Final Project Timeline

- Panel is coming quickly:
  - Tues @ 10:15
  - Colloquium Room (220 Deschutes)

- Grades:
  - At panel: 100%
  - By 4pm Thurs: 90% (new)
  - By 2pm Fri: 80%
  - After: 60%

- All projects must be presented to me
Panel

- My prediction: enough people will take the 90% option that all people who want to present to the panel can.
- Do you have to present to the panel?: no.
  - 100% option available prior to panel, and immediately following panel.
- Do you have to attend the panel?: yes.
  - -3% of grade for non-attendance.
Outline

- Shaders
- level-of-detail
Outline

- Shaders
- level-of-detail
Shaders

- Shader: computer program used to do “shading”
- “Shading”: general term that covers more than just shading/lighting
  - Used for many special effects
- Increased control over:
  - position, hue, saturation, brightness, contrast
- For:
  - pixels, vertices, textures
Motivation: Bump Mapping

- **Idea:**
  - typical rasterization, calculate fragments
  - fragments have normals (as per usual)
  - also interpolate texture on geometry & fragments
    - use texture for “bumps”
    - take normal for fragment and displace it by “bump” from texture

[Images showing bump mapping effectiveness]
Bump Mapping Example

Concept

BumpMapping allows designers to express their creativity through a 100,000+ polygons creature. Once art is done, a low poly model (5000 polygons) is automatically generated along with a normal map.

At runtime, details are added back by combining the low model with the normal map.

credit: http://www.fabiensanglard.net/bumpMapping/
Results

Bump Mapping Example
How to do Bump Mapping?

- Answer: easy to imagine doing it in your Project 1A-1F infrastructure
  - You have total control

- But what OpenGL commands would do this?
  - Not possible in V1 of the GL interface, which is what we have learned

- It is possible with various extensions to OpenGL
  - We will learn to do this with shaders
Shading Languages

- shading language: programming language for graphics, specifically shader effects
- Benefits: increased flexibility with rendering
- OpenGL (as we know it so far): fixed transformations for color, position, of pixels, vertices, and textures.
- Shader languages: custom programs, custom effects for color, position of pixels, vertices, and textures.
ARB assembly language

- ARB: low-level shading language
  - at same level as assembly language
- Created by OpenGL Architecture Review Board (ARB)
- Goal: standardize instructions for controlling GPU
- Implemented as a series of extensions to OpenGL
- You don’t want to work at this level, but it was an important development in terms of establishing foundation for today’s technology
GLSL: OpenGL Shading Language

- GLSL: high-level shading language
  - also called GLSLang
  - syntax similar to C

- Purpose: increased control of graphics pipeline for developers, but easier than assembly
  - This is layer where developers do things like “bump mapping”

- Benefits:
  - Benefits of GL (cross platform: Windows, Mac, Linux)
  - Support over GPUs (NVIDIA, ATI)
  - HW vendors support GLSL very well
Other high-level shading languages

- Cg (C for Graphics)
  - based on C programming language
  - outputs DirectX or OpenGL shader programs
  - deprecated in 2012

- HLSL (high-level shading language)
  - used with MicroSoft Direct3D
  - analogous to GLSL
  - similar to CG

- RSL (Renderman Shading Language)
  - C-like syntax
  - for use with Renderman: Pixar’s rendering engine
Relationship between GLSL and OpenGL

Versions [edit]

GLSL versions have evolved alongside specific versions of the OpenGL API. It is only with OpenGL versions 3.3 and above that the GLSL and OpenGL major and minor version numbers match. These versions for GLSL and OpenGL are related in the following table:

<table>
<thead>
<tr>
<th>GLSL Version</th>
<th>OpenGL Version</th>
<th>Date</th>
<th>Shader Preprocessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.10.59 [1]</td>
<td>2.0</td>
<td>April 2004</td>
<td>#version 110</td>
</tr>
<tr>
<td>1.20.8 [2]</td>
<td>2.1</td>
<td>September 2006</td>
<td>#version 120</td>
</tr>
<tr>
<td>1.30.10 [3]</td>
<td>3.0</td>
<td>August 2008</td>
<td>#version 130</td>
</tr>
<tr>
<td>1.40.08 [4]</td>
<td>3.1</td>
<td>March 2009</td>
<td>#version 140</td>
</tr>
<tr>
<td>1.50.11 [5]</td>
<td>3.2</td>
<td>August 2009</td>
<td>#version 150</td>
</tr>
<tr>
<td>3.30.6 [6]</td>
<td>3.3</td>
<td>February 2010</td>
<td>#version 330</td>
</tr>
<tr>
<td>4.00.9 [7]</td>
<td>4.0</td>
<td>March 2010</td>
<td>#version 400</td>
</tr>
<tr>
<td>4.10.6 [8]</td>
<td>4.1</td>
<td>July 2010</td>
<td>#version 410</td>
</tr>
<tr>
<td>4.20.11 [9]</td>
<td>4.2</td>
<td>August 2011</td>
<td>#version 420</td>
</tr>
<tr>
<td>4.30.8 [10]</td>
<td>4.3</td>
<td>August 2012</td>
<td>#version 430</td>
</tr>
<tr>
<td>4.50 [12]</td>
<td>4.5</td>
<td>August 2014</td>
<td>#version 450</td>
</tr>
</tbody>
</table>

Source: wikipedia
4 Types of Shaders

- Vertex Shaders
- Fragment Shaders
- Geometry Shaders
- Tessellation Shaders

- It is common to use multiple types of shaders in a program and have them interact.
How Shaders Fit Into the Graphics Pipeline

- You can 0 or 1 of each shader type
- Vertex & fragment: very common
- Geometry & tessellation: less common
  - adaptive meshing

Transform Vertices from World Space to Device Space

Vertex shaders: custom implementation

Rasterize

Fragments to Buffers

Fragment shaders: custom implementation

Geometry & tessellation shaders: create new geometry before rasterized
Vertex Shader

- Run once for each vertex
- Can: manipulate position, color, texture
- Cannot: create new vertices
- Primary purpose: transform from world-space to device-space (+ depth for z-buffer).
  - However: A vertex shader replaces the transformation, texture coordinate generation and lighting parts of OpenGL, and it also adds texture access at the vertex level
- Output goes to geometry shader or rasterizer
Geometry Shader

- Run once for each geometry primitive
- Purpose: create new geometry from existing geometry.
- Output goes to rasterizer
- Examples: glyphing, mesh complexity modification
- Formally available in GL 3.2, but previously available in 2.0+ with extensions

- Tessellation Shader: doing some of the same things
- Available in GL 4.0
Fragment Shader

- Run once for each fragment
- Purpose: replaces the OpenGL 1.4 fixed-function texturing, color sum and fog stages
- Output goes to buffers
- Example usages: bump mapping, shadows, specular highlights
- Can be very complicated: can sample surrounding pixels and use their values (blur, edge detection)
- Also called pixel shaders
How to Use Shaders

- You write a shader program: a tiny C-like program
- You write C/C++ code for your application
- Your application loads the shader program from a text file
- Your application sends the shader program to the OpenGL library and directs the OpenGL library to compile the shader program
- If successful, the resulting GPU code can be attached to your (running) application and used
- It will then supplant the built-in GL operations
How to Use Shaders: Visual Version

Project2B’ C++ code → g++ → Project2B’ binary

OpenGL library

Program is used on GPU to support Project2B’ binary

shader program

sends “char *” version of program to GL via function call

OpenGL compiles program, binary made just for the current execution

shader program is a binary

reads text file when running
Compiling Shader

GLuint vertexShader = glGetUniformLocation(GL_VERTEX_SHADER);
std::string vertexProgram = loadFileToString("vs.glsl");
const char *vertex_shader_source = vertexProgram.c_str();
GLint const vertex_shader_length = strlen(vertex_shader_source);
glShaderSource(vertexShader, 1, &vertex_shader_source, &vertex_shader_length);
glCompileShader(vertexShader);
GLint isCompiledVS = 0;
gGetShaderiv(vertexShader, GL_COMPILE_STATUS, &isCompiledVS);
if(isCompiledVS == GL_FALSE)
{
    cerr << "Did not compile VS" << endl;

    GLint maxLength = 0;
    glGetShaderiv(vertexShader, GL_INFO_LOG_LENGTH, &maxLength);

    // The maxLength includes the NULL character
    std::vector<GLchar> errorLog(maxLength);
    glGetShaderInfoLog(vertexShader, maxLength, &maxLength, &errorLog[0]);
    cerr << "Vertex shader log says " << &errorLog[0] << endl;
    exit(EXIT_FAILURE);
}
Compiling Multiple Shaders

```cpp
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
std::string vertexProgram = loadFileToString("vs.glsl");
const char *vertex_shader_source = vertexProgram.c_str();
GLint const vertex_shader_length = strlen(vertex_shader_source);
glShaderSource(vertexShader, 1, &vertex_shader_source, &vertex_shader_length);
glCompileShader(vertexShader);
GLint isCompiledVS = 0;
gGetShaderiv(vertexShader, GL_COMPILE_STATUS, &isCompiledVS);

if(isCompiledVS == GL_FALSE)
{
    cerr << "Did not compile VS" << endl;

    GLint maxLength = 0;
gGetShaderiv(vertexShader, GL_INFO_LOG_LENGTH, &maxLength);

    // The maxLength includes the NULL character
    std::vector<GLchar> errorLog(maxLength);
gGetShaderInfoLog(vertexShader, maxLength, &maxLength, &errorLog[0]);
cerr << "Vertex shader log says " << &errorLog[0] << endl;
exit(EXIT_FAILURE);
}

GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
std::string fragmentProgram = loadFileToString("fs.glsl");
const char *fragment_shader_source = fragmentProgram.c_str();
GLint const fragment_shader_length = strlen(fragment_shader_source);
glShaderSource(fragmentShader, 1, &fragment_shader_source, &fragment_shader_length);
glCompileShader(fragmentShader);
GLint isCompiledFS = 0;
gGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &isCompiledFS);
```
Attaching Shaders to a Program

```c
GLuint program = glCreateProgram();
glAttachShader(program, vertexShader);
glAttachShader(program, fragmentShader);

glLinkProgram(program);

glDetachShader(program, vertexShader);
glDetachShader(program, fragmentShader);
```
GLint isLinked = 0;
gGetProgramiv(program, GL_LINK_STATUS, (int *)&isLinked);
if(isLinked == GL_FALSE)
{
    GLint maxLength = 0;
    glGetProgramiv(program, GL_INFO_LOG_LENGTH, &maxLength);

    //The maxLength includes the NULL character
    std::vector<GLchar> infoLog(maxLength);
    glGetProgramInfoLog(program, maxLength, &maxLength, &infoLog[0]);
    cerr << "Couldn't link" << endl;
    cerr << "Log says " << &(infoLog[0]) << endl;

    exit(EXIT_FAILURE);
}
BUT: this doesn’t work in VTK...

- VTK has its own shader handling, and it doesn’t play well with the GL calls above...

```cpp
vtkSmartPointer<vtkShaderProgram2> pgm = vtkShaderProgram2::New();
pgm->SetContext(renWin);

vtkSmartPointer<vtkShader2> vertexShader=vtkShader2::New();
vertexShader->SetType(VTK_SHADER_TYPE_VERTEX);
std::string vertexProgram = loadFileToString("v_vs.glsl");
vertexShader->SetSourceCode(vertexProgram.c_str());
vertexShader->SetContext(pgm->GetContext());

pgm->GetShaders()->AddItem(vertexShader);

vtkSmartPointer<vtkShader2> fragmentShader=vtkShader2::New();
fragmentShader->SetType(VTK_SHADER_TYPE_FRAGMENT);
std::string fragmentProgram = loadFileToString("v_fs.glsl");
fragmentShader->SetSourceCode(fragmentProgram.c_str());
fragmentShader->SetContext(pgm->GetContext());

pgm->GetShaders()->AddItem(fragmentShader);

((vtkOpenGLProperty*)win3Actor->GetProperty())->SetPropProgram(pgm);
```

note: VTK6.1 much better for shaders than 6.0
Simplest Vertex Shader

```c
void main(void)
{
  gl_Position = gl_ModelViewProjectionMatrix*gl_Vertex;
}
```

Many built-in variables.
Some are input.
Some are required output (gl_Position).
VTK uses special names
  propFuncVS: vertex shader
  propFuncFS: fragment shader
somehow it changes these into “main” just in time…
Bump-mapping with GLSL

bump map texture

output
Will need to load a texture...

```c
// from swiftless.com
GLuint LoadTexture( const char * filename, int width, int height )
{
    GLuint texture;
    unsigned char * data;
    FILE * file;

    //The following code will read in our RAW file
    file = fopen( filename, "rb" );

    if ( file == NULL ) return 0;
    data = (unsigned char *)malloc( width * height * 3 );
    fread( data, width * height * 3, 1, file );

    fclose( file );

    glGenTextures( 1, &texture ); //generate the texture with the loaded data
    glBindTexture( GL_TEXTURE_2D, texture ); //bind the texture to it’s array

    glTexParameteri( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE ); //set texture environment parameters

    //And if you go and use extensions, you can use Anisotropic filtering textures which are of an
    //even better quality, but this will do for now.
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    //Here we are setting the parameter to repeat the texture instead of clamping the texture
    //to the edge of our shape.
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );

    //Generate the texture
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);

    free( data ); //free the texture
    return texture; //return whether it was successfull
}```
Need to put 2D textures on our triangles...

```cpp
class Triangle {
  public:
    double X[3];
    double Y[3];
    double Z[3];
    double Tu[3];
    double Tv[3];
};

void DrawSphere()
{
  int recursionLevel = 3;
  Triangle t;
  t.X[0] = 1;
  t.Y[0] = 0;
  t.Z[0] = 0;
  t.Tu[0] = 0;
  t.Tv[0] = 0;
  t.X[1] = 0;
  t.Y[1] = 1;
  t.Z[1] = 0;
  t.Tu[1] = 1;
  t.Tv[1] = 0;
  t.X[2] = 0;
  t.Y[2] = 0;
  t.Tu[2] = 1;
  t.Tv[2] = 1;
  std::vector<Triangle> list;
  list.push_back(t);
  for (int r = 0; r < recursionLevel; r++)
  {
    list = SplitTriangle(list);
  }
  std::vector<Triangle> output(4*list.size());
  for (unsigned int i = 0; i < list.size(); i++)
  {
    double mid1[5], mid2[5], mid3[5];
    mid1[0] = (list[i].X[0]+list[i].X[1])/2;
    mid1[1] = (list[i].Y[0]+list[i].Y[1])/2;
    mid1[2] = (list[i].Z[0]+list[i].Z[1])/2;
    mid1[3] = (list[i].Tu[0]+list[i].Tu[1])/2;
    mid1[4] = (list[i].Tv[0]+list[i].Tv[1])/2;
    mid2[0] = (list[i].X[1]+list[i].X[2])/2;
    mid2[3] = (list[i].Tu[1]+list[i].Tu[2])/2;
    mid2[4] = (list[i].Tv[1]+list[i].Tv[2])/2;
    mid3[0] = (list[i].X[0]+list[i].X[2])/2;
    mid3[1] = (list[i].Y[0]+list[i].Y[2])/2;
    mid3[2] = (list[i].Z[0]+list[i].Z[2])/2;
    mid3[3] = (list[i].Tu[0]+list[i].Tu[2])/2;
    mid3[4] = (list[i].Tv[0]+list[i].Tv[2])/2;
    output[4*i+0].X[0] = list[i].X[0];
    output[4*i+0].Y[0] = list[i].Y[0];
    output[4*i+0].Z[0] = list[i].Z[0];
    output[4*i+0].Tu[0] = list[i].Tu[0];
    output[4*i+0].Tv[0] = list[i].Tv[0];
    output[4*i+0].X[1] = mid1[0];
    output[4*i+0].Y[1] = mid1[1];
    output[4*i+0].Z[1] = mid1[2];
    output[4*i+0].Tu[1] = mid1[3];
    output[4*i+0].Tv[1] = mid1[4];
    output[4*i+0].X[2] = mid3[0];
    output[4*i+0].Y[2] = mid3[1];
    output[4*i+0].Z[2] = mid3[2];
    output[4*i+0].Tu[2] = mid3[3];
    output[4*i+0].Tv[2] = mid3[4];
    output[4*i+1].X[0] = list[i].X[1];
    output[4*i+1].Y[0] = list[i].Y[1];
    output[4*i+1].Z[0] = list[i].Z[1];
    output[4*i+1].Tu[0] = list[i].Tu[1];
    output[4*i+1].Tv[0] = list[i].Tv[1];
  }
}
Need to set up shaders and textures...

```cpp
vtkSmartPointer<vtkShaderProgram2> pgm = vtkShaderProgram2::New();
pgm->SetContext(renWin);

vtkSmartPointer<vtkShader2> vertexShader=vtkShader2::New();
vertexShader->SetType(VTK_SHADER_TYPE_VERTEX);
//std::string vertexProgram = loadFileToString("vs.glsl");
std::string vertexProgram = loadFileToString("v_vs.glsl");
vertexShader->SetSourceCode(vertexProgram.c_str());
vertexShader->SetContext(pgm->GetContext());

pgm->GetShaders()->AddItem(vertexShader);

vtkSmartPointer<vtkShader2> fragmentShader=vtkShader2::New();
fragmentShader->SetType(VTK_SHADER_TYPE_FRAGMENT);
//std::string fragmentProgram = loadFileToString("light_fs.glsl");
std::string fragmentProgram = loadFileToString("v_fs.glsl");
fragmentShader->SetSourceCode(fragmentProgram.c_str());
fragmentShader->SetContext(pgm->GetContext());

pgm->GetShaders()->AddItem(fragmentShader);

((vtkOpenGLProperty*)win3Actor->GetProperty())->SetPropProgram(pgm);
win3Actor->GetProperty()->ShadingOn();

GLuint texture = LoadTexture("normal_map.raw", 256, 256);
glEnable(GL_TEXTURE_2D);
int texture_location = glGetUniformLocation(fragmentShader->GetId(), "color_texture");
glUniform1i(texture_location, 0);
glBindTexture(GL_TEXTURE_2D, texture);
```
So what is the vertex shader program?...

```c
void propFuncVS(void)
{
    gl_TexCoord[0] = gl_MultiTexCoord0;

    // Set the position of the current vertex
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```
And what is the fragment shader program...

uniform sampler2D color_texture;
uniform sampler2D normal_texture;

void propFuncFS(void)
{
    // Extract the normal from the normal map
    vec3 normal = normalize(texture2D(normal_texture, gl_TexCoord[0].st).rgb * 2.0 - 1.0);

    // Determine where the light is positioned (this can be set however you like)
    vec3 light_pos = normalize(vec3(1.0, 1.0, 1.5));

    // Calculate the lighting diffuse value
    float diffuse = max(dot(normal, light_pos), 0.0);

    vec3 color = diffuse * texture2D(color_texture, gl_TexCoord[0].st).rgb;

    // Set the output color of our current pixel
    gl_FragColor = vec4(color, 1.0);
}
Outline

- Shaders
- level-of-detail
Mipmaps

- Mipmaps: pre-calculated, optimized collections of images that accompany a main texture, intended to increase rendering speed and reduce aliasing artifacts.
- Widely used in 3D computer games, flight simulators and other 3D imaging systems.
- In use, it is called “mipmappings.”
- The letters "MIP" in the name are an acronym of the Latin phrase multum in parvo, meaning "much in little".
Mipmaps
Level of detail (LOD) techniques

- **level of detail**: decreasing the complexity of some 3D object representations, because they
  - are far away
  - are moving fast
  - are not important

- increases the efficiency of rendering by decreasing the workload on graphics pipeline stages
  - reduced visual quality of the model is often unnoticed because of the small effect on object appearance when distant or moving fast.
Types of LOD

- Two types:
  - Discrete LoD (DLoD)
  - Continuous LoD (CLoD)
Discrete LoD (DLoD)

- Make a fixed amount of models, ranging from highest quality to coarse approximation & render appropriate one based on importance factor
- Fastest in practice, but leads to “popping”
## Discrete LoD example

<table>
<thead>
<tr>
<th>Rendered images</th>
<th>Brute</th>
<th>DLOD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Render time</td>
<td>27.27 ms</td>
<td>1.29 ms</td>
</tr>
<tr>
<td>Scene vertices (thousands)</td>
<td>2328.48</td>
<td>109.44</td>
</tr>
</tbody>
</table>
OK, how do we create coarse versions?

- How do we take [image] and make these?

- Answer: surface decimation
Decimation of Triangle Meshes

Paper by W.J. Schroeder et al.

Presented by Guangfeng Ji
Goal

- Reduce the total number of triangles in a triangle mesh
- Preserve the original topology and a good approximation of the original geometry
Overview

- A multiple-pass algorithm

- During each pass, perform the following three basic steps on every vertex:
  - Classify the local geometry and topology for this given vertex
  - Use the decimation criterion to decide if the vertex can be deleted
  - If the point is deleted, re-triangulate the resulting hole.

- This vertex removal process repeats, with possible adjustment of the decimation criteria, until some termination condition is met.
Feature Edge

A feature edge exists if the angle between the surface normals of two adjacent triangles is greater than a user-specified “feature angle”.
Characterize Local Geometry and Topology

- Each vertex is assigned one of five possible classifications:
  - Simple vertex
  - Complex vertex
  - Boundary vertex
  - Interior edge vertex
  - Corner vertex
Evaluate the Decimation Criteria

- Complex vertices are not deleted from the mesh.
- Use the distance to plane criterion for simple vertices.
- Use the distance to edge criterion for boundary and interior edge vertices.
- Corner vertex?
Criterion for Simple Vertices

- Use the distance to plane criterion.
- If the vertex is within the specified distance to the average plane, it can be deleted. Otherwise, it is retained.
Overview

- A multiple-pass algorithm

- During each pass, perform the following three basic steps on every vertex:
  - Classify the local geometry and topology for this given vertex
  - Use the decimation criterion to decide if the vertex can be deleted
  - If the point is deleted, re-triangulate the resulting hole.

- This vertex removal process repeats, with possible adjustment of the decimation criteria, until some termination condition is met.
Continuous LoD (CLoD)

- considers the polygon mesh being rendered as a function which must be evaluated requiring to avoid excessive errors which are a function of some heuristic (usually distance) themselves.

- The given "mesh" function is then continuously evaluated and an optimized version is produced according to a tradeoff between visual quality and performance.