About me

- PhD student in Computerized Image Analysis
- Develops methods and tools for interactive analysis of medical 3D (volume) images
Graphics programming

Typically deals with

- How to define a 3D scene with a virtual camera, 3D objects, light sources, textures, etc
- How to create a 2D projection of the 3D scene so that it can be displayed on a computer monitor
Real-time vs. offline rendering

- **Real-time rendering:**
  - used in games and other interactive applications
  - scene must be updated at 30-60 frames per second
  - speed > image quality

- **Offline rendering:**
  - used for visual effects, Pixar movies, etc
  - rendering a single frame may take several hours!
  - image quality > speed

Image sources: id Software, Pixar
Polygon meshes

- Used to represent graphical 3D models
- Collections of **vertices**, **edges**, and **faces**
- The faces of a mesh can be arbitrary polygons, but are usually triangles or quadrilaterals

The Stanford Bunny
Textures

- Allow us to add details without adding lots of polygons

Model and textures: http://www.turbosquid.com

Original mesh

Same mesh with shading and textures

Model and textures: http://www.turbosquid.com
Shaders

- In real-time rendering, a *shader* is a small program that is compiled on the CPU and executed on the GPU.
- The most common usage of shaders is to apply transformations on *vertices* and compute the level of light and color of *fragments* (pixel candidates).
- By using different shaders or shader inputs, we can drastically change the visual appearance of a 3D object.

Image source: http://www.realtimerendering.com
A trip through the programmable graphics pipeline

- Vertices
- Vertex shader
- Clipper and primitive assembler
- Rasterizer
- Fragments
- Fragment shader
Vertex input

Vertices

○

○ ○
Vertex shader
Clipping and primitive assembly
Rasterization

Vertices → Vertex shader → Clipper and primitive assembler → Rasterizer → Fragments
Fragment shader

Vertices → Clipper and primitive assembler → Rasterizer → Fragments

- Vertex shader
- Clipper and primitive assembler
- Rasterizer
- Fragment shader
OpenGL

- Cross-platform, low-level API for rendering 2D and 3D graphics
- First version released in 1992. The specification is currently managed by the Khronos Group.
- Used in games, simulations, scientific visualization, CAD, mobile applications, virtual reality, etc
- State-based
- Only deals with rendering, not windowing or input
- Callable from most programming languages (C, C++, Python, Java, C#, JavaScript, Rust, Haskell, ...)

Khronos Group
Flavors

• OpenGL
  - For desktop applications

• OpenGL ES
  - For embedded systems (smartphones, tablets, etc)

• WebGL
  - Allows you to render 3D graphics in a web browser. Based on OpenGL ES.
No, you don't have to learn all this...
OpenGL functions

• Executed in the host CPU application

• Responsible for:
  - creating and initializing buffers, shaders, textures, etc
  - uploading data to GPU-accessible memory
  - configuring render state
  - submitting draw calls
  - clearing and swapping buffers
OpenGL utility libraries

• Since OpenGL is only concerned about rendering, we need to use various utility libraries in our applications

• We will use the following utility libraries in the labs:
  - FreeGLUT (used for creating and managing windows)
  - GLEW (OpenGL extension loader)
  - GLM (mathematics library)
  - AntTweakBar (widget for tweaking parameters)
The OpenGL Utility Toolkit (GLUT)

• Simple and portable library that provides an interface between the graphics system and the operating and window system

• Allows the user to
  - create a window for displaying OpenGL graphics
  - execute the rendering loop
  - set up mouse and keyboard interaction

• The original GLUT library hasn't been maintained since 1998...
  - we'll use the more recent FreeGLUT library instead

• Other options: GLFW, SDL, Qt
#include <GL/glut.h>

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glutSwapBuffers();
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitWindowSize(500, 500);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);
    glutCreateWindow("Empty window");
    glutDisplayFunc(&display);
    glutMainLoop();
    return 0;
}
The OpenGL Extension Wrangler Library (GLEW)

- Cross-platform open-source C/C++ library for loading OpenGL extensions
- Will search for and pull in all the OpenGL extensions that you need (or at least the ones that are supported)
- After installing GLEW, you can use it by including GL/glew.h in your application
OpenGL Mathematics (GLM) library

- A header-only C++ mathematics library for graphics programming
- Provides vector and matrix datatypes, transforms, quaternions, noise functions, and much more
- Similar syntax as GLSL
- Used in the host (CPU) C++ application only, not in the GLSL* shader!
- Link: http://glm.g-truc.net/

*GLSL has its own built-in math functions and datatypes
AntTweakBar

- Minimal GUI library for tweaking rendering parameters
- We'll use it in Assignment 3 and the project
Vertex data

- Each vertex in a graphical model has one or several different **attributes**, e.g.,
  - Position
  - Color
  - Normal vector
  - Texture coordinate

- Vertex data is loaded/generated on the CPU and uploaded to GPU-accessible memory via **buffer objects**
Surface normals

- A surface normal is a vector that describes the local orientation of a surface at some vertex or face. Important for shading.
Vertex buffer objects (VBOs)

- Used for uploading arrays of vertex data/attributes to GPU memory
- Before submitting a draw call, we must bind the VBO(s)
- We can use a separate VBO for each vertex attribute, or (usually more efficient) interleave several vertex attributes in a single VBO

Separate VBOs

- Positions
- Normals
- Texture coordinates

Interleaved attributes in a single VBO
Vertex array object (VAOs)

- A VAO stores references to one or several VBOs, along with their states and configurations
  - At drawing, you only have to bind the VAO
  - Simplifies code: you don't have to bind and configure the VBOs separately at each draw call
- Recent OpenGL core profiles require that you use VAOs
- See https://www.opengl.org/wiki/Vertex_Specification for more details about VBOs and VAOs
The OpenGL Shading Language (GLSL)

- A high-level, cross-platform shading language for real-time rendering
- Based on the C programming language. Uses similar naming conventions, data types, and control structures.
- Built-in:
  - vector and matrix types (vec2, vec3, mat4, ...)
  - various math and utility functions for graphics programming (exp, max, dot, cross, normalize, mix, smootstep, ...)
  - texture lookup functions
Shader types

- OpenGL supports five types of shaders:
  - **Vertex** shaders
  - **Fragment** shaders
  - **Geometry** shaders
  - **Tesselation** shaders
  - **Compute** shaders

- We will only cover vertex and fragment shaders in this course
The vertex shader

- Applies (among other things) colors and transformations on the input vertices
- Passes varying data to the fragment shader
The vertex shader (cont.)

Vertex displacement
The fragment shader

- Takes uniforms and interpolated data from the vertex shader and rasterizer as input
- Computes the final color of fragments (pixel candidates) by, e.g., evaluating lighting equations
The fragment shader (cont.)

Per-fragment lighting
GLSL source files

- GLSL source code is usually stored in plain ASCII text files.
- We'll use the suffixes *.vert for vertex shaders and *.frag for fragment shaders.
- At program startup, the host application loads the GLSL source files into strings and compile them into shaders that can be executed on the GPU.
Creating, compiling, and linking GLSL shader programs

- Main steps:
  1) Read vertex and fragment shader source files into strings
  2) Create vertex shader object from the vertex shader source string
  3) Create fragment shader object from the fragment shader source string
  4) Create a program object and attach the vertex and fragment shader objects to it
  5) Compile the program
  6) Link the compiled program and check for errors
  7) Deattach and delete the shader objects
Example – Simple triangle
GLSL vertex shader (triangle.vert)

```glsl
#version 130 // specifies the GLSL version
in vec4 a_position; // input vertex position

void main() {
  // just sets the output vertex position
  // to the input vertex position
  gl_Position = a_position;
}
```
GLSL fragment shader
(triangle.frag)

```glsl
#version 130

void main() {
    // sets the output fragment color to white
    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}
```
Variables for communicating with shaders

- Three types of variables:
  - *Attribute* variables
  - *Varying* variables
  - *Uniform* variables
Attribute variables

- A GLSL vertex shader can access vertex attributes via the `in` qualifier. Example:

```glsl
#version 130

in vec4 a_position; // input vertex position
in vec3 a_color; // input vertex color

out vec3 v_color; // output vertex color

void main() {
    v_color = a_color;
    gl_Position = a_position;
}
```
Varying variables

- Provide an interface for passing data (color, normals, texture coordinates, etc) between the vertex shader and the fragment shader.
- By default, varying data will be linearly interpolated over the geometric primitive.
Varying variable defined in the vertex shader

- The vertex shader uses the `out` qualifier to pass varying data to the fragment shader

```glsl
#version 130

in vec4 a_position; // input vertex position
in vec3 a_color; // input vertex color

out vec3 v_color; // output vertex color

void main() {
    v_color = a_color;
    gl_Position = a_position;
}
```
Varying data accessed in the fragment shader

- The fragment shader accesses the data via the `in` qualifier

```glsl
#version 130

in vec3 v_color; // interpolated vertex color

void main() {
  gl_FragColor = vec4(v_color, 1.0);
}
```
Uniform variables

- Data that should be constant for all vertices and fragments can be passed to the shaders as *uniforms*.

- Examples of data commonly passed as uniforms:
  - transforms
  - material properties (color, shininess, opacity, etc)
  - light sources
  - texture samplers
  - time
  - flags for enabling/disabling parts of the shading
Uniform variables

- Can be accessed in both the vertex shader and the fragment shader via the `uniform` qualifier

```glsl
#version 130

uniform vec3 u_color;

void main() {
    // sets the output fragment color to the color defined by the uniform variable
    gl_FragColor = vec4(u_color, 1.0);
}
```
glDrawArrays and glDrawElements

- Draw commands for rendering graphics primitives (points, lines, triangles) from array data stored in VBOs or VAOs

```c
void drawTriangle(GLuint program, GLuint vao) {
    glUseProgram(program);
    glBindVertexArray(vao);
    glDrawArrays(GL_TRIANGLES, 0, 3);
    glBindVertexArray(0);
    glUseProgram(0);
}
```
OpenGL primitives

- The following geometric primitives are the basic building blocks in OpenGL applications:
  - Point sprites: \texttt{GL_POINTS}
  - Lines: \texttt{GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP}
  - Triangles: \texttt{GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN}
Coordinate systems

- The OpenGL graphics pipeline defines six different coordinate systems:
  - *Object* (or model) coordinates
  - *World* coordinates
  - *Eye* (or camera) coordinates
  - *Clip* coordinates
  - *Normalized device* coordinates
  - *Window* (or screen) coordinates

- Stefan will talk more about them next week
Transforms

- Used to manipulate the position, orientation, size, and shape of objects in 2D/3D scenes
- We also use transforms to define our virtual camera and go from one coordinate system to another
- Usually represented as 3-by-3 or 4-by-4 matrices
- Examples of basic transforms:
  - Translation
  - Rotation
  - Scaling
- Stefan will cover this topic
Transforms in shader-based OpenGL

- Typically, we construct our transforms in the host C++ program (using, e.g., GLM) and pass them to the vertex shader as uniform variables.
- On the GPU side, the vertex shader applies the transforms on the incoming vertex data.

```
#version 130

uniform mat4 u_transform;
in vec4 a_position; // input vertex position

void main() {
    gl_Position = u_transform * a_position;
}
```
Summary

- The host CPU application loads/creates vertex data, shaders, textures, etc, uploads them to GPU-accessible memory, and submits draw calls.
- Shaders do all the visual magic! Vertices and fragments are processed in parallel on the GPU.
OpenGL web resources

• Good tutorials for modern shader-based OpenGL programming:
  - http://learnopengl.com/
  - http://openglbook.com/
  - http://duriansoftware.com/joe
  - http://open.gl/
  - http://tomdalling.com/blog/category/modern-opengl/

• Reference pages:
  - http://opengl.anteru.net/
Blender

- Free open-source 3D computer graphics software for 3D modelling, rendering, animation, etc.

[Image of Blender interface]

www.blender.org/features/
This is not a course about blender or 3D modelling...

but I recommend that you at least learn some of the basic features (might be useful for the project)

In blender, you can also play around with transformations, materials, lighting, etc
For fun: **Unity 5 or Unreal Engine 4**

- Free game engines. Easy to get started!
See you at the lab next week!