Only values in the range [0, 128] are accepted. The initial specular exponent is 0.

GL_AMBIENT_AND_DIFFUSE

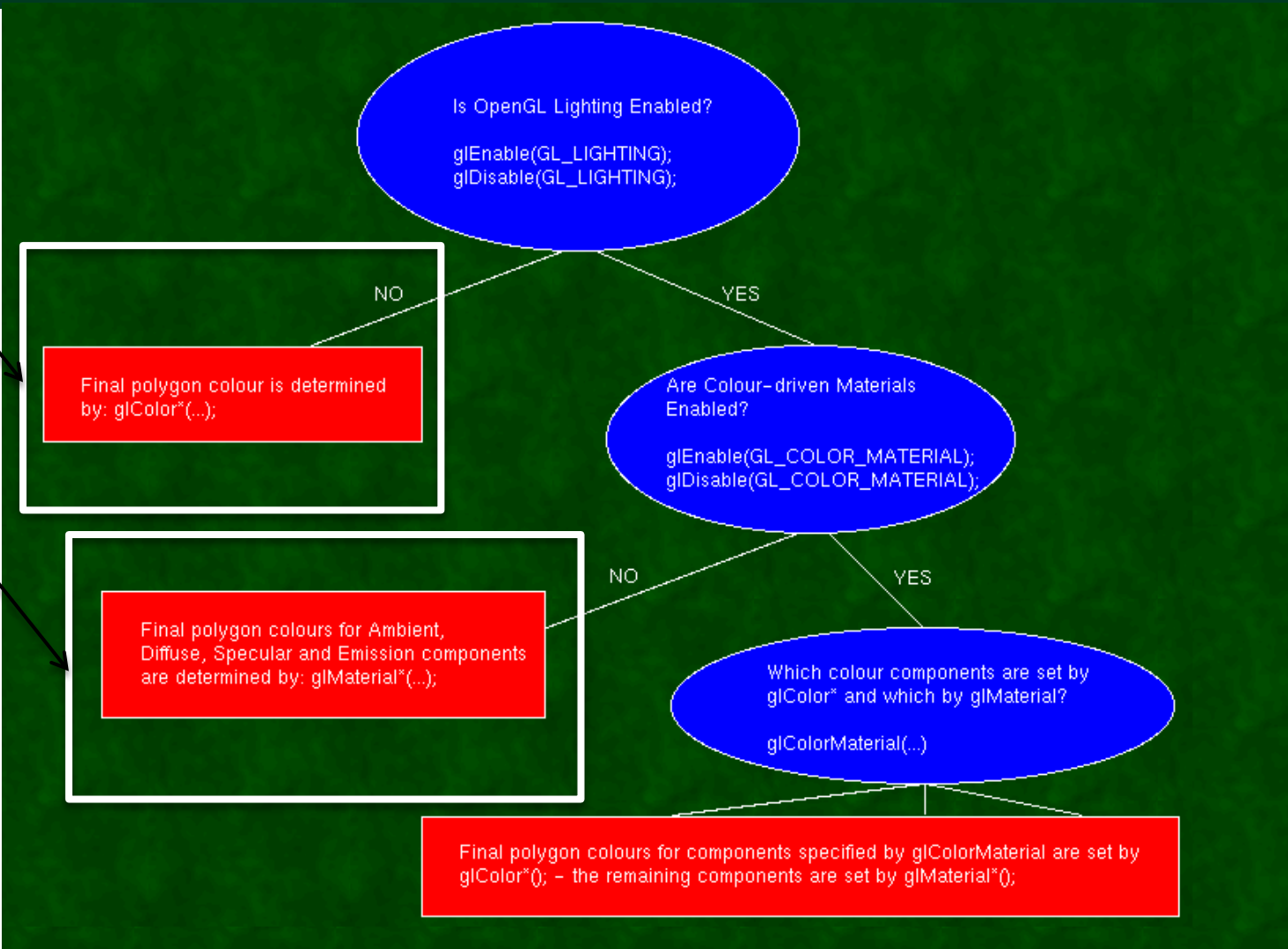
Equivalent to calling `glMaterial` twice with the same parameter values, once with `GL_AMBIENT` and once with `GL_DIFFUSE`.

OpenGL: very complex model for lighting and colors



glMaterial
not used

glColor
is no-op



Basic OpenGL light model



- The OpenGL light model presumes that the light that reaches your eye from the polygon surface arrives by four different mechanisms:
 - AMBIENT
 - DIFFUSE
 - SPECULAR
 - EMISSION - in this case, the light is actually emitted by the polygon - equally in all directions.

Difference between lights and materials



- There are three light colours for each light:
 - Ambient, Diffuse and Specular (set with `glLight`)
- There are four colors for each surface
 - Same three + Emission(set with `glMaterial`).
- All OpenGL implementations support at least eight light sources - and the `glMaterial` can be changed at will for each polygon

Interactions between lights and materials



- The final polygon colour is the sum of all four light components, each of which is formed by multiplying the `glMaterial` colour by the `glLight` colour (modified by the directionality in the case of Diffuse and Specular).
- Since there is no Emission colour for the `glLight`, that is added to the final colour without modification.

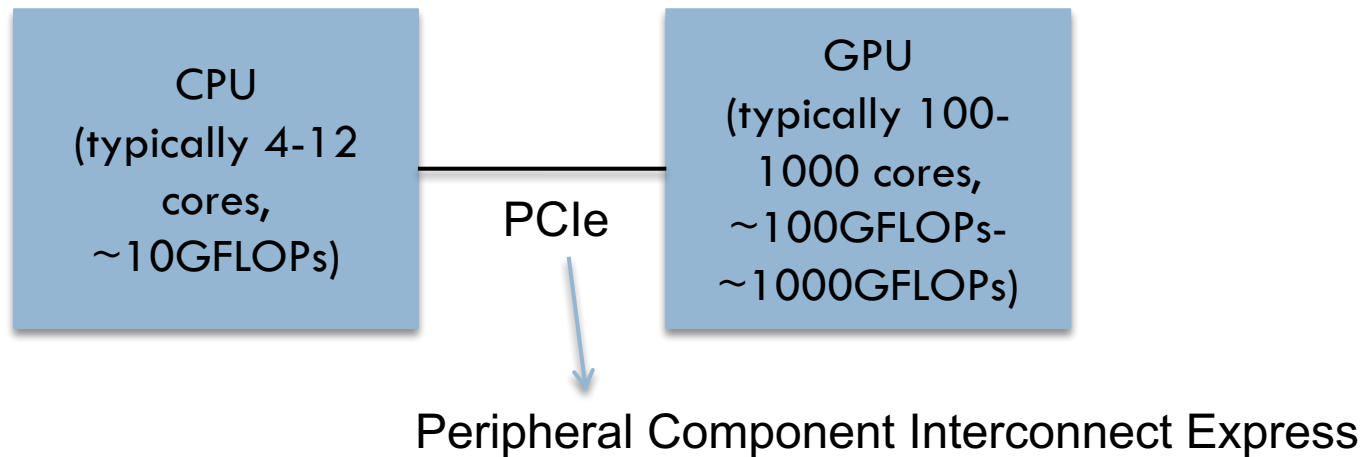
Display Lists



CPU and GPU



- Most common configuration has CPU and GPU on separate dies
 - I.e., plug GPU in CPU



What are the performance ramifications of this architecture?

Display lists



□ Idea:

- send geometry and settings to GPU once, give it an identifier
- GPU stores geometry and settings
- Just pass the identifier for every subsequent render



Display lists

- Generate an identifier:

```
GLuint displayList = glGenLists(1);
```

- Tell GPU that all subsequent geometry is part of the list:

```
glNewList(displayList, GL_COMPILE);
```

- Specify geometry (i.e., glVertex, etc)

- Tell GPU we are done specifying geometry:

```
glEndList();
```

- Later on, tell GPU to render all the geometry and settings associated with our list:

```
glCallList(displayList);
```

Display lists in action



```
for (int frame = 0 ; frame < nFrames ; frame++)  
{  
    SetCamera(frame, nFrames);  
    glBegin(GL_TRIANGLES);  
    for (int i = 0 ; i < triangles.size() ; i++)  
    {  
        for (int j = 0 ; j < 3 ; j++)  
        {  
            glColor3ubv(triangles[i].colors[j]);  
            glColor3fv(triangles[i].vertices[j]);  
        }  
    }  
    glEnd();  
}
```



```
GLuint displayList = glGenLists(1);  
glNewList(displayList, GL_COMPILE);  
glBegin(GL_TRIANGLES);  
for (int i = 0 ; i < triangles.size() ; i++)  
{  
    for (int j = 0 ; j < 3 ; j++)  
    {  
        glColor3ubv(triangles[i].colors[j]);  
        glColor3fv(triangles[i].vertices[j]);  
    }  
}  
glEnd();  
glEndList();  
  
for (int frame = 0 ; frame < nFrames ; frame++)  
{  
    SetCamera(frame, nFrames);  
    glCallList(displayList);  
}
```

What are the performance ramifications between the two?

glNewList



- GL_COMPILE
 - Make the display list for later use.
- GL_COMPILE_AND_EXECUTE
 - Make the display list and also execute it as you go.