In two subsequent replies to this message, I'll attach two parts of an export of the "Avatar 2" codebase, in which I did a lot of OpenGL related programming using the C# language. I'm sending this first header message in case the later messages are blocked due to attachment size or contents limits, so if you don't get two additional message with attachments soon, let me know. I'll send yet another message much later with some info on what's important in the overall codebase.

So the OpenGL related code is in a few places...

**Avatar.Plugin.ErrorSimulator**

    The main project where I've been doing the most OpenGL stuff lately is in the Avatar.Plugin.ErrorSimulator project, in the Rendering directory.

    The ErrorRenderer class (split over multiple files) handles nearly all of the high level OpenGL drawing and loading commands.  It exposes several public properties (for specifying data files and settings) and public methods (for executing the rendering).

    In the Components directory, FreeLookOpenGLControl calls the public methods in ErrorRendeer (if in Visual Studio, then right-click on this file and choose View Code).  The method HandledUserOnPaint(...) is called within an infinite loop to perform all of the rendering calls.  It calls DoRender(), which makes the calls upon ErrorRenderer.

    All of the code that loads data onto the GPU is called from the StartRender() method in ErrorRenderer.cs. Also in the ErrorRenderer class are FillProjectors(...), DrawSceneForCustom(...), and DrawCustom(), which perform the actual drawing (or they call other methods that do).

    I've written several shaders, which are in the Rendering/GLSL directory and its subdirectories.  The shader code is the .frag and .vert files, and the Object Oriented wrappers around those shaders are the .cs files with similar prefix names.

    The code in this project makes extensive use of the other libraries in this solution, including some listed below.

**MaskTracer.OpenGL:**

     This project contains a lot of helper classes for manipulating OpenGL state.

    The various parts are separated by folders. FBO contains helper classes for using FrameBuffer Objects.  GLSL has helper classes for manipulating shaders.  PBO is for pixel buffer objects.  Components has .NET UI classes that provide OpenGL contexts and render-loop support.

    The MouseKeyboardLookHandler classes provide some mouse-looking helpers.  ManagedModelEx loads models and textures onto the GPU.

**MaskTracer:**

    This is a general purpose library with a wide range of features, but not OpenGL directly.

    The Buffers folder has image manipulation classes.  The OBJFileEx class in the Model folder can load OBJ model files, which the aforementioned MangedModelEx class uses.  The ProjectionMatrixHelper class in the Cameras folder can help create OpenGL projection matrices given a camera calibration specification.

**OpenTK and OpenTK.GLControl:**

     I did not write this library.  It provides bindings to all of the raw OpenGL functions through the [OpenTK.Graphcis.OpenGL4.GL](http://opentk.graphcis.opengl4.gl/" \t "_blank) class (see Graphics/OpenGL4/GL4.cs) as static methods.  It also provides some .NET UI classes from which I derived my UI classes.

    Also, I've temporarily posted a build of that code [here](http://www.cs.unc.edu/~plincoln/tmp/Avatar2_r855_x86.zip" \t "_blank), if you want to try to run it, though it'll take some configuration when first run and some data files that aren't included.

    I'll let you know later this week when I've set up a basic framework for you to use in the repository.

    - Peter

Yinan,

    I've created some framework code that will hopefully give you a starting point when you're ready to start implementing the rendering algorithm.  It's been committed to the repository in the "trunk/ImageGeneration" directory.  It's set up using Visual Studio 2012, so you probably won't need 2010 anymore unless you want to load up the old legacy applications.

    The main program is set up to be "LowLatency."  It'll provide a control window (ControlForm) and a property grid for controlling any relevant settings you might need.  It also provides the controls for generating the OpenGL rendering window (RenderForm).  The main method is in Program.cs, and it has some code there for parsing any command line options you might want.

   The rendering library is set up to be "LowLatency.Rendering."  RenderForm from the main project uses the RenderControl class from the rendering library project to supply the OpenGL context and renderer.  The LowLatencyRenderer class should supply the important methods for performing the actual rendering needed.  It inherits from RendererBase (which I mostly copied from another research project), which provides a few helper debugging methods.

    The settings that LowLatencyRenderer uses and the main form shows are all defined in theRenderingSettings class, which is split into multiple files (RenderingSettings.cs and RenderingSettingsCore.cs).  RenderingSettingsCore.cs is an auto-generated file, which uses the T4 Text Templating script in RenderingSettingsCore.tt to write the C# code in the .cs file.  When you need more settings to expose to the user, you can modify the GetEntries() method in the .tt file and then hit save.  It'll regenerate the .cs file.  A few sample settings are shown, which use the Color, double, and an enum type.  I recommend installing the Tangible T4 editor plugin listed in the Google Doc when editing the .tt file (it'll provide intellisense and keyword highlighting).  (Note: RenderingSettingsCore.tt includes the SettingsImport.ttinclude file from the MaskTracer library, so if you want to see how the .tt script actually writes the .cs file, take a look in the .ttinclude file [e.g. the properties of the Entry class].)  This settings class handles cross-thread settings synchronization automatically.  Namely, settings changed in the GUI will not be applied until the next step of the render loop starts (look for StartOfPaint() in RenderControl.cs).

    The RenderForm is set up to run on a separate thread from the ControlForm.  As a result, any GUI related commands that pass between the forms must be invoked appropriately (or else exceptions or non-deterministic behavior might happen).  Some examples of how to do this are in the LowLatencyContext class (look for calls to methods containing the word "Invoke").  If you end up needing to supply some settings (like data files or the like) outside of the RenderingSettings class, then you'll need to use this style of cross-thread method calls.

    Both projects reference the MaskTracer (general helpers), MaskTracer.OpenCV (OpenCV library wrapper), MaskTracer.OpenGL (OpenGL helpers), NVidia.ToolsExt (a debugger helper), OpenTK (OpenGL bindings), and OpenTK.GLControl (OpenGL GUI controls) libaries, so any public classes and methods exposed by those libraries are available.

    I've put a bunch of TODO: comments in the two LowLatency\* projects, where I suspect you'll need to add things at a minimum.

    Hope this helps, and let me know if you have any questions.

    - Peter

P.S. Have you tried checking out the code?  I still haven't heard back from Murray about adding you, so I don't know if you have been yet or not.