About me

- PhD student in Computerized Image Analysis
- I develop methods and tools for interactive analysis of medical 3D (volume) images
- Have been programming in OpenGL more or less regularly since I took this course in 2007
Graphics programming

- Graphics programming typically deals with how to define a 3D scene with a virtual camera, 3D objects, light sources, textures, etc, and create a 2D projection of this scene that can be displayed on the screen.
OpenGL

- A cross-platform, low-level API for rendering 2D and 3D graphics.
- The first version of the API was released in 1992. The specification is currently managed by the Khronos Group.
- Fairly easy to learn (at least the basics) and provides excellent rendering performance. Suitable for real-time rendering.
- Defined as a finite state machine.
- Omits windowing and input to avoid window system dependencies.
- Callable from most programming languages (C, C++, Python, Java, C#, JavaScript, Common Lisp, Haskell...)
OpenGL and object-oriented programming

- OpenGL is callable from C++, but is not object-oriented.
- You can wrap OpenGL calls inside C++ classes (very common in practice), but keep in mind that OpenGL relies on global state and that it is easy to introduce undesirable side-effects.
No, you don't have to learn all this...
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 three OpenGL utility libraries in this course:
  - FreeGLUT (used for creating and managing windows)
  - GLEW (OpenGL extension loader)
  - GLM (mathematics library)
GLUT

- The OpenGL Utility Toolkit (GLUT) is a simple and highly portable library that provides an interface between the graphics system and the underlying operating and window system.
- It allows the user to create a window for displaying OpenGL graphics, monitor input devices, create pop-up menus, etc.
- The original GLUT library hasn't been maintained since 1998, so you should use the more recent FreeGLUT library instead.
- Other options: GLFW, SDL, Qt
Creating a double-buffered window with GLUT

```c
#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;
}
```
GLEW

- The OpenGL Extension Wrangler Library (GLEW) is a cross-platform open-source C/C++ library for loading OpenGL extensions
- It will search for and pull in all the OpenGL extensions that you need (or at least the ones that are supported on your system)
- 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
- Link: http://glm.g-truc.net/
- Alternative mathematics libraries:
  - Eigen (a more heavy-weight linear algebra library)
  - CML
OpenGL primitives

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

![Image source: http://www.realtimerendering.com](http://www.realtimerendering.com)
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
Surface normals

- A surface normal is a vector that describes the local orientation of a surface at some vertex or face. Important for shading.
The programmable OpenGL graphics pipeline

- Takes vertices (and their attributes) as input and produces pixels on the screen as output
Shader programming

- In real-time rendering, a shader is a small program that is compiled on the CPU and executed on the GPU.
- Shaders perform various rendering calculations (but can also be used for more general computations).
- The most common usage of shaders is to apply transformations on vertices and compute the level of light and color of fragments (pixel candidates).
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.

• Also provides vector and matrix types (vec2, vec3, mat4, etc), along with functions that operates on them

• Supports swizzling of vectors/matrices (e.g., v.xy = v.yx)

• See http://www.opengl.org/documentation/documentation/glsl/ for more information
OpenGL shaders

- OpenGL supports five types of shaders:
  - Vertex shaders
  - Fragment shaders
  - Geometry shaders
  - Tessellation 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 and passes varying data to the fragment shader.
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
Example – Simple triangle
GLSL vertex shader
(triangle.vert)

```
#version 130 // specifies the GLSL version

in vec4 a_position; // input vertex position

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

#version 130

void main() {
    // sets the output fragment color to white
    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}

![Triangle](triangle.png)
GLSL source files

- GLSL source code is usually stored in plain ASCII text files
- It is common (but not necessary) to give these files the suffixes .vert, .frag, or .glsl
- At program startup, the host application loads the GLSL source files into strings and compiles these source strings into shaders that can be executed on the GPU.
- GLSL source code can also be embedded as strings in your .cpp or .h file (can be convenient some times)
Creating, compiling, and linking shader programs

- The (somewhat complicated) procedure for creating, compiling and linking a shader program looks like follow:

1) Read the GLSL vertex and fragment shader source files into strings
2) Create a vertex shader object from the vertex shader source string
3) Create a 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 object
6) Link the compiled program and check for errors
7) Deattach and delete the shader objects
Utility classes for working with GLSL shaders

- In the first assignment, I will provide a GLSLSourceReader class for reading GLSL source files and a GLSLProgram class for creating and representing GLSL shaders.

- By using these utility classes you can focus on more important tasks like learning how to write shaders and pass data to the GPU.

- Later on you can look at the GLSLProgram source code or some of the listed tutorials to learn the technical details.
Vertex attributes

- Each vertex in a mesh have one or several different attributes, e.g.,
  - Position
  - Color
  - Normal vector
  - Texture coordinate
- These attributes must be generated on the CPU and uploaded to the GPU memory via buffer objects
Vertex attributes

- A GLSL vertex shader can access the vertex attributes using the `in` or `attribute` qualifiers. Example:

```glsl
#version 130 // specifies the GLSL version

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

- Varying variables provides an interface to pass data (color, normals, texture coordinates, etc) between the vertex shader and the fragment shader.
- The vertex shader uses the `out` or `varying` qualifiers to pass varying data to the fragment shader.
- The fragment shader accesses the varying data via the `in` or `varying` qualifiers.
Varying variables

- By default, varying data will be linearly interpolated over the geometric primitive (line or triangle).
- You can use the `flat` keyword to specify that the varying data should not be interpolated.
Varying variable defined in the vertex shader

```glsl
#version 130 // specifies the GLSL version

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

```glsl
#version 130 // specifies the GLSL version

in vec3 v_color; // interpolated vertex color

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

• Uniform variables are used to pass the shader data that should be uniform for all vertices or fragments of the rendered object

• Uniform variables can be accessed in both the vertex shader and the fragment shader via the uniform qualifier

• They are read-only variables and cannot be defined or changed in the shaders (which means that you must define them on the CPU in your host program)

• Examples of data commonly passed as uniforms: transforms, material properties, light sources, texture samplers, time variables
Uniform variables

``` glut
#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);
}
```
Vertex buffer objects

- A vertex buffer object (VBO) is a buffer filled with arrays of vertex attributes
- You define the vertex attributes on the CPU, put them in VBOs, and upload them to the GPU memory.
- You can use a separate VBO for each vertex attribute, or (usually more efficient) interleave several vertex attributes in a single VBO

Separate VBOs

-interleave-

| Position | Normal | Texcoord |

Interleaved attributes in a single VBO
Vertex array objects

• Several VBOs can be bound to a so called vertex array object (VAO).

• This simplifies the draw calls since you just have to bind the VAO (if you don't use a VAO you have to bind the VBOs separately at each draw call)

• Some of the most recent OpenGL version require that you create at least one VAO in your OpenGL program
DrawArrays

- Renders graphics primitives (points, lines, triangles) from array data stored in VBOs or VAOs

```c
void drawTriangle(void)
{
    globals.program.enable();

    glBindVertexArray(globals.triangleVAO);
    glDrawArrays(GL_TRIANGLES, 0, 3);
    glBindVertexArray(0);
    globals.program.disable();
}
```
DrawElements

- Similar to DrawArray, but allows us to render indexed meshes where triangles (or other primitives) share vertices

```c
void drawTriangle(void)
{
    globals.program.enable();

    glBindVertexArray(globals.triangleVAO);
    glDrawElements(GL_TRIANGLES, 0, 3);
    glBindVertexArray(0);

    globals.program.disable();
}
```
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

- Pontus will tell you more about these coordinate systems in lecture 3 and 4
Transforms

- Transforms allow us to manipulate the position, orientation, size, and shape of objects in our 3D scene and define and set up cameras, light sources, etc.

- We also need transforms to change from one coordinate system to another (e.g., go from world to eye coordinates).

- Usually represented as 3-by-3 or 4-by-4 matrices.

- Examples of basic transforms:
  - Translation
  - Rotation
  - Scaling

- Pontus will tell you more about this topic.
Transforms in shader-based OpenGL

- Transformation matrices are typically constructed on the host CPU and sent to the GPU (as uniform variables), where the vertex shader applies them on the input vertices.
- There is no mathematics library in OpenGL, so you either have to implement your own (not recommended)...
- or (better) use one of the many existing third-party mathematics libraries for graphics programming (like GLM)
Blender

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

www.blender.org/features/
Blender

- This is NOT a course about blender or 3D modelling...
- but I recommend that you at least learn how to import existing 3D models into blender so that you can inspect them and convert them to different file formats (which will be useful for the labs and the project)
- In blender you can also play around with transformations, materials, lighting models, etc
Web resources

- A few tutorials that does a good job of teaching modern shader-based OpenGL programming:
  - http://open.gl/
  - http://www.arcsynthesis.org/gltut/
  - http://tomdalling.com/blog/category/modern-opengl/