The Open Toolkit Manual

The manual is not complete. You can find (and add) experimental pages here. Also, check the available translations.

Welcome and thanks for using the Open Toolkit library!

This manual will guide you through the necessary steps to develop a project with OpenTK. You will learn how to setup a new project, how to successfully use the tools provided by OpenTK and, finally, how to distribute your project to your end-users. You will also find information on writing performing code and maintaining cross-platform compatibility.

This manual is written in a way that allows skipping from section to section as needed, so feel free to do that; or you can read it sequentially from start to end, if you prefer. Keep in mind that you may add a comment at any point - we always try to improve the manual and high-quality feedback will help not only you but future OpenTK users, too. You can also browse the OpenTK API reference.

It is our hope that the time invested reading this book will be paid back in full. So let's get started!

Table of Contents

- Table of Contents
- Chapter 0: Learn OpenTK in 15'
- Chapter 1: Installation
  - Linux
  - Windows
  - Troubleshooting
- Chapter 2: Introduction to OpenTK
  - The DisplayDevice class
  - The GameWindow class
  - The NativeWindow class
  - Building a Windows.Forms + GLControl based application
  - Avoiding pitfalls in the managed world
- Chapter 3: OpenTK.Math
  - Half-Type
- Chapter 4: OpenTK.Graphics (OpenGL and ES)
  - The GraphicsContext class
    - Using an external OpenGL context with OpenTK
  - Textures
    - Loading a texture from disk
    - 2D Texture differences
    - S3 Texture Compression
  - Frame Buffer Objects (FBO)
  - Geometry
    - 1. The Vertex

        - 2. Geometric Primitive Types
        - 3.a Vertex Buffer Objects
        - 3.b Attribute Offsets and Strides
        - 3.c Vertex Arrays
        - 4. Vertex Array Objects
        - 5. Drawing
        - 5.b Drawing Optimizations
        - 6. OpenTK's procedural objects
    - OpenGL rendering pipeline
    - Fragment Operations
        - 01. Pixel Ownership Test
        - 02. Scissor Test
        - 03. Multisample Fragment Operations (WIP)
        - 04. Stencil Test
        - 05. Depth Test
        - 06. Occlusion Query
            - Conditional Render
        - 07. Blending
        - 08. sRGB Conversion
        - 09. Dithering
        - 10. Logical Operations
    - How to save an OpenGL rendering to disk
    - How to render text using OpenGL
- Chapter 5: OpenTK.Audio (OpenAL)
    - 1. Devices, Buffers and X-Ram
    - 2. Sources and EFX
    - 3. Context and Listener
- Chapter 6: OpenTK.Compute (OpenCL)
- Chapter 7: OpenTK.Input
- Chapter 8: Advanced Topics
    - Vertex Cache Optimizations
    - Garbage Collection Performance
    - GC & OpenGL (work in progress)
- Chapter 9: Hacking OpenTK
    - Project Structure
    - OpenTK Structure
    - Wrapper Design
- Appendix 1: Frequently asked questions
- Appendix 2: Function Reference
- Appendix 3: The project database
    - Creating a project
    - Creating a project release
- Appendix 4: Links
    - Models and Textures
    - OpenGL Books and Tutorials
    - Programming links
    - Tools & Utilities
    - Tutorials
- Appendix 5: Translations

Chapter 0: Learn OpenTK in 15'

So, you have downloaded the latest version of OpenTK - what now?

This is a short tutorial that will help you get started with OpenTK in 3 simple steps.

[Step 1: Installation]

Open the zip you downloaded and extract it to a folder of your choosing. I usually create a 'Projects' folder on my desktop or in my documents but any folder will do.

[Step 2: Use OpenTK]

Open the folder you just extracted. Inside, you will find three solutions: OpenTK.sln, Generator.sln and QuickStart.sln. The first two contain the OpenTK source code - no need to worry about them right now. The QuickStart solution is what we are interested in.

Double-click QuickStart.sln. This will launch your .Net IDE (don't have a .Net IDE? Check out MonoDevelop or Visual Studio Express).

Take a few moments to take in the contents of the QuickStart project:

- Game.cs: this contains the code for the game. Scroll to the bottom: the `Main()` method is where everything begins.
- References: click on the '+' sign to view the project references. The 'OpenTK' reference is the only one you need in order to use OpenTK.

Now, press F5 to run the project. A window with a colored triangle will show up - not very interesting, is it? Press escape to close it.

[Step 3: Play]

Now it's time to start playing with the code. This is a great way to learn OpenGL and OpenTK at the same time.

Every OpenTK game will contain 4 basic methods:

1. `OnLoad`: this is the place to load resources from disk, like images or music.
2. `OnUpdateFrame`: this is a suitable place to handle input, update object positions, run physics or AI calculations.
3. `OnRenderFrame`: this contains the code that renders your graphics. It typically begins with a call to `GL.Clear()` and ends with a call to `SwapBuffers`.
4. `OnResize`: this method is called automatically whenever your game window changes size. Fullscreen applications will typically call it only once. Windowed applications may call it more often. In most circumstances, you can simply copy & paste the code from Game.cs.

Why don't you try modifying a few things? Here are a few suggestions:

1. Change the colors of the triangle or the window background (OnLoad and OnRenderFrame methods).
2. Make the triangle change colors when you press a key (OnUpdateFrame and OnRenderFrame methods).
3. Make the triangle move across the screen. Use the arrow keys or the mouse to control its position (OnUpdateFrame and OnRenderFrame methods).
4. Use a for-loop to render many triangles arranged on a plane (OnRenderFrame method).
5. Rotate the camera so that the plane above acts as ground (OnRenderFrame method).
6. Use the keyboard and mouse to walk on the ground. Make sure you can't fall through it! (OnUpdateFrame and OnRenderFrame methods).

Some things you might find useful: `Vector2`, `Vector3`, `Vector4` and `Matrix4` classes for camera manipulations. `Mouse` and `Keyboard` properties for interaction with the mouse and keyboard, respectively. `Joysticks` property for interaction with joystick devices.

Don't be afraid to try things and see the results. OpenTK lends itself to explorative programming - even if something breaks, the library will help you pinpoint the cause of the error.

[Step: next]

There's a lot of functionality that is not visible at first glance: audio, advanced opengl, display devices, support for GUIs through GLControl... Then there's the subject of proper engine and game design, which could cover a whole book by itself.

Hopefully, you'll have gained a feel of the library by now and you'll be able to accomplish more complex tasks. You might wish to consult the complete documentation for the more advanced aspects of OpenTK and, of course, don't hesitate to post at the forums if you hit any roadblocks!

Chapter 1: Installation

[Prerequisites]

OpenTK is a managed library that targets the .Net 2.0 framework. To use it, you will need either the .Net or Mono runtime, plus device drivers for OpenGL (graphics), OpenAL (audio) and OpenCL (compute), depending on the parts of OpenTK you wish to use.

Most operating systems come with a version of the .Net runtime preinstalled, which means OpenTK is typically usable out of the box. In a few cases, you might need to install manually a version of .Net runtime (Windows) or the Mono runtime (Linux/Mac OS X/Windows). Any version equal to or newer than .Net 2.0 / Mono 2.0 will work fine. Earlier versions of Mono may also work, but are no longer supported. Earlier versions of .Net will not work.

Additionally, most recent operating systems come with OpenGL drivers preinstalled. For OpenAL and OpenCL drivers, you should refer to the website of your hardware vendors. *[todo: add links to common hardware vendors]*

Last, but not least, you will need to download the latest OpenTK release.

[Installation]

OpenTK releases are simple compressed archives. Simply extract the archive contents to a location on your disk and add *OpenTK.dll* as a reference to your project. You can find *OpenTK.dll* under the Binaries/OpenTK folder of the OpenTK archive.

Additionally, you should add *OpenTK.dll.config* to your project and instruct your IDE to copy this file to the output directory. This is necessary for your project to function under Linux and Mac OS X.

The following pages contain specific instructions for using or building OpenTK on different platforms.

Linux

Installing Mono

If you are using a recent Linux distribution, all prerequisites for OpenTK projects should be readily available: the Mono runtime and the Mono compilers. Execute "`mono --version`" and "`gmcs --version`" and check if the output looks like this:

```
$ mono --version
Mono JIT compiler version 1.2.6 (tarball)
Copyright (C) 2002-2007 Novell, Inc and Contributors. www.mono-
project.com
        TLS:           __thread
        GC:            Included Boehm (with typed GC)
        SIGSEGV:       altstack
        Notifications: epoll
        Architecture:  amd64
        Disabled:      none

$ gmcs --version
Mono C# compiler version 1.2.6.0
```

If one or both of these commands fail, you'll have to install Mono. Mono packages should be readily available through your package manager:

```
# Ubuntu and other .deb-based distributions
sudo apt-get install mono mono-gmcs
# or
su -c "apt-get install mono mono-gmcs"

# Fedora Core and .rpm-based distributions
su -c "yum install mono mono-gmcs"
```

If no Mono packages are available, or they are outdated (`mono --version` returns something less than 1.2.6), you should build Mono from source. There is a message in the support forum describing the process of building mono from source here.

Alternatively, you can find use one of the Mono binary packages on the Mono download page.

Using a binary release

Download the latest opentk-x.y.z-mono.tar.gz release from Sourceforge and untar it:

```
tar -xvf opentk-0.3.13-mono.tar.gz
```

A new opentk-x.y.z will be created with four subfolders: "Documentation", "Examples", "Libraries" and "QuickStart". Try running the examples contained in the second folder to make sure everything works alright:

```
cd opentk-0.3.13/Examples
mono Examples.exe
```

A new window will hopefully show up, listing all available examples. If not, check the troubleshooting section below.

The "Libraries" folder contains the main OpenTK assembly (OpenTK.dll) and the OpenTK.dll.config file - these are all you need to run OpenTK projects. If you are using MonoDevelop, check the "QuickStart" folder for a ready-to-use project. Last, don't forget to take a look at the release notes contained in the "Documentation" folder.

Troubleshooting

The following error has been reported on Fedora Core 8, when running Examples.exe:

```
Unhandled Exception: System.TypeInitializationException: An exception
was thrown by the type initializer for System.Windows.Forms.Form --->
System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. --->
System.TypeInitializationException: An exception was thrown by the
type initializer for System.Drawing.GDIPlus --->
System.DllNotFoundException: gdiplus.dll
  at (wrapper managed-to-native)
System.Drawing.GDIPlus:GdiplusStartup
(ulong&,System.Drawing.GdiplusStartupInput&,System.Drawing.GdiplusSta
rtupOutput&)
  at System.Drawing.GDIPlus..cctor () [0x00000] --- End of inner
exception stack trace ---
```

This is caused by a missing entry in `"/etc/mono/config"`. To correct this issue, open the aforementioned file (you must be root!), and add this line: `<dllmap dll="gdiplus.dll" target="/usr/lib/libgdiplus.so.0" />`. Now, Examples.exe should work.

Building OpenTK from source

OpenTK's build system currently uses NAnt, so you'll need to install that:

```
# Ubuntu
sudo apt-get install nant

# Debian
su -c "apt-get install nant"

# Fedora
su -c "yum install nant"
```

Once that is out of the way, untar the source release and cd to the Build folder:

```
unzip opentk-0.3.13.zip
cd opentk-0.3.13/Build
mono Build.exe mono
```

Wait a few seconds for the compilation to end, and check the "Binaries" folder that just appeared in the base OpenTK directory. To build the debug version, append "debug" so that the last command looks like:

```
mono Build.exe mono debug
```

[Add an appendix that describes how to build Mono from source, in case there is no package available]

Windows

OpenTK does not come with any installer or setup. Instead, you download the OpenTK binaries and add a reference to "OpenTK.dll" in your Visual Studio/SharpDevelop/MonoDevelop project. (Unzip the binaries first!)

OpenTK demo
To run all of the OpenTK builtin examples, the following software is required:

1. .NET2.0 or Mono 1.2.6
2. OpenAL 2.0.3

This is also the software required for an end-user running an OpenTK application. Note that OpenAL is not strictly required if the application does not use any sound.

OpenTK development
If you want to start developing applications using OpenTK, first make sure the items under "OpenTK demo" are installed, then download a compiler/IDE for .NET/mono. Here are some popular choices:

1. SharpDevelop
2. MonoDevelop (bundled in the mono installer)
3. Visual Studio Express

Setting up an OpenTK application in Visual Studio Express
It is a good idea to add "OpenTK.dll.config" to your project, and make sure the "Copy To Output Folder" (not "compile"!) is set to "Copy Always". The application will run without this on Windows, but not on Linux or Mac OS X.

Last, but not least, make sure the "Copy Local" property is set to true for the OpenTK reference, to simplify the distribution of your application.

Setting up an OpenTK application in SharpDevelop
Include the "OpenTK.dll.config" in your project, if you want it to run under Linux Mac OS X.
Visual explanation:

Troubleshooting

Most problems with running OpenTK-based Applications are related to the target platform missing the proper drivers.

OpenTK requires these components installed:

- Either Mono or .Net (not both).
- An OpenGL driver for your graphics card.
- The OpenAL driver for your Operating System.

Below are links for your convenience. Note: Many of those sites require Javascript enabled to function.

Mono

Novell (Linux, Mac & Windows) http://www.go-mono.com/mono-downloads/download.html

.Net

Microsoft (Windows)
http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=...

OpenAL

Note: It doesn't matter what brand your soundcard is, just chose the proper Operating System.

Creative Labs (Mac & Windows) http://www.openal.org/downloads.html
Direct link to download page for Windows:
http://connect.creativelabs.com/developer/Wiki/OpenAL%20Installer%20for%...
Strangesoft (Linux) http://kcat.strangesoft.net/openal.html

OpenGL

ATi (Linux, Mac & Windows) http://ati.amd.com/support/driver.html
NVIDIA (Linux & Windows) http://www.nvidia.com/Download/index.aspx?lang=en-us
Intel (Windows) http://downloadcenter.intel.com/
Intel (Linux) http://intellinuxgraphics.org/download.html
Mesa 3D (software rendering)
http://sourceforge.net/project/showfiles.php?group_id=3

If you have a laptop with an Nvidia card, you can obtain updated drivers through:
http://www.laptopvideo2go.com

Last edit of the links: March 2008

Tags for searches:
help problem error outdated trouble crash fail failure exception abort opengl openal driver ati intel nvidia

Chapter 2: Introduction to OpenTK

First of all, what **is** OpenTK?

Simply put, the Open Toolkit is a free project that allows you to use OpenGL, OpenGL|ES, OpenCL and OpenAL APIs from managed languages.

OpenTK started life as an experimental fork of the Tao framework before during the summer of 2006. It's original intention was to provide a cleaner wrapper than Tao.OpenGL, but it quickly grew in focus: right now, it provides access to various Khronos and Creative APIs and handles the necessary initialization logic for each API. As such, the Open Toolkit is most similar to projects like Tao, SlimDX, SDL or GLFW.

Unlike similar libraries, OpenTK attempts provide a consistent interface that utilizes the superior managed runtime. Instead of untyped pointers, OpenTK provides generics. Instead of plain constants, OpenTK uses strongly-typed enumerations. Instead of plain function lists, OpenTK separates functions per extension category. A common math library is integrated and directly usable by each API.

Features:

- Written in cross-platform C# and usable by all managed languages (F#, Boo, VB.Net, C++/CLI).
- Consistent, strongly-typed bindings, suitable for RAD development.
- Usable stand-alone or integrated with Windows.Forms, GTK#, WPF.
- Cross-platform binaries that are portable on .Net and Mono without recompilation.
- Wide platform support: Windows, Linux, Mac OS X, with iPhone port in progress.

The Open Toolkit is suitable for games, scientific visualizations and all kinds of software that requires advanced graphics, audio or compute capabilities. It's license makes it suitable for both free and commercial applications.

The DisplayDevice class

There are three main types of *display devices*: monitors, projectors and TV screens. OpenTK exposes all of them through the same interface: `OpenTK.DisplayDevice`.

You can use `OpenTK.DisplayDevice` to query available display devices, discover and modify their properties.

Example 1: discover available devices

```
using OpenTK;

foreach (DisplayDevice device in DisplayDevice.AvailableDisplays)
{
    Console.WriteLine(device.IsPrimary);
    Console.WriteLine(device.Bounds);
    Console.WriteLine(device.RefreshRate);
    Console.WriteLine(device.BitsPerPixel);
    foreach(DisplayResolution res in device.AvailableResolutions)
   {
      Console.WriteLine(res);
   }
}
```

Example 2: set the resolution of the first device to 800x600x32@60Hz:

```
using OpenTK;

devices[0].ChangeResolution(800, 600,  32,60);
```

Example 3: set the resolution of the second device to 800x600x32 using the preferred refresh rate:

```
using OpenTK;

devices[1].ChangeResolution(800, 600, 32, -1);
```

Example 4: restore all devices to their default settings

```
using OpenTK;

foreach (DisplayDevice device in DisplayDevice.AvailableDevices)
{
    device.RestoreResolution();
}
```

OpenTK will try to match your custom resolution to the closest supported resolution. If a specific bit depth or refresh rate is not supported, the current bit depth or refresh rate will be used. If no custom resolution matches the specified parameters, the

current `DisplayResolution` will be used. You can specify a negative number or zero to indicate that a specific parameter is of no interest: for example, specifying a refresh rate of 0 will result in the default refresh rate being used.

Note that it is **your** responsibility to call `RestoreResolution()` prior to exiting your application. Failing to call this method will result in undefined behavior.

[Todo: add information about hotplugging support]

The GameWindow class

[Describe the functionality of the GameWindow class]

The NativeWindow class

[Describe the functionality of the NativeWindow class]

Building a Windows.Forms + GLControl based application

Note 1: This tutorial is somewhat dated. For example, no "garbage" is seen anymore in the design view of Visual Studio. And the default color is beige, not black. Apart from those (minor) issues, this tutorial will get you started using OpenTK+Windows.Forms.
Note 2: If you have some spare time and want to contribute to the OpenTK project, please update the below page (by clicking the "Edit" link above when logged in) to reflect the current state of OpenTK.

This tutorial assumes familiarity with `Windows.Forms` application development in Visual Studio 2005/C#, and at least basic knowledge of OpenGL. It also assumes a top-to-bottom readthrough; it is a guide and not a reference.

To begin with, it is quite a different approach one has to take when designing a game/application using the `GLControl` in a Windows.Form compared to using the `GameWindow`. `GLControl` is more low-level than `GameWindow` so you'll have to pull the strings on for example time measurements by yourself. In `GameWindow`, you get more for free!

Just as in the GameWindow case, GLControl uses the default OpenGL driver of the system, so with the right drivers installed it will be hardware accelerated. However, with large windows it will be slower than the corresponding fullscreen GameWindow, because of how the underlying windowing system works [someone with more detailed knowledge than me may want to elaborate on this..].

If you come from a "main-loop-background" (C/SDL/Allegro etc.) when it comes to coding games, you'll have to rethink that fundamentally. You'll have to change into a

mindset of "what event should I hook into, and what events should I trigger, and when?" instead.

Why use Windows.Forms+GLControl instead of GameWindow?
The first thing you'll have to decide is:

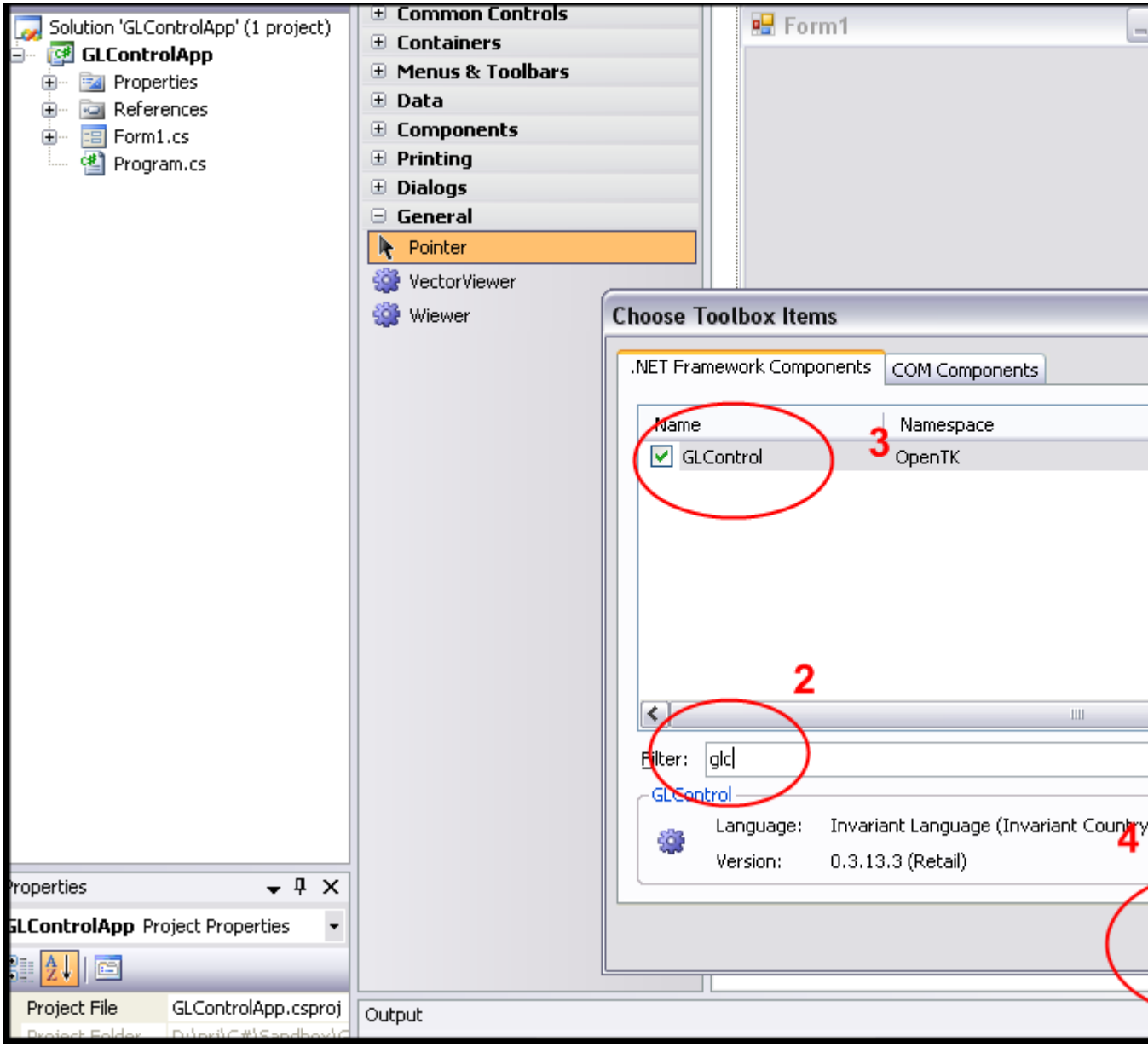
"Do I really need the added complexity of `Windows.Forms + embedded GLControl` compared to a windowed `GameWindow`?"

Here are some reasons why you would like to add that complexity:

1. You want to build a GUI-rich/RAD kind of application using the `Windows.Forms` controls. Eg. level editors or model viewers/editors may be in this category while a windowed game leans more towards a `GameWindow` kind of application.
2. You want to have an embedded OpenGL rendering panel inside an already existing `Windows.Forms` application.
3. You want to be able to do drag-and-drop into the rendering panel, for example dropping a model file into a model viewer.

Assuming you've got at least one of those reasons to build a `Windows.Forms+GLControl` based application, here's the steps, gotchas and whys for you.

Adding the GLControl to your Windows.Form
(I am assuming you are using Visual Studio 2005 Express Edition. Your mileage may vary if using VS2008 or Monodevelop -- I don't know the details for them -- but the follow sections should be the same no matter how you add the GLControl)
To begin with, create a Form on which you will place your `GLControl`. Right click in some empty space of the Toolbox, pick "Choose Items..." and browse for OpenTK.dll. Make sure you can find the "GLControl" listed in the ".NET Framework Components", as in the image below.



Then you can add the `GLControl` to your form as any .NET control. A GLControl named `glControl1` will be added to your `Form`.

The first thing you'll notice is the "junk graphics" inside `glControl1`. Under the hood, `GLControl` uses a so called `GLContext` to do the actual GL-rendering and so on, and this context is only created in runtime, not in design time. So no worries.

Order of creation

The fact that `glControl1`'s `GLContext` is created in runtime is important to remember however, since you cannot access or change `glControl1`'s properties reliably until the `GLContext` has been created. The same is true for any `GL.*` commands (or `Glu` for that matter!). The conceptual order is this:

1. First the Windows.Form constructor runs. *Don't touch* *`glControl/GL`*
2. Then the `Load` event of the form is triggered. *Don't touch* *`glControl/GL`*

3. Then the `Load` event of the GLControl is triggered. *OK to touch `glControl/GL`*
4. After the `Load` event handler has run, any event handler may touch `glControl/GL`.

So one approach to address this problem is having a `bool loaded=false;` member variable in your Form, which is set to `true` in the `Load` event handler of the GLControl:

```
using OpenTK.Graphics;
using OpenTK.Graphics.OpenGL;

public partial class Form1 : Form
{
  bool loaded = false;

  public Form1()
  {
    InitializeComponent();
  }

  private void glControl1_Load(object sender, EventArgs e)
  {
    loaded = true;
  }
}
```

Note: the GLControl.Load event is never fired -- please use the Form.Load event instead until this issue of OpenTK is fixed.

Then in any event handler where you are going to access `glControl1/GL` you can put a guard first of all:

```
 private void glControl1_Resize(object sender, EventArgs e)
{
  if (!loaded)
    return;
}
```

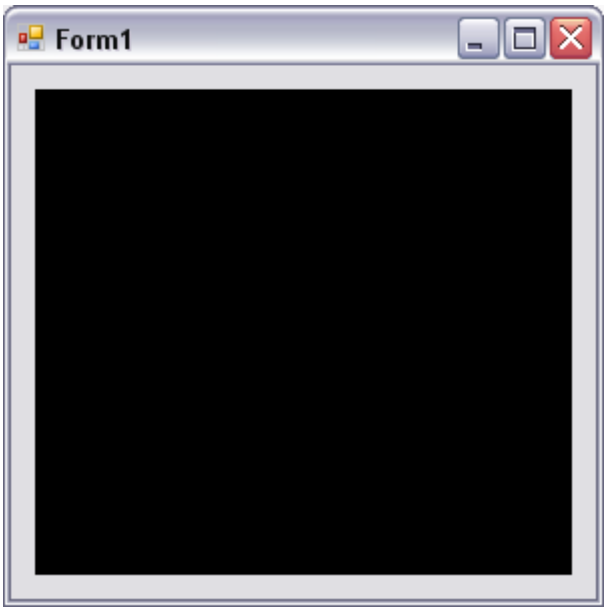
A popular way of adding a Load event handler to a Form is via the Properties window of Visual Studio, something like this:

1. Click anywhere on the GLControl in the Designer
2. Make sure glControl1 is listed in the Properties window
3. Click the "Events" button to list all events of glControl1
4. Double-click the empty cell right of the Load event to create and hook an event handler for the Load event

Hello World!
The absolutely minimal code you can add at this stage to see something is adding an event handler to the `Paint` event of `glControl1` and filling it with this:

```
 private void glControl1_Paint(object sender, PaintEventArgs e)
{
```

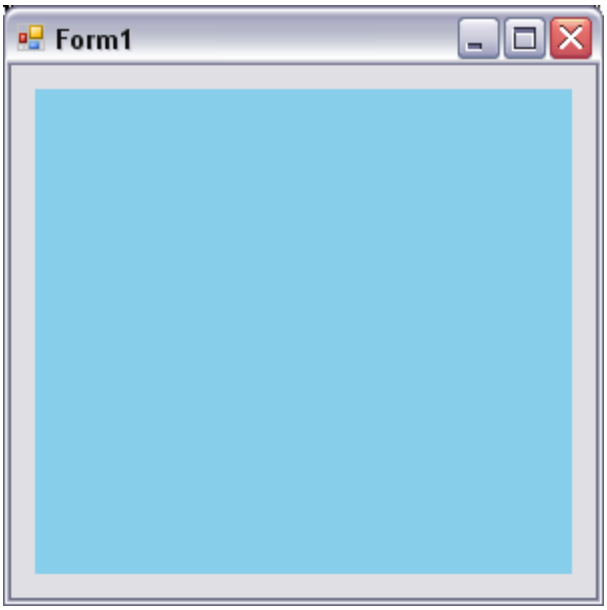
```
      if (!loaded) // Play nice
        return;

      GL.Clear(ClearBufferMask.ColorBufferBit |
ClearBufferMask.DepthBufferBit);
      glControl1.SwapBuffers();
    }
```



Yes! A black viewport. Notice that the `GLControl` provides a color- and a depth buffer, which we have to clear using `GL.Clear()`. *[TODO: how to control which buffers and formats the GLControl has? Possible at all?]*

Next thing would be setting the clear color. An appropriate place to do GL initialization is in the form's `Load` event handler:

```
     private void glControl1_Load(object sender, EventArgs e)
    {
      loaded = true;
      GL.ClearColor(Color.SkyBlue); // Yey! .NET Colors can be used
directly!
    }
```



Viewport initialization

Next thing we want to do is draw a single yellow triangle.

First we need to be good OpenGL citizen and setup an orthographic projection matrix using `GL.Ortho()`. We need to call `GL.Viewport()` also.

For now we'll add this in the `Load` event handler by the other initialization code -- ignoring the fact that we may want to allow the user to resize the window/`GLControl`. We'll look into window resizing later.

I put the viewport initialization in a separate method to make it a little more readable.

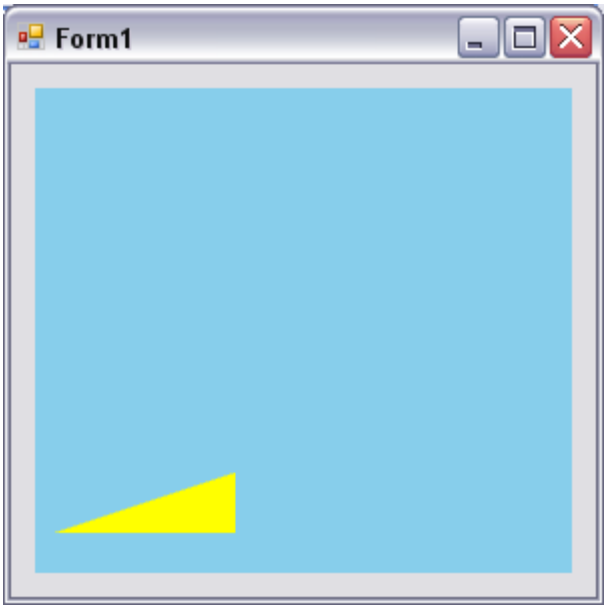
```
     private void glControl1_Load(object sender, EventArgs e)
    {
      loaded = true;
      GL.ClearColor(Color.SkyBlue);
      SetupViewport();
    }

    private void SetupViewport()
    {
      int w = glControl1.Width;
      int h = glControl1.Height;
      GL.MatrixMode(MatrixMode.Projection);
      GL.LoadIdentity();
      GL.Ortho(0, w, 0, h, -1, 1); // Bottom-left corner pixel has
coordinate (0, 0)
      GL.Viewport(0, 0, w, h); // Use all of the glControl painting
area
    }
```

And between `Clear()` and `SwapBuffers()` our yellow triangle:

```
     private void glControl1_Paint(object sender, PaintEventArgs e)
    {
      if (!loaded)
```

```
        return;

      GL.Clear(ClearBufferMask.ColorBufferBit |
ClearBufferMask.DepthBufferBit);

      GL.MatrixMode(MatrixMode.Modelview);
      GL.LoadIdentity();
      GL.Color3(Color.Yellow);
      GL.Begin(BeginMode.Triangles);
      GL.Vertex2(10, 20);
      GL.Vertex2(100, 20);
      GL.Vertex2(100, 50);
      GL.End();

      glControl1.SwapBuffers();
    }
```

Voila!



Keyboard input
Next thing we want to do is animate the triangle via user interaction. Every time `Space` is pressed, we want the triangle to move one pixel right.

The two general approaches to keyboard input in a `GLControl` scenario is using `Windows.Forms` key events and using the OpenTK `KeyboardDevice`. Since the rest of our program resides in the `Windows.Forms` world (our window might be a very small part of a large GUI) we'll play nice and use `Windows.Forms` key events in this guide.

We'll have an `int x=0;` variable that we'll increment in a `KeyDown` event handler. Adding it to the `glControl1` and not the Form, means the `glControl` has to be focused, ie. clicked by the user for key events to be sent to our handler.

```
     int x = 0;
    private void glControl1_KeyDown(object sender, KeyEventArgs e)
    {
      if (e.KeyCode == Keys.Space)
```

```
        x++;
    }
```

We add `GL.Translate()` to our `Paint` event handler:

```
     private void glControl1_Paint(object sender, PaintEventArgs e)
    {
      if (!loaded)
        return;

      GL.Clear(ClearBufferMask.ColorBufferBit |
ClearBufferMask.DepthBufferBit);

      GL.MatrixMode(MatrixMode.Modelview);
      GL.LoadIdentity();

      GL.Translate(x, 0, 0); // position triangle according to our x
variable

      ...
    }
```

.. and we run our program. But wait! Nothing happens when we push Space! The reason is, `glControl1` is not painted all the time; the operating systems window manager (Windows/X/OSX) makes sure as few `Paint` events as possible happens. Only on resize, obscured window and a couple of more situations do `Paint` events actually get triggered.

What we would like to do is have a way of telling the window manager *"This control needs to be repainted since the data it relies on has changed"*. We want to notify the window manager that our glControl1 should be repainted. Easy, `Invalidate()` to the resque:

```
     private void glControl1_KeyDown(object sender, KeyEventArgs e)
    {
      if (!loaded)
        return;
      if (e.KeyCode == Keys.Space)
      {
        x++;
        glControl1.Invalidate();
      }
    }
```

Focus behaviour
If you're anything like me you're wondering a little how this focusing behaves; let's find out!

A simple way is painting the triangle yellow when `glControl1` is focused and blue when it is not. Right:

```
     private void glControl1_Paint(object sender, PaintEventArgs e)
    {
      ...

      if (glControl1.Focused) // Simple enough :)
```

```
    GL.Color3(Color.Yellow);
  else
    GL.Color3(Color.Blue);
  GL.Begin(BeginMode.Triangles);

  ...
}
```

So now anytime the triangle is yellow, Space should work alright, and when it's blue any keyboard input will be ignored.

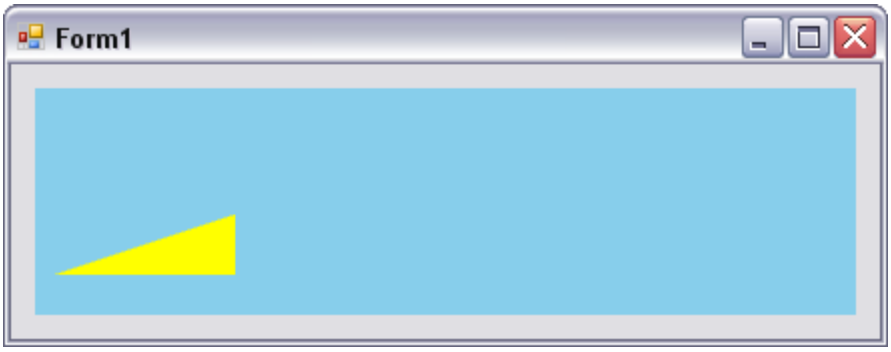
Freedom at last: resizing the window
We will now turn our attention to a sore teeth: window resizing.

Anytime a Windows.Forms control is resized, it's Resize event is triggered. That is true for glControl1 too. That's one piece in the puzzle.

The other piece to find is *"What do we need to update when a GLControl is resized?"* and the answer is *"The viewport and the projection matrix"*.

Well it seems our `SetupViewport()` will come in handy once more! Add an event handler to the Resize event of glControl1 and fill it in:

```
 private void glControl1_Resize(object sender, EventArgs e)
{
  SetupViewport();
}
```



There is still one problem though: if you shrink the window using eg. the bottom-right resize grip of the window, no repaint will trigger automatically. That's because the window manager makes assumptions about where the (0, 0) pixel of a control resides, namely in the top-left corner of the control. (Try resizing using the top-left grip instead - the triangle is repainted continously!). Our general fix to alleviate this problem is instructing the window manager that we really want the repaint to occur upon any resize event:

```
 private void glControl1_Resize(object sender, EventArgs e)
{
  SetupViewport();
  glControl1.Invalidate();
}
```

I want my main loop: driving animation using Application.Idle
So, what if we wanted our triangle to rotate continously? This would be childs play in a main loop scenario: just increment a `rotation` variable in the main loop, before we render the triangle.

But we don't have any loop. We only have events!

To mend for this lack of continuity we have to force Windows.Forms to do it our way. We want an event triggered every now and then, fast enough to get that realtime interactive feeling.

Now there are several ways to achieve this. One is adding a `Timer` control to our form, changing `rotation` in the timers `Tick` event handler. Another is adding a wild `Thread` to the soup. The first is a little too high-level and slow while the second is really low-level and a bit harder to get right.

We will take a third path and use a `Windows.Forms` event designed just for the purpose of being executed "when nothing else is going on": meet the `Application.Idle` event.

This event is special in a number of ways as you may have guessed already. It is not associated with any Form or other control, but with the program as a whole. You cannot hook into it from the GUI Designer; you'll have to add it manually -- for example in the `Load` event:

```
     private void glControl1_Load(object sender, EventArgs e)
    {
      loaded = true;
      GL.ClearColor(Color.SkyBlue);
      SetupViewport();
      Application.Idle += new EventHandler(Application_Idle); //
press TAB twice after +=
    }

    void Application_Idle(object sender, EventArgs e)
    {
    }
```

One good thing about the `Idle` event is that the corresponding event handlers are executed *on the Windows.Forms thread*. That is good since it means we can access all GUI controls without having to worry about threading issues, a pain we would have to deal with if we cooked our own thread.

So we simply increment our `rotation` variable in the `Idle` event handler and `Invalidate()` glControl1 -- business as usual.

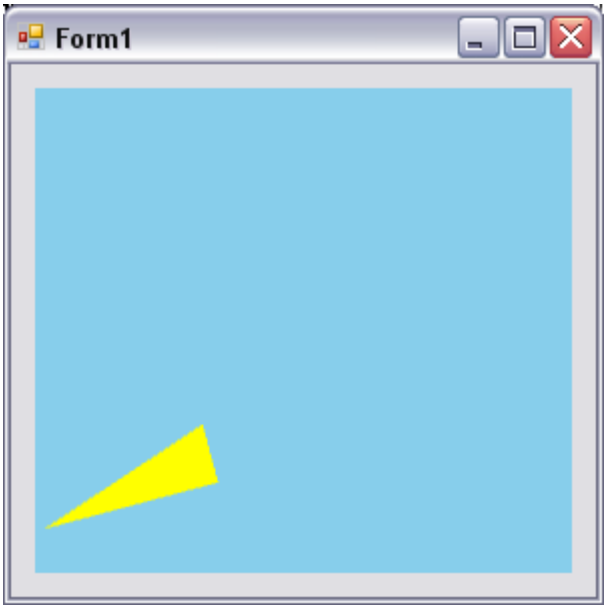
```
    float rotation = 0;
    void Application_Idle(object sender, EventArgs e)
    {
      // no guard needed -- we hooked into the event in Load handler
      while (glControl1.IsIdle)
      {
          rotation += 1;
```

```
            Render();
        }
    }
```

Let's update our rendering code while we're at it:

```
    private void Render()
    {
      ...
      if (glControl1.Focused)
        GL.Color3(Color.Yellow);
      else
        GL.Color3(Color.Blue);
      GL.Rotate(rotation, Vector3.UnitZ); // OpenTK has this nice
Vector3 class!
      GL.Begin(BeginMode.Triangles);
      ...
    }

    private void glControl1_Paint(object sender, PaintEventArgs e)
    {
        Render();
    }
```

Happy wonders! Look:



The triangle rotates slower when the window is big! How come?
(This might not be true if you have a super-fast-computer with a super-fast-graphics-card; but you want your game to run on your neighbours computer too, don't you?)

The reason is that windowed 3d rendering just is a lot slower than full-screen rendering, in general.

But you can reduce the damage by using a technique called *frame-rate independent animation*. The idea is simple: increment the `rotation` variable not with `1` but with

an amount that depends on the current rendering speed (if the speed is slow, increment `rotation` with a larger amount than if the speed is high).

But you need to be able to measure the current rendering speed or, equivalently, how long time it takes to render a frame.

Since .NET2.0 there is a class available to do high-precision time measurements called `Stopwatch`. Here's how to use it:

```
    Stopwatch sw = new Stopwatch();
    sw.Start();
    MyAdvancedAlgorithm();
    sw.Stop();
    double milliseconds = sw.Elapsed.TotalMilliseconds;
```

(*don't try with `DateTime.Now` -- it has a granularity of 10 or more milliseconds, which is in the same size as typical frame rendering -- worthless.*.)

Now, if we could measure the time it takes to perform the glControl painting, we would be close to making some kind of frame-rate-independent animation. But there is an even more elegant way: let's measure all time that is not `Application.Idle` time! Then we'll be sure it is not just the painting that is measured, but everything that has been going on since our last Idle run:

```
     Stopwatch sw = new Stopwatch(); // available to all event
handlers
    private void glControl1_Load(object sender, EventArgs e)
    {
      ...
      sw.Start(); // start at application boot
    }

    float rotation = 0;
    void Application_Idle(object sender, EventArgs e)
    {
      // no guard needed -- we hooked into the event in Load handler

      sw.Stop(); // we've measured everything since last Idle run
      double milliseconds = sw.Elapsed.TotalMilliseconds;
      sw.Reset(); // reset stopwatch
      sw.Start(); // restart stopwatch

      // increase rotation by an amount proportional to the
      // total time since last Idle run
      float deltaRotation = (float)milliseconds / 20.0f;
      rotation += deltaRotation;

      glControl1.Invalidate();
    }
```

Cool! The triangle spins with the same speed regardless of window size.

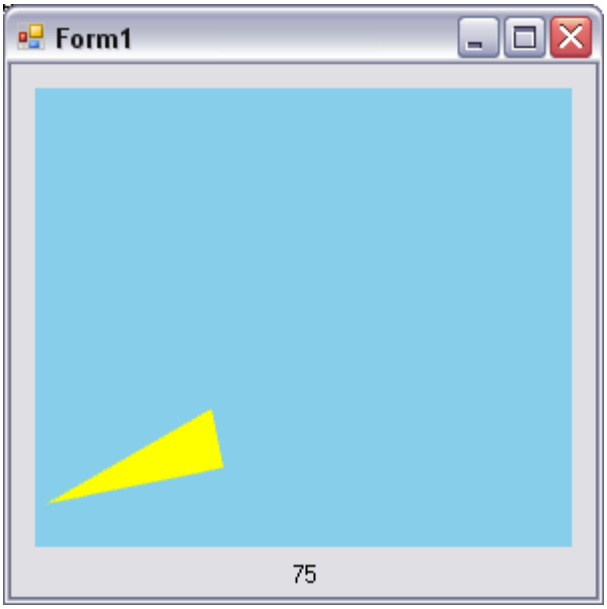
I want an FPS counter!
Yeah me too. It's quite simple now that we've got a `Stopwatch`.

The idea is just counting the Idle runs, and every second or so, update a Label control with the counter! But we'll have to know when a second has passed, so we need an accumulator variable adding all time slices together.

Quite a lot of logic started to add up in the `Idle` event handler, so I split it up a little:

```
 void Application_Idle(object sender, EventArgs e)
{
  double milliseconds = ComputeTimeSlice();
  Accumulate(milliseconds);
  Animate(milliseconds);
}

private double ComputeTimeSlice()
{
  sw.Stop();
  double timeslice = sw.Elapsed.TotalMilliseconds;
  sw.Reset();
  sw.Start();
  return timeslice;
}

float rotation = 0;
private void Animate(double milliseconds)
{
  float deltaRotation = (float)milliseconds / 20.0f;
  rotation += deltaRotation;
  glControl1.Invalidate();
}

double accumulator = 0;
int idleCounter = 0;
private void Accumulate(double milliseconds)
{
  idleCounter++;
  accumulator += milliseconds;
  if (accumulator > 1000)
  {
    label1.Text = idleCounter.ToString();
    accumulator -= 1000;
    idleCounter = 0; // don't forget to reset the counter!
  }
}
```

Our FPS counter in all its glory:



Can't you put the complete source code somewhere?
Sure, here it is:

```
 using System;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using OpenTK.Graphics;
using OpenTK.Graphics.OpenGL;
using System.Diagnostics;

namespace GLControlApp
{
  public partial class Form1 : Form
  {
    bool loaded = false;

    public Form1()
    {
      InitializeComponent();
    }

    Stopwatch sw = new Stopwatch(); // available to all event
handlers
    private void glControl1_Load(object sender, EventArgs e)
    {
      loaded = true;
      GL.ClearColor(Color.SkyBlue); // Yey! .NET Colors can be used
directly!
      SetupViewport();
      Application.Idle += new EventHandler(Application_Idle); //
press TAB twice after +=
      sw.Start(); // start at application boot
    }

    void Application_Idle(object sender, EventArgs e)
    {
```

```
      double milliseconds = ComputeTimeSlice();
      Accumulate(milliseconds);
      Animate(milliseconds);
    }

    float rotation = 0;
    private void Animate(double milliseconds)
    {
      float deltaRotation = (float)milliseconds / 20.0f;
      rotation += deltaRotation;
      glControl1.Invalidate();
    }

    double accumulator = 0;
    int idleCounter = 0;
    private void Accumulate(double milliseconds)
    {
      idleCounter++;
      accumulator += milliseconds;
      if (accumulator > 1000)
      {
        label1.Text = idleCounter.ToString();
        accumulator -= 1000;
        idleCounter = 0; // don't forget to reset the counter!
      }
    }

    private double ComputeTimeSlice()
    {
      sw.Stop();
      double timeslice = sw.Elapsed.TotalMilliseconds;
      sw.Reset();
      sw.Start();
      return timeslice;
    }

    private void SetupViewport()
    {
      int w = glControl1.Width;
      int h = glControl1.Height;
      GL.MatrixMode(MatrixMode.Projection);
      GL.LoadIdentity();
      GL.Ortho(0, w, 0, h, -1, 1); // Bottom-left corner pixel has
coordinate (0, 0)
      GL.Viewport(0, 0, w, h); // Use all of the glControl painting
area
    }

    private void glControl1_Paint(object sender, PaintEventArgs e)
    {
      if (!loaded)
        return;

      GL.Clear(ClearBufferMask.ColorBufferBit |
ClearBufferMask.DepthBufferBit);

      GL.MatrixMode(MatrixMode.Modelview);
      GL.LoadIdentity();

      GL.Translate(x, 0, 0);
```

```
      if (glControl1.Focused)
        GL.Color3(Color.Yellow);
      else
        GL.Color3(Color.Blue);
      GL.Rotate(rotation, Vector3.UnitZ); // OpenTK has this nice
Vector3 class!
      GL.Begin(BeginMode.Triangles);
      GL.Vertex2(10, 20);
      GL.Vertex2(100, 20);
      GL.Vertex2(100, 50);
      GL.End();

      glControl1.SwapBuffers();
    }

    int x = 0;
    private void glControl1_KeyDown(object sender, KeyEventArgs e)
    {
      if (e.KeyCode == Keys.Space)
      {
        x++;
        glControl1.Invalidate();
      }
    }

    private void glControl1_Resize(object sender, EventArgs e)
    {
      SetupViewport();
      glControl1.Invalidate();
    }
  }
}
```

Next step: What about multiple GLControls at once?
Yeah it is possible. It is even simple!

All you have to do is "make the appropriate GLControl current".

Let's say you have one GLControl named glCtrl1 and one named glCtrl2. And you have added handlers to the Paint event of both. This is what you do in the event handler of glCtrl1 (of course you do something similar in the event handler of glCtrl2!):

```
     private void glCtrl1_Paint(object sender, PaintEventArgs e)
    {
      if (!loaded)
        return;

      glCtrl1.MakeCurrent(); // Ohh.. It's that simple?

      GL.Clear(ClearBufferMask.ColorBufferBit |
ClearBufferMask.DepthBufferBit);
      ...
     }
```

The same is true for any code calling a GL.* or Glu.* method!

Although each GLControl has its own GraphicsContext, OpenTK will share OpenGL resources by default. This means, any context can use textures, display lists, etc. created on any other context. You can disable this behavior via the `GraphicsContext.ShareContexts` property.

Avoiding pitfalls in the managed world

This text is intended for someone with a C/OpenGL background.

Even though OpenTK automatically translates GL/AL calls from C# to C, some things work slightly differently in the managed world, when compared to plain C. This page describes a few rules you need to keep in mind:

Rules of thumb

1. **Use server storage rather than client storage.**
   A few legacy OpenGL functions use pointers to memory managed by the user. The most popular example is Vertex Arrays, with the GL.***Pointer family of functions.

   This approach cannot be used in a Garbage Collected environment (as .NET), as the garbage collector (GC) may move the contents of the buffer in memory. It is strongly recommended that you replace legacy Vertex Arrays with Vertex Buffer Objects, which do not suffer from this problem.

   Unlike OpenGL 2.1, OpenGL 3.0 will *not* contain any functions with client storage.

2. **Try to minimize the number of OpenGL calls per frame.**
   This is true for any programming environment utilizing OpenGL, but a little more important in the OpenTK case; while the OpenGL/OpenAL bindings are quite optimized, the transition from managed into unmanaged code incurs a small, but measurable, overhead.

   To minimize the impact of this overhead, try to minimize the number of OpenGL/OpenAL calls. A good rule of thumb is to make no more than 300-500 OpenGL calls per frame, which can be achieved by avoiding Immediate Mode, in favour of Display Lists and VBO's.

3. **For optimal math routine performance, use the `ref` and `out` overloads.**
   This is because Vector3, Matrix4 etc. are structures, not classes. Classes are passed by reference by default in C#.

```
4. Vector3 v1 = Vector3.UnitX;
5. Vector3 v2 = Vector3.UnitZ;
6. Vector3 v3 = Vector3.Zero;
7. v3 = v1 + v2;                          // requires three copies;
   slow.
   Vector3.Add(ref v1, ref v2, out v3); // nothing is copied;
   fast!
```

The same holds true when calling OpenGL functions:

```
GL.Vertex3(ref v1.X);  // pass a pointer to v1; fast!
GL.Vertex3(v1);        // copy the whole v1 structure; slower!
```

Measuring performance

1. *GameWindow provides built-in frames-per-second counters* [Someone with confidence in the details please fill in here] However, `GLControl` does not.
2. *A simple and convenient way* to measure the performance of your code, is via the .NET 2.0 / Mono 1.2.4 `Stopwatch` class. Use it like this:

```
Stopwatch sw = new Stopwatch();
sw.Start();
// Your code goes here.
sw.Stop();
double ms = sw.Elapsed.TotalMilliseconds;
```

   Note: Avoid using `DateTime.Now` or other `DateTime`-based method on any periods shorter than a couple of seconds, since its granularity is 10 ms or worse. (rumour has it, it may even go backwards on occasion!) Using `DateTime` to measure very long operations (several seconds) is OK.

7. *If you are on Windows*, you can download Fraps to measure how many frames per second are rendered in your application. For linux (and Windows), you can use the commercial tool gDEBugger [anyone: any similar tools for Mac? Any free tool for Linux?]

References:
http://www.gamedev.net/community/forums/topic.asp?topic_id=484756&whichp...

Chapter 3: OpenTK.Math

To help you write cleaner code, OpenTK defines commonly used vectors, quaternions and matrices to extend the standard scalar types. They are generally nicer to handle than the arrays which the C API expects.

For example: OpenTK allows you to set the parameters of `GL.Color()` in various ways, the color cyan is used in this case.

```
GL.Color( 0.0f, 1.0f, 1.0f );

Vector3 MyColor = new Vector3( 0.0f, 1.0f, 1.0f );
GL.Color( MyColor );

// requires the System.Drawing library
GL.Color( Color.Cyan );
```

Function overloads like this can be found all over OpenTK.Graphics and OpenTK.Audio, where applicable.

Overview
Everything listed below is type-safe, code using these types will work across all supported platforms.

Vectors

- Half-precision Floating-point
  - `Half` - new scalar type with 16 Bits of precision. More information can be found here.
  - `Vector2h` - 2-component vector of the type `Half`.
  - `Vector3h` - 3-component vector of the type `Half`.
  - `Vector4h` - 4-component vector of the type `Half`.
- Single-precision Floating-point
  - `Single` - standard scalar type with 32 Bits of precision.
  - `Vector2` - 2-component vector of the type `Single`.
  - `Vector3` - 3-component vector of the type `Single`.
  - `Vector4` - 4-component vector of the type `Single`.
- Double-precision Floating-point
  - `Double` - standard scalar type with 64 Bits of precision.
  - `Vector2d` - 2-component vector of the type `Double`.
  - `Vector3d` - 3-component vector of the type `Double`.
  - `Vector4d` - 4-component vector of the type `Double`.

Quaternion

- `Quaternion` - single-precision floating-point Quaternion using 4 components.
- `Quaterniond` - double-precision floating-point Quaternion using 4 components.

Row-Major Matrices

- `Matrix3d` - 3x3 double-precision Matrix.
- `Matrix4` - 4x4 single-precision Matrix.
- `Matrix4d` - 4x4 double-precision Matrix.

These are all value types (struct), not reference types. The implications of this can be read here.

Casts

For symmetry with established types, all OpenTK.Math types can be cast and allow serialization.

```
Vector2d TexCoord = new Vector2d( 0.2, 0.5 );
Vector2h HalfTexCoord = (Vector2h)TexCoord;

Vector3h Normal = (Vector3h)Vector3.UnitX;
```

Instance and Static Methods

The exact methods for each struct would be too numberous to list here, the function reference and inline documentation serve that purpose. It might not be obvious that some functionality is only available for instances of the structs, while others are static.

```
Vector3 Normal = Vector3.UnitX;
Normal.Normalize();

Vector4 TransformedVector = Vector4.Transform( Vector4.UnitX,
Matrix4.Identity );
```

Half-Type

The new half-precision floating-point type in OpenTK.Math is specifically designed for computer graphics. It is commonly used to reduce the memory footprint of floating-point textures, which can become huge at high resolutions. It is also useful for Vertex attributes, because it can help your vertex struct to stay within the 16 or 32 Byte boundary, which processors like.

Internally the 16 Bit are represented similar to IEEE floating-point numbers:

- 1 Sign Bit.
- 5 Exponent Bits.
- 10 Mantissa Bits. This is sometimes called the Significand, but called Mantissa in all OpenTK documentation.

It is important to understand that this type is a pure container to transfer data to or from the GPU, it does not support arithmetic operations. So if you want to use `Half` for Texture Coordinates or Normals, you have to perform any calculations with it in single- or double-precision first and then cast the final result to half-precision.

When using the `Half` type you should be aware of it's poor precision:

- Casting Math.PI to `Half` will result in 3.1406...
- Numbers above 2048 and below -2048 will be rounded to the closest representable number.
- The further away the value of the floating-point number gets from 1.0, the worse the Epsilon gets. Try to stay within [0.0 .. 1.0] range for best accuracy.

Chapter 4: OpenTK.Graphics (OpenGL and ES)

In order to use OpenGL functions, your System requires appropriate drivers for hardware acceleration.

The OpenGL Programming Guide is a book written by Silicon Graphics engineers and will introduce the reader into graphics programming. It is highly recommended you take a look at this resource to learn about the essential concepts in OpenGL.

These pages are more focused about OpenTK specific changes to the C API, and how to use OpenTK.Utility classes to assist with some common tasks.

Todo: Missing Pages

- window-related, etc. (brings the reader to the level of a Quickstart Template)
- GLSL related changes.
- OpenTK.Utility related

The GraphicsContext class

[Introduction]

The `OpenTK.Graphics.GraphicsContext` is a cross-platform wrapper around an *OpenGL context*. The context routes your OpenGL commands to the hardware driver for execution - which means you cannot use any OpenGL commands without a valid `GraphicsContext`.

[Constructors]

OpenTK creates a `GraphicsContext` automatically as part of the `GameWindow` and `GLControl` classes:

- You can specify the desired `GraphicsMode` of the context using the `mode` parameter. Use `GraphicsMode.Default` to set a default, compatible mode or specify the color, depth, stencil and anti-aliasing level manually.
- You can specify the OpenGL version you wish to use through the `major` and `minor` parameters. As per the OpenGL specs, the context will use the highest version that is *compatible* with the version you specified. The default values are 1 and 0 respectively, resulting in a 2.1 context.
- You can request an embedded (ES) context by specifying `GraphicsContextFlags.Embedded` to the `flags` parameter. The default value will construct a desktop (regular OpenGL) context.
- If you are creating the context manually, you must specify a valid `IWindowInfo` instance to the `window` parameter (see below). This is the default window the `GraphicsContext` will draw on and can be modified later using the `MakeCurrent` method.

Examples:

```
// Creates a 1.0-compatible GraphicsContext with GraphicsMode.Default
GameWindow window = new GameWindow();

// Creates a 3.0-compatible GraphicsContext with 32bpp color, 24bpp
depth
// 8bpp stencil and 4x anti-aliasing.
GLControl control = new GLControl(new GraphicsMode(32, 24, 8, 4), 3,
0);
```

Sometimes, you might wish to create a second context for your application. A typical use case is for background loading of resources. This is very simple to achieve:

```
// The new context must be created on a new thread
// (see remarks section, below)
// We need to create a new window for the new context.
// Note 1: new windows remain invisible unless you call
//         INativeWindow.Visible = true or IGameWindow.Run()
// Note 2: invisible windows fail the pixel ownership test.
//         This means that the contents of the front buffer are
undefined, i.e.
//         you cannot use an invisible window for offscreen
rendering.
//         You will need to use a framebuffer object, instead.
// Note 3: context sharing will fail if the main context is in use.
// Note 4: this code can be used as-is in a GLControl or GameWindow.

EventWaitHandle context_ready = new EventWaitHandle(false,
EventResetMode.AutoReset);
Context.MakeCurrent(null);

Thread thread = new Thread(() =>
{
    INativeWindow window = new NativeWindow();
    IGraphicsContext context = new
GraphicsContext(GraphicsMode.Default, window.WindowInfo);
    context.MakeCurrent(window.WindowInfo);

    while (window.Exists)
    {
         window.ProcessEvents();

         // Perform your processing here

         Thread.Sleep(1); // Limit CPU usage, if necessary
    }
});
thread.IsBackground = true;
thread.Start();

context_ready.WaitOne();
MakeCurrent();
```

If necessary, you can also instantiate a `GraphicsContext` manually. For this, you will need to provide an amount of platform-specific information, as indicated below:

```
using OpenTK.Graphics;
using OpenTK.Platform;
using Config = OpenTK.Configuration;
using Utilities = OpenTK.Platform.Utilities;

// Create an IWindowInfo for the window we wish to render on.
// handle - a Win32, X11 or Carbon window handle
// display - the X11 display connection
// screen - the X11 screen id
// root - the X11 root window
// visual - the desired X11 visual for the window
IWindowInfo wi = null;
if (Config.RunningOnWindows)
```

```
    wi = Utilities.CreateWindowsWindowInfo(handle);
else if (Config.RunningOnX11)
    wi = Utilities.CreateX11WindowInfo(display, screen, handle, root,
visual);
else if (Config.RunningOnMacOS)
    wi = Utilities.CreateMacOSCarbonWindowInfo(handle, false, false);

// Construct a new IGraphicsContext using the IWindowInfo from above.
IGraphicsContext context = new GraphicsContext(GraphicsMode.Default,
wi);
```

Finally, it is possible to instantiate a 'dummy' GraphicsContext for any OpenGL context created outside of OpenTK. This allows you to use OpenTK.Graphics with windows created through SDL or other libraries:

```
// The 'external' context has to be current on the calling thread:
GraphicsContext context = GraphicsContext.CreateDummyContext();
```

A common use-case is integration of OpenGL 3.x through OpenTK.Graphics into an existing application.

[Remarks]

A single GraphicsContext may be *current* on a single thread at a time. All OpenGL commands are routed to the context which is current on the calling thread - issuing OpenGL commands from a thread without a current context may result in a `GraphicsContextMissingException`. This is a safeguard placed by OpenTK: under normal circumstances, you'd get an abrupt and unexplained crash.

[Methods]

- **MakeCurrent**

  You can use the `MakeCurrent()` instance method **to make a context current** on the calling thread. If a context is already current on this thread, it will be replaced and made non-current. A single context may be current on a single thread at any moment - trying to make it current on two or more threads will result in an error. You can make a context non-current by calling `MakeCurrent(null)` on the correct thread.

  To retrieve the current context use the `GraphicsContext.CurrentContext` static property.

  If you wish to use OpenGL on multiple threads, you can either:

  - create one OpenGL context for each thread, or
  - use `MakeCurrent()` to make move a single context between threads.

  Both alternatives can be quite complicated to implement correctly. For this reason, it is usually advisable to restrict all OpenGL commands to a single thread, typically your main application thread, and use asynchronous method calls to invoke OpenGL from different threads. The `GLControl` provides the

`GLControl.BeginInvoke()` method to simplify asynchronous method calls from secondary threads to the main `System.Windows.Forms.Application` thread. The `GameWindow` does not provide a similar API.

To use multiple contexts on a single thread, call MakeCurrent to select the correct context prior to any OpenGL commands. For example, if you have two GLControls on a single form, you must call MakeCurrent() on the correct GLControl for each Load, Resize or Paint event.

`GLControl.MakeCurrent()` and `GameWindow.MakeCurrent()` are instance methods that simplify the handling of contexts.

- **SwapBuffers**

  OpenTK creates **double-buffered contexts** by default. Single-buffered contexts are considered deprecated, since they do not work correctly with compositors found on modern operating systems.

  A double-buffered context offers two color buffers: a "back" buffer, where all rendering takes place, and a "front" buffer which is displayed to the user. The `SwapBuffers()` method "swaps" the front and back buffers and displays the result of the rendering commands to the user. The contents of the new back buffer are undefined after the call to `SwapBuffers()`.

  The typical rendering process looks similar to this:

```
// Clear the back buffer.
GL.Clear(ClearBufferMask.ColorBufferBit |
ClearBufferMask.DepthBufferBit);

// Issue rendering commands, like GL.DrawElements

// Display the final rendering to the user
GraphicsContext.CurrentContext.SwapBuffers();
```

  Note that caching the current context will be more efficient than retrieving it through `GraphicsContext.CurrentContext`. For this reason, both `GameWindow` and `GLControl` use a cached `GraphicsContext` for efficiency.

[Stereoscopic rendering]

You can create a `GraphicsContext` that supports **stereoscopic rendering** (also known as "quad-buffer stereo"), by setting the `stereo` parameter to `true` in the context constructor. `GameWindow` and `GLControl` also offer this parameter in their constructors.

Contexts that support quad-buffer stereo distinguish the back and front buffers between "left" and "right" buffer. In other words, the context contains four distinct color buffers (hence the name): back-left, back-right, front-left and front-right. Check out the stereoscopic rendering page for more information (*[Todo: add article and link]*).

Please note that quad-buffer stereo is typically not supported on consumer video cards. You will need a workstation-class video card, like Ati's FireGL/FirePro or Nvidia's Quadro series, to enable stereo rendering. Trying to enable stereo rendering on consumer video cards will typically result in a context without stereo capabilities.

[Accessing internal information]

`GraphicsContext` hides an amount of low-level, implementation-specific information behind the `IGraphicsContextInternal` interface. This information includes the raw context handle, the platform-specific `IGraphicsContext` implementation and methods to initialize OpenGL entry points (`GetAddress()` and `LoadAll()`).

To access this information, cast your `GraphicsContext` to `IGraphicsContextInternal`:

```
IntPtr raw_context_handle = (my_context as
IGraphicsContextInternal).Context.Handle;
IntPtr function_address = (my_context as
IGraphicsContextInternal).GetAddress("glGenFramebufferEXT");
```

Using an external OpenGL context with OpenTK

Starting with version 0.9.1, OpenTK requires the existence of an OpenGL context prior to the initialization of the OpenGL subsystem. In other words, you cannot use any OpenGL methods prior to the creation of a `GraphicsContext`.

If you create the OpenGL context through an external library (for example SDL or GTK#), you will need to inform OpenTK of the context's existence using the `GraphicsContext.CreateDummyContext()` static method. This method will return a new `GraphicsContext` instance for the context that is current on the calling thread. Optionally, you can pass the handle (`IntPtr`) of a specific external context to `CreateDummyContext`; in this case, the external context need not be current on the calling thread.

You will typically call this method as soon as the external context is created. For example, using Tao.Glfw:

```
Tao.Glfw.glfwOpenWindow(640, 480, 8, 8, 8, 8, 16, 0,
Tao.Glfw.GLFW_WINDOW);
OpenTK.Graphics.GraphicsContext.CreateDummyContext();

// You may now use OpenTK.Graphics methods normally.
```

Please note that it is an error to call `CreateDummyContext()` multiple times for the same external context.

Textures

The following pages will describe the concepts of OpenGL Textures, Frame Buffer Objects and Pixel Buffer Objects. These concepts apply equally to OpenGL and OpenGL|ES - differences between the two will be noted in the text or in the example source code.

Loading a texture from disk

Before going into technical details about textures in the graphics pipeline, it is useful to know how to actually *load* a texture into OpenGL.

A simple way to achieve this is to use the `System.Drawing.Bitmap` class (MSDN documentation). This class can decode BMP, GIF, EXIG, JPG, PNG and TIFF images into system memory, so the only thing we have to do is send the decoded data to OpenGL. Here is how:

```
using System.Drawing;
using System.Drawing.Imaging;
using OpenTK.Graphics.OpenGL;

int LoadTexture(string filename)
{
    if (String.IsNullOrEmpty(filename))
        throw new ArgumentException(filename);

    int id = GL.GenTexture();
    GL.BindTexture(TextureTarget.Texture2D, id);

    Bitmap bmp = new Bitmap(filename);
    Bitmap bmp_data = bitmap.LockBits(new Rectangle(0, 0, bmp.Width,
bmp.Height), ImageLockMode.ReadOnly,
System.Drawing.Imaging.PixelFormat.Format32bppArgb);

    GL.TexImage2D(TextureTarget.Texture2D, 0,
PixelInternalFormat.Rgba, bmp_data.Width, bmp_data.Height, 0,
        OpenTK.Graphics.OpenGL.PixelFormat.Bgra,
PixelType.UnsignedByte, bmp_data.Scan0);

    bmp.UnlockBits(bmp_data);

    // We haven't uploaded mipmaps, so disable mipmapping (otherwise
the texture will not appear).
    // On newer video cards, we can use GL.GenerateMipmaps() or
GL.Ext.GenerateMipmaps() to create
    // mipmaps automatically. In that case, use
TextureMinFilter.LinearMipmapLinear to enable them.
    GL.TexParameter(TextureTarget.Texture2D,
TextureParameterName.TextureMinFilter, (int)TextureMinFilter.Linear);
    GL.TexParameter(TextureTarget.Texture2D,
TextureParameterName.TextureMagFilter, (int)TextureMagFilter.Linear);

    return id;
}
```

Now you can bind this texture id to a sampler (with `GL.Uniform1`) and use it in your shaders. If you are not using shaders, you should enable texturing (with `GL.Enable`) and bind the texture (`GL.BindTexture`) prior to rendering.

2D Texture differences

The most commonly used textures are 2-dimensional. There exist 3 kinds of 2D textures:

1. Texture2D
   Power of two sized (POTS) E.g: 1024²
   These are supported on all OpenGL 1.2 drivers.
   - MipMaps are allowed.
   - All filter modes are allowed.
   - Texture Coordinates are addressed parametrically: [0.0f ... 1.0f]x[0.0f ... 1.0f]
   - All wrap modes are allowed.
   - Borders are supported. (Exception: S3TC Texture Compression does not allow borders)
2. Texture2D
   Non power of two sized (NPOTS) E.g: 640*480.
   `GL.SupportsExtension( "ARB_texture_non_power_of_two" )` must evaluate to true.
   - MipMaps are allowed.
   - All filter modes are allowed.
   - Texture Coordinates are addressed parametrically: [0.0f ... 1.0f]x[0.0f ... 1.0f]
   - All wrap modes are allowed.
   - Borders are supported. (Exception: S3TC Texture Compression does not allow borders)
3. TextureRectangle
   Arbitrary size. E.g: 640*480.
   `GL.SupportsExtension( "ARB_texture_rectangle" )` must evaluate to true.
   - MipMaps are **not** allowed.
   - Only Nearest and Linear filter modes are allowed. (default is Linear)
   - Texture Coordinates are addressed non-parametrically: [0..width]x[0..height]
   - Only Clamp and ClampToEdge wrap modes are allowed. (default is ClampToEdge)
   - Borders are **not** supported.

Note that 1 and 2 both use the same tokens. The only difference between them is the size.

S3 Texture Compression

Introduction
A widely available Texture Compression comes from S3, mostly due to Microsoft licensing it and including it into DirectX. It was added into the file format DDS (DirectDraw Surface), which is basically a copy of the Texture in Video Memory.

Every graphics accelerator compatible with DirectX 7 or higher supports this Texture Compression.

The DXT Formats
What the S3 Texture Compression (abbreviation: S3TC. The Formats are named DXTn, where (1 <= n <= 5)) does is encode the whole Image into Blocks of 4x4 Texel, instead of storing every single Texel of the Image. Thus the ideal compressed Texture dimension is a multiple of 4, like 640x480 or a power of 2, which can be nicely fit into these Blocks. This is the ideal and not a restriction, the specification allows any non-power-of-two dimension, but will internally use a 4x4 Block for a Texture with the size of 2x1 (the other Texels in the Block are undefined).

This results into an 1:6 compression for DXT1 and 1:4 compression for DXT3/5, which translates into smaller disk size, load times and also render times.

DXT1, 8 Bytes per Block, Accuracy: R5G6B5 or R5G5B5A1
DXT3, 16 Bytes per Block, Accuracy: R5G6B5A8
DXT5, 16 Bytes per Block, Accuracy: R5G6B5A8

The formats DXT2 and DXT4 do exist, but they include pre-multiplied Alpha which is problematic when blending with images with explicit Alpha (RGBA, DXT3/5, etc). That's why those formats have barely been used, and are partially not supported by hardware and export/import tools. Avoiding DXT2/4 is strongly recommended, as they offer no beneficial functionality over established formats.

Compressed vs. Uncompressed
You probably guessed it already, there is a catch involved when reliably shrinking an image to 25% of it's uncompressed size: A lossy compression technique. This quality loss involved, which can be altered by tweaking the Filter options when compressing the image, is different to the one used in JPG compression. Although both formats - .dds and .jpg - are designed to compress an Image, the S3TC format was developed with graphics hardware in mind.

A bilinear Texture lookup usually reads 2x2 Texels from the Texture and interpolates those 4 Texels to get the final Color. Since a Block consists of 4x4 Texels, there is a good chance that all 4 Texels - which must be examined for the bilinear lookup - are in the same Block. This means that the worst case scenario involves reading 4 Blocks, but usually only 1-2 Blocks are used to achieve the bilinear lookup. When using uncompressed Textures, every bilinear lookup requires reading 4 Texels.

If you do the maths now you will notice that the compressed image actually needs 16 Bytes for 1 Block of RGBA Color, but the uncompressed 4 Texels of RGBA need 16 Bytes too. And yes, if you would only draw a single Pixel on the screen all this would not bring any noticable performance gains, actually it would be slower if multiple Blocks must be read to do the lookup.

However in OpenGL you typically draw more than a single Pixel, at least a Triangle. When the Triangle is rasterized, alot of Pixels will be very close to each other, which means their 2x2 lookup is very likely in the same 4x4 Block used by the last lookup, or a close neighbour. Graphic cards usually support this locality by using a small

amount of memory in the chip for a dedicated Texture Cache. If a Cache hit is made, the cost for reading the Texels is very low, compared to reading from Video Memory.

That's why S3TC does decrease render times: the earlier mentioned 16 Bytes of a DXTn Block contain 16 Texels (1 Byte per Texel), while 16 Bytes of uncompressed Texture only contain 4 Texels (4 Bytes per Texel). Alot more data is stored in the 16 Bytes of DXTn, and alot of lookups will be able to use the fast Texture Cache. The game Quake 3 Arena's Framerate increases by ~20% when using compressed Textures, compared to using uncompressed Textures.

Restrictions
Although you might be convinced now that Texture Compression is something worth looking into, do handle it with care. After all, it's a lossy compression Technique which introduces compression Artifacts into the Texture. For Textures that are close to the Viewer this will be noticed, that's why 2D Elements which are drawn very close to the near Plane - like the Mouse Cursor, Fonts or User Interface Elements like the Health display - are usually done with uncompressed Textures, which do not suffer from Artifacts.
As a rule of thumb, do not use Texture Compression where 1 Texel in the Texture will map to 1 Pixel on the Screen.

Using OpenTK.Utilities .dds loader
At the time of writing, the .dds loader included with OpenTK can handle compressed 2D Textures and compressed Cube Maps. Keep in mind that the loader expects a valid OpenGL Context to be present. It will only read the file from disk and upload all MipMap levels to OpenGL. It will NOT set minification/magnification filter or wrapping mode, because it cannot guess how you intent to use it.

void LoadFromDisk(string filename, bool flip, out int texturehandle, out TextureTarget dimension)

Input Parameter: filename
A string used to locate the DDS file on the harddisk, note that escape-sequences like "\n" are NOT stripped from the string.

Input Parameter: flip
The DDS format is designed to be used with DirectX, and that defines GL.TexCoord2(0.0, 0.0) at top-left, while OpenGL uses bottom-left. If you wish to use the default OpenGL Texture Matrix, the Image must be flipped before loading it as Texture into OpenGL.

Output Parameter: texturehandle
If there occured any error while loading, the loader will return "0" in this parameter. If >0 it's a valid Texture that can be used with GL.BindTexture.

Output Parameter: dimension
This parameter is used to identify what was loaded, currently it can return "Invalid", "Texture2D" or "TextureCube".

Example Usage

```
TextureTarget ImageTextureTarget;
int ImageTextureHandle;
ImageDDS.LoadFromDisk( @"YourTexture.dds", true, out
ImageTextureHandle, out ImageTextureTarget );
if ( ImageTextureHandle == 0 || ImageTextureTarget ==
TextureTarget.Invalid )
   // loading failed

// load succeeded, Texture can be used.
GL.BindTexture( ImageTextureTarget, ImageTextureHandle );
GL.TexParameter( ImageTextureTarget,
TextureParameterName.TextureMagFilter, (int) TextureMagFilter.Linear
);
int[] MipMapCount = new int[1];
GL.GetTexParameter( ImageTextureTarget,
GetTextureParameter.TextureMaxLevel, out MipMapCount[0] );
if ( MipMapCount == 0 ) // if no MipMaps are present, use linear
Filter
  GL.TexParameter( ImageTextureTarget,
TextureParameterName.TextureMinFilter, (int) TextureMinFilter.Linear
);
else // MipMaps are present, use trilinear Filter
  GL.TexParameter( ImageTextureTarget,
TextureParameterName.TextureMinFilter, (int)
TextureMinFilter.LinearMipmapLinear );
```

Remember that you must first GL.Enable the states Texture2D or TextureCube, before using the Texture in drawing.

Useful links:

ATi Compressonator:
http://ati.amd.com/developer/compressonator.html

nVidia's Photoshop Plugin:
http://developer.nvidia.com/object/photoshop_dds_plugins.html

nVidia's GPU-accelerated Texture Tools:
http://developer.nvidia.com/object/texture_tools.html

Detailed comparison of uncompressed vs. compressed Images:
http://www.digit-life.com/articles/reviews3tcfxt1/

OpenGL Extension Specification:
http://www.opengl.org/registry/specs/EXT/texture_compression_s3tc.txt

Microsoft's .dds file format specification (was used to build the OpenTK .dds loader)
http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/dx81_c/...

DXT Compression using CUDA
http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/d...

Real-Time YCoCg-DXT Compression
http://news.developer.nvidia.com/2007/10/real-time-ycocg.html

Last Update of the Links: January 2008

Frame Buffer Objects (FBO)

Every OpenGL application has at least one framebuffer. You can think about it as a digital copy of what you see on your screen. But this also implies a restriction, you can only see 1 framebuffer at a time on-screen, but it might be desireable to have multiple off-screen framebuffers at your disposal. That's where Frame Buffer Object (FBO) comes into play.

Typical usage for FBO is High Dynamic Range Rendering, Shadow Mapping and other Render-To-Texture effects. Assuming the buzzwords tell you nothing, here's a quick example scenario. We have a Texture2D of a sign that has some wooden texture and reads "Blacksmith". However you intend to localize that sign, so the german version of your game reads "Schmiede" or the spanish version "herrería". What are the options? Manually create a new Texture for every sign in the game with a paint program? No. All you need is the wooden texture of the sign, without any letters. The texture can be used as target for Render-To-Texture, and OpenTK.Fonts provides you a way to write any text you like ontop of that texture.

The traditional approach to achieve that was rendering into the visible framebuffer, read the information back with GL.ReadPixels() or GL.CopyTexSubImage(), then clear the screen and proceed with rendering as usual. With FBO the copy can be avoided, since it allows to render directly into a texture.

Framebuffer Layout

A framebuffer consists of at least one of these buffers:

- A depth buffer, with or without stencil mask. Typical depth buffer formats are 16, 24, 32 Bit integer or 32 Bit floating point. Stencil buffers can only be 8 Bits in size, a good mixed depth and stencil format is depth 24 Bit with stencil 8 Bit.
- Color buffer(s) have 1-4 components, namely Red, Green, Blue and Alpha. Typical color buffer formats are RGBA8 (8 Bit per component, total 32 Bit) or RGBA16f (16 Bit floating point per component, total 64 Bit). This list is far from complete, there exist dozens of formats with different amount of components and precision per component.

Please note that there is no requirement to use both. It's perfectly valid to create a FBO which has only a color attachment but no depth attachment. Or the other way around.

When you use more than one buffer, some restrictions apply: All attachments to the FBO must have the same width and height. All color buffers must use the same format. For example, you cannot attach a RGBA8 and a RGBA16f Texture to the same FBO, even if they have the same width and height. OpenGL 3.0 does relax this restriction, by allowing attachments of different sizes to be attached. But only the smallest area covered by all attachments can be written to. The Extension

EXTX_mixed_framebuffer_formats allows attaching different formats to the framebuffer, however this is reported to be very slow so far.

Renderbuffers

FBO allows 2 different types of targets to be attached to it. The already known textures 1D, 2D, Rectangle, 3D or Cube map, and a new type: the renderbuffer. They are not restricted to depth or stencil like the name might suggest, they can be used for color formats aswell.

Renderbuffer
Pro:

- May support formats which are not available as texture.
- Allows multisampling through Extensions.

Con:

- Does not allow MipMaps, filter or wrapping mode to be specified.
- Cannot be bound as sampler for shaders.
- Restricted to be a 2-dimensional image.

Texture2D
Pro:

- Allows MipMaps, filter and wrapping modes, just like every other texture.
- Can be bound as sampler to a shader.

Con:

- Might be slower than a renderbuffer, depending on hardware.

As a rule of thumb, do not use a renderbuffer if you plan to use the FBO attachments as textures at some later stage. The copy from renderbuffer into a texture will perform worse than rendering directly to the texture.

Let's take the wooden "Blacksmith" sign example from earlier again. The required end result must be a Texture2D, which can be bound when drawing the geometry of the sign. To give an overview about the options, here are some brief summaries how to accomplish obtaining the desired Texture2D:

1. Using a visible framebuffer.
   The wooden texture is drawn into the framebuffer. Text is drawn. The final Texture is copied into a Texture2D. The screen must be cleared when done.
2. Using a renderbuffer.
   The renderbuffer must be attached and the FBO bound. The wooden texture is drawn. Text is drawn. The final Texture is copied into a Texture2D. Either the renderbuffer is redundant now, or the screen must be cleared.
3. Using a Texture2D.
   The texture must be attached and the FBO bound. Only Text is drawn. Done.

Example Setup

To give a concrete example how all this theory looks in practice: let's create a color texture, a depth renderbuffer and a FBO, then attach the texture and renderbuffer to the FBO. I'm assuming you read the VBO tutorial before this, so I'm not going through the purpose of handles, GL.Gen*, GL.Bind* and GL.Delete* functions again. Note that this is a C API and the same rule of binding 0 to disable or detach something is valid here too. E.g. GL.Ext.BindFramebuffer(FramebufferTarget.FramebufferExt, 0); will disable the last bound FBO and return rendering back to the visible window-system provided framebuffer.

```
const int FboWidth = 512;
const int FboHeight = 512;

uint FboHandle;
uint ColorTexture;
uint DepthRenderbuffer;

// Create Color Texture
GL.GenTextures( 1, out ColorTexture );
GL.BindTexture( TextureTarget.Texture2D, ColorTexture );
GL.TexParameter( TextureTarget.Texture2D,
TextureParameterName.TextureMinFilter, (int) TextureMinFilter.Nearest
);
GL.TexParameter( TextureTarget.Texture2D,
TextureParameterName.TextureMagFilter, (int) TextureMagFilter.Nearest
);
GL.TexParameter( TextureTarget.Texture2D,
TextureParameterName.TextureWrapS, (int) TextureWrapMode.Clamp );
GL.TexParameter( TextureTarget.Texture2D,
TextureParameterName.TextureWrapT, (int) TextureWrapMode.Clamp );
GL.TexImage2D( TextureTarget.Texture2D, 0, PixelInternalFormat.Rgba8,
FboWidth, FboHeight, 0, PixelFormat.Rgba, PixelType.UnsignedByte,
IntPtr.Zero );

// test for GL Error here (might be unsupported format)

GL.BindTexture( TextureTarget.Texture2D, 0 ); // prevent feedback,
reading and writing to the same image is a bad idea

// Create Depth Renderbuffer
GL.Ext.GenRenderbuffers( 1, out DepthRenderbuffer );
GL.Ext.BindRenderbuffer( RenderbufferTarget.RenderbufferExt,
DepthRenderbuffer );
GL.Ext.RenderbufferStorage(RenderbufferTarget.RenderbufferExt,
(RenderbufferStorage)All.DepthComponent32, FboWidth, FboHeight);

// test for GL Error here (might be unsupported format)

// Create a FBO and attach the textures
GL.Ext.GenFramebuffers( 1, out FboHandle );
GL.Ext.BindFramebuffer( FramebufferTarget.FramebufferExt, FboHandle
);
GL.Ext.FramebufferTexture2D( FramebufferTarget.FramebufferExt,
FramebufferAttachment.ColorAttachment0Ext, TextureTarget.Texture2D,
ColorTexture, 0 );
GL.Ext.FramebufferRenderbuffer( FramebufferTarget.FramebufferExt,
FramebufferAttachment.DepthAttachmentExt,
RenderbufferTarget.RenderbufferExt, DepthRenderbuffer );
```

```
// now GL.Ext.CheckFramebufferStatus(
FramebufferTarget.FramebufferExt ) can be called, check the end of
this page for a snippet.

// since there's only 1 Color buffer attached this is not explicitly
required
GL.DrawBuffer(
(DrawBufferMode)FramebufferAttachment.ColorAttachment0Ext );

GL.PushAttrib( AttribMask.ViewportBit ); // stores GL.Viewport()
parameters
GL.Viewport( 0, 0, FboWidth, FboHeight );

// render whatever your heart desires, when done ...

GL.PopAttrib( ); // restores GL.Viewport() parameters
GL.Ext.BindFramebuffer( FramebufferTarget.FramebufferExt, 0 ); //
return to visible framebuffer
GL.DrawBuffer( DrawBufferMode.Back );
```

At this point you may bind the ColorTexture as source for drawing into the visible framebuffer, but be aware that it is still attached as target to the created FBO. That is only a problem if the FBO is bound again and the texture is used at the same time for being a FBO attachment target and the source of a texturing operation. This will cause feedback effects and is most likely not what you intended.
You may detach the ColorTexture from the FBO - the texture contents itself is not affected - by calling GL.Ext.FramebufferTexture2D() and attach a different target than ColorTexture to the ColorAttachment0 slot, for example simply 0. However the FBO would then be incomplete due to the missing color attachment, the best course of action is to detach the DepthRenderbuffer too and delete the renderbuffer and the FBO. Do not repeatedly attach and detach the same Texture if you want to update it every frame - just keep it attached to the FBO and make sure no feedback situation arises.

It is valid to attach the same texture or renderbuffer to multiple FBO at the same time. Example: you can avoid copies and save memory by attaching the same depth buffer to the FBOs, instead of creating multiple depth buffers and copy between them.

Special care has to be taken about 2 states that are always affected by FBOs: GL.Viewport() and GL.DrawBuffer(s). When switching from the visible framebuffer to a FBO, you should always set a proper viewport and drawbuffer. Switching framebuffer targets is such an expensive operation that the cost of the 2 extra calls to set up drawbuffers and viewport can be ignored. In the example setup above, the Viewport was stored and restored using GL.PushAttrib() and GL.PopAttrib(), but you may ofcourse specify it manually using GL.Viewport().

GL.DrawBuffer(s)

A FBO supports multiple color buffer attachments, if they have the same dimension and the same format. It is allowed to attach multiple color buffers - but only draw to one of them - by using the GL.DrawBuffer() command. Selecting multiple color buffers to write to is done with the GL.DrawBuffer**s**() command, which expects an array like this:

```
DrawBuffersEnum[] bufs = new DrawBuffersEnum[2] {
(DrawBuffersEnum)FramebufferAttachment.ColorAttachment0Ext,
(DrawBuffersEnum)FramebufferAttachment.ColorAttachment1Ext }; //
fugly, will be addressed in 0.9.2

GL.DrawBuffers( bufs.Length, bufs );
```

This code declares the color attachments 0 and 1 as buffers that can be written to. In practice this makes only sense if you're writing shaders with GLSL. (Look up "gl_FragData" for further info)

The exact number how many attachments are supported by the hardware must be queried through GL.GetInteger(GetPName.MaxColorAttachmentsExt, ...) and the number of allowed Drawbuffers at the same time through GL.GetInteger(GetPName.MaxDrawBuffers, ...)

To select which buffer is affected by GL.ReadPixels() or GL.CopyTex*() calls, use GL.ReadBuffer().

Remarks

For the sake of simplicity, the window-system provided framebuffer was called "visible framebuffer". In reality this is only true if you requested a single-buffer context from OpenGL, but the more likely case is that you requested a double-buffered context. When using double buffers, the 'back' buffer is the one used for drawing and never visible on screen, the 'front' buffer is the one that is visible on screen. The two buffers are swapped with each other when you call `this.SwapBuffers()`, to avoid that slow computers show unfinished images on screen. FBO are not designed to be double buffered, because they are off-screen at all times.

The wooden "Blacksmith" sign example has some hidden complexities that are ignored for the sake of simplicity, such as that you may not want to print with standard fonts on the sign, that words in different languages can have different length or that it might be desireable to add an additional mask when writing the text to simulate the paint peeling off the sign.

This page does not cover all commands exposed by FBO. For a more detailed description you'll have to dig through the official specification.

These Extensions were merged into ARB_framebuffer_objects with OpenGL 3.0:

EXT_framebuffer_multisample allows the creation of renderbuffers with n samples per image.

EXT_framebuffer_blit allows to bind 2 FBO at the same time. One for reading and one for writing. Without this Extension the active framebuffer is used for both: reading and writing.

Snippet how to interpret the possible results from GL.CheckFramebufferStatus

```
        private bool CheckFboStatus( )
        {
            switch ( GL.Ext.CheckFramebufferStatus(
FramebufferTarget.FramebufferExt ) )
            {
            case FramebufferErrorCode.FramebufferCompleteExt:
                {
                    Trace.WriteLine( "FBO: The framebuffer is
complete and valid for rendering." );
                    return true;
                }
            case
FramebufferErrorCode.FramebufferIncompleteAttachmentExt:
                {
                    Trace.WriteLine( "FBO: One or more attachment
points are not framebuffer attachment complete. This could mean
there’s no texture attached or the format isn’t renderable. For color
textures this means the base format must be RGB or RGBA and for depth
textures it must be a DEPTH_COMPONENT format. Other causes of this
error are that the width or height is zero or the z-offset is out of
range in case of render to volume." );
                    break;
                }
            case
FramebufferErrorCode.FramebufferIncompleteMissingAttachmentExt:
                {
                    Trace.WriteLine( "FBO: There are no attachments."
);
                    break;
                }
            /* case
FramebufferErrorCode.GL_FRAMEBUFFER_INCOMPLETE_DUPLICATE_ATTACHMENT_E
XT:
                 {
                     Trace.WriteLine("FBO: An object has been
attached to more than one attachment point.");
                     break;
                 }*/
            case
FramebufferErrorCode.FramebufferIncompleteDimensionsExt:
                {
                    Trace.WriteLine( "FBO: Attachments are of
different size. All attachments must have the same width and height."
);
                    break;
                }
            case
FramebufferErrorCode.FramebufferIncompleteFormatsExt:
                {
                    Trace.WriteLine( "FBO: The color attachments have
different format. All color attachments must have the same format."
);
                    break;
                }
            case
FramebufferErrorCode.FramebufferIncompleteDrawBufferExt:
                {
                    Trace.WriteLine( "FBO: An attachment point
referenced by GL.DrawBuffers() doesn’t have an attachment." );
                    break;
                }
```

```
            case
FramebufferErrorCode.FramebufferIncompleteReadBufferExt:
                {
                    Trace.WriteLine( "FBO: The attachment point
referenced by GL.ReadBuffers() doesn't have an attachment." );
                    break;
                }
            case FramebufferErrorCode.FramebufferUnsupportedExt:
                {
                    Trace.WriteLine( "FBO: This particular FBO
configuration is not supported by the implementation." );
                    break;
                }
            default:
                {
                    Trace.WriteLine( "FBO: Status unknown. (yes, this
is really bad.)" );
                    break;
                }
            }
            return false;
        }
```

Geometry

These pages of the book discuss how to define, reference and draw geometric Objects using OpenGL.

Focus is on storing the Geometry directly in Vertex Buffer Objects (VBO), for using Immediate Mode please refer to the red book.

1. The Vertex

A **Vertex** (pl. Vertices) specifies a number of Attributes associated with a single Point in space. In the fixed-function environment a Vertex commonly includes Position, Normal, Color and/or Texture Coordinates. The only Attribute that is not optional an must be specified is the Vertex's Position, usually consisting of 3 float.
In Shader Program driven rendering it is also possible to specify custom Vertex Attributes which are previously unknown to OpenGL, such as Radius or Bone Index and Weight for Skeletal Animation. For the sake of simplicity we'll re-create one of the Vertex formats OpenGL already knows, namely InterleavedArrayFormat.T2fN3fV3f. This format contains `2 float` for Texture Coordinates, `3 float` for the Normal direction and `3 float` to specify the Position.

Thanks to the included Math-Library in OpenTK, we're allowed to specify an arbitrary Vertex struct for our requirements, which is much more elegant to handle than a `float[]` array.

```
[StructLayout(LayoutKind.Sequential)]
struct Vertex
{ // mimic InterleavedArrayFormat.T2fN3fV3f
  public Vector2 TexCoord;
  public Vector3 Normal;
```

```
  public Vector3 Position;
}
```

This leads to a Vertex consisting of 8 `float`, or 32 `byte`. We can now declare an Array of Vertices to describe multiple Points and allow easy indexing/referencing them.

```
Vertex[] Vertices;
```

The Vertex-Array Vertices can now be created and filled with data. Addressing elements is as convenient as in the following example:

```
Vertices = new Vertex[ n ]; // -1 < i < n  (Remember that arrays
start at Index 0 and end at Index n-1.)

// examples how to assign values to the Vector's components:
Vertices[ i ].Position = new Vector3( 2f, -3f, .4f ); // create a new
Vector and copy it to Position.
Vertices[ i ].Normal = Vector3.UnitX; // this will copy Vector3.UnitX
into the Normal Vector.
Vertices[ i ].TexCoord.X = 0.5f; // the Vectors are structs, so the
new keyword is not required.
Vertices[ i ].TexCoord.Y = 1f;

// Ofcourse this also works the other way around, using the Vectors
as the source.
Vector2 UV = Vertices[ i ].TexCoord;
```

An **Index** is simply a `byte`, `ushort` or `uint`, referencing an element in the Vertices Array. So if we decide to draw a single Vertex 100 times at the same spot, instead of storing 100 times the same Vertex in Vertices, we can reference it 100 times from the Indices Array:

```
uint[] Indices;
```

Basically the Indices Array is used to describe the primitives and the Vertex Array is used to declare the corner points.

We can also use collections to store our Vertices, but it's recommended you stick with a simple Array to make sure your Indices are valid at all times.
Now the Vertices and Indices Arrays can be used to describe the edges of any Geometric Pritimitve Type.

Once the Arrays are filled with data it can be drawn in Immediate Mode, as Vertex Array or sent into a Vertex Buffer Object.

2. Geometric Primitive Types

OpenGL requires you to specify the Geometric Primitive Type of the Vertices you wish to draw. This is usually expected when you begin drawing in either Immediate Mode (GL.Begin), GL.DrawArrays or GL.DrawElements.

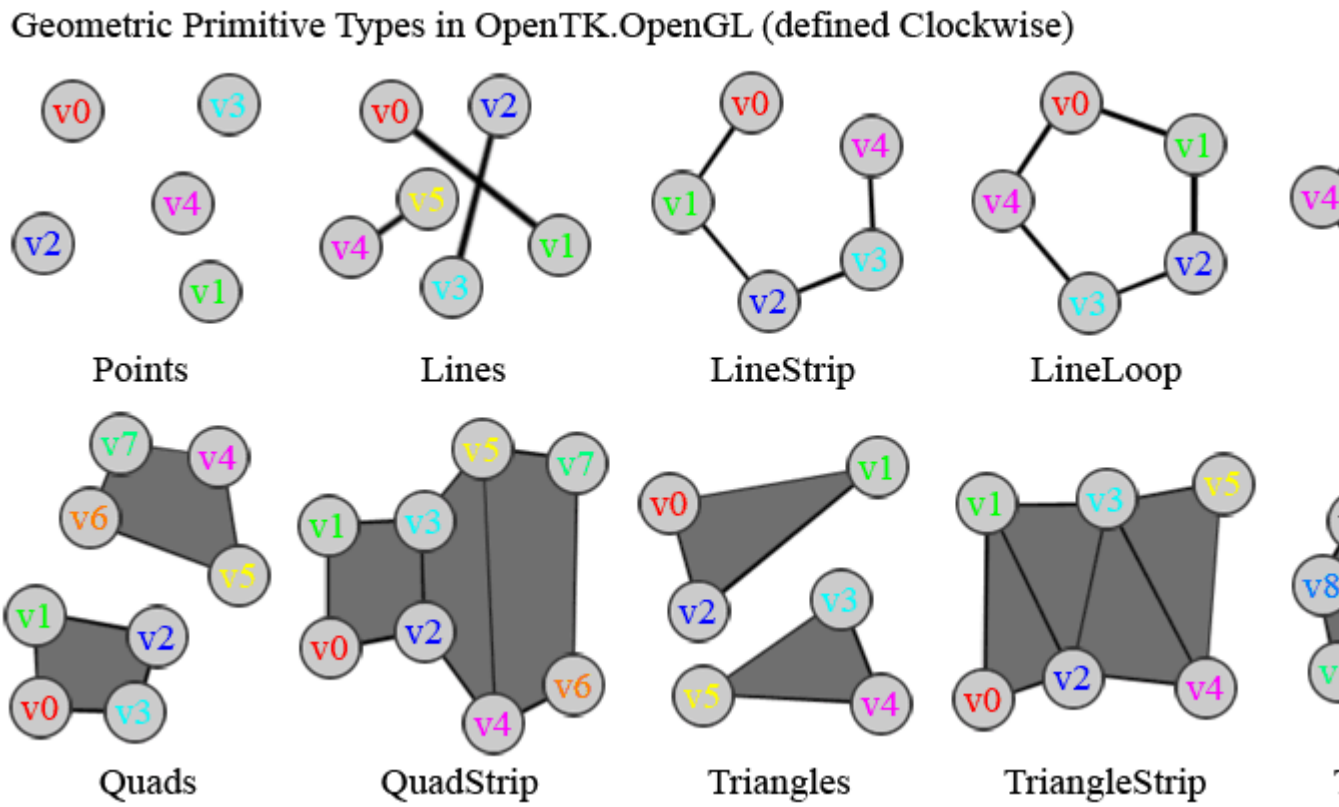


Fig. 1: In the above graphic all valid Geometric Primitive Types are shown, their winding is Clockwise (irrelevant for Points and Lines).

This is important, because drawing a set of Vertices as Triangles, which are internally set up to be used with Quads, will result only in garbage being displayed.
Examine Figure 1, you will see that v3 in a Quad is used to finish the shape, while Triangles uses v3 to start the next shape. The next drawn Triangle will be v3, v4, v5 which isn't something that belongs to any surface, if the Vertices were originally intended to be drawn as Quads.

However Points and Lines are an Exception here. You can draw every other Geometric Primitive Type as Points, in order to visualize the Vertices of the Object. Some more possibilities are:

- QuadStrip, TriangleStrip and LineStrip can be interchanged, if the source data isn't a LineStrip.
- Quads can be drawn as Lines with the restriction that there are no lines between v1, v2 and v3, v0.
- Polygon can be drawn as LineLoop
- TriangleFan can be drawn as Polygon or LineLoop

The smallest common denominator for all filled surfaces (i.e. no Points or Lines) is the Triangle. This Geometric Primitive Type has the special attribute of always being planar and is currently the best way to describe a 3D Object to GPU hardware.
While OpenGL allows to draw Quads or Polygons aswell, it is quite easy to run into lighting problems if the surface is not perfectly planar. Internally, OpenGL breaks Quads and Polygons into Triangles, in order to rasterize them.

1. **Points**
   Specifies 1 Point per Vertex v, thus this is usually only used with GL.DrawArrays().
   n Points = Vertex * (1n);
2. **Lines**
   Two Vertices form a Line.
   n Lines = Vertex * (2n);
3. **LineStrip**
   The first Vertex issued begins the LineStrip, every consecutive issued Vertex marks a joint in the Line.
   n Line Segments in the Strip = Vertex * (1+1n)
4. **LineLoop**
   Same as LineStrip, but the very first and last issued Vertex are automatically connected by an extra Line segment.
   n Line Segments in the Loop = Vertex * (1n);
5. **Polygon**
   Note that the first and the last Vertex will be connected automatically, just like LineLoop.
   Polygon with n Edges = Vertex * (1n);
   Note: This primitive type should really be avoided whenever possible, basically the Polygon will be split to Triangles in the end anyways. Like Quads, polygons must be planar or be displayed incorrectly. Another Problem is that there is only 1 single Polygon in a begin-end block, which leads to multiple draw calls when drawing a mesh, or using the Extensions GL.MultiDrawElements or GL.MultiDrawArrays.
6. **Quads**
   Quads are especially useful to work in 2D with bitmap Images, since those are typically rectangular aswell. Care has to be taken that the surface is planar, otherwise the split into Triangles will become visible.
   n Quads = Vertex * (4n);
7. **QuadStrip**
   Like the Triangle-strip, the QuadStrip is a more compact representation of a sequence of connected Quads.
   n Quads in Quadstrip = Vertex * (2+2n);
8. **Triangles**
   This way to represent a mesh offers the most control over how the Triangles are sorted, a Triangle always consists of 3 Vertex.
   n Triangles = Vertex * (3n);
   Note: It might look like an inefficient brute force approach at first, but it has it's advantages over TriangleStrip. Most of all, since you are not required to supply Triangles in sequenced strips, it is possible to arrange Triangles in a way that makes good use of the Vertex Caches. If the Triangle you currently want to draw shares an edge with one of the Triangles that have been recently drawn, you get 2 Vertices, that are stored in the Vertex Cache, almost for free. This is basically the same what stripification does, but you are not restricted to a certain Direction and forced to insert degenerated Triangles.
9. **TriangleStrip**
   The idea behind this way of drawing is that if you want to represent a solid and closed Object, most neighbour Triangles will share 2 Vertices (an edge). You start by defining the initial Triangle (3 Vertices) and after that every new

Triangle will only require a single new Vertex for a new Triangle.
n Triangles in Strip = Vertex * (2+1n);
Note: While this primitive type is very useful for storing huge meshes (2+1n Vertices per strip as opposed to 3n for BeginMode.Triangles), the big disadvantage of TriangleStrip is that there is no command to tell OpenGL that you wish to start a new strip while inside the glBegin/glEnd block. Ofcourse you can glEnd(); and start a new strip, but that costs API calls. A workaround to avoid exiting the begin/end block is to create 2 or more degenerate Triangles (you can imagine them as Lines) at the end of a strip and then start the next one, but this also comes at the cost of processing Triangles that will inevitably be culled and aren't visible. Especially when optimizing an Object to be in a Vertex Cache friendly layout, it is essential to start new strips in order to reuse Vertices from previous draws.

10. **TriangleFan**
    A fan is defined by a center Vertex, which will be reused for all Triangles in the Fan, followed by border Vertices. It is very useful to represent convex n-gons consisting of more than 4 vertices and disc shapes, like the caps of a cylinder.

When looking at the graphic, Triangle- and Quad-strips might look quite appealing due to their low memory usage. They are beneficial for certain tasks, but Triangles are the best primitive type to represent an arbitrary mesh, because it's not restricting locality and allows further optimizations. It's just not realistic that you can have all your 3D Objects in Quads and OpenGL will split them internally into Triangles anyway. 3 `ushort` per Triangle isn't much memory, and still allows to index 64k unique Vertex in a mesh, the number of Triangles can be much higher. Don't hardwire BeginMode.Triangles into your programs though, for example Quads are very commonly used in orthographic drawing of UI Elements such as Buttons, Text or Sprites.

Should TriangleStrip get an core/ARB command to start a new strip within the begin/end block (only nVidia driver has such an Extension to restart the primitive) this might change, but currently the smaller data structure of the strip does not make up for the performance gains a Triangle List gets from Vertex Cache optimization. Ofcourse you can experiment with the GL.MultiDraw Extension mentioned above, but using it will break using other Extensions such as DirectX 10 instancing.

3.a Vertex Buffer Objects

Introduction
The advantage of VBO (Vertex Buffer Objects) is that we can tell OpenGL to store information used for drawing - like Position, Colors, Texture Coordinates and Normals - directly in the Video-card's Memory, rather than storing it in System Memory and pass it to the graphics Hardware every time we wish to draw it. While this has been already doable with Display Lists before, VBO has the advantage that we're able to retrieve a Pointer to the data in Video Memory and read/write directly to it, if necessary. This can be a huge performance boost for dynamic meshes and is for years the best overall solution for storing - both, static and dynamic - Meshes.

Creation
Handling VBOs is very similar to handling Texture objects, we can generate&delete handles, bind them or fill them with data. For this tutorial we will need 2 objects, one VBO containing all Vertex information (Texture, Normal and Position in this example case) and an IBO (Index Buffer Object) referencing Vertices from the VBO to form Triangles. This has the advantage that, when we have uploaded the data to the VBO/IBO later on, we can draw the whole mesh with a single GL.DrawElements call.

First we acquire two Objects to use:

```
uint[] VBOid = new uint[ 2 ];
GL.GenBuffers( 2, out VBOid );
```

Although it is unlikely, OpenGL could complain that it ran out of memory or that the extension is not supported, it should be checked with GL.GetError. If everything went smooth we have 2 objects to work with available now.

Delete
The OpenGL driver should clean up all our mess when it deletes the render context, it's always a good idea to clean up on your own where you can. We remove the objects we reserved at the buffer creation by calling:

```
GL.DeleteBuffers( 2, ref VBOid );
```

Binding
To select which object you currently want to work with, simply bind the handle to either BufferTarget.ArrayBuffer or BufferTarget.ElementArrayBuffer. The first is used to store position, uv, normals, etc. (named VBO) and the later is pointing at those vertices to define geometry (named IBO).

```
GL.BindBuffer( BufferTarget.ArrayBuffer, VBOid[ 0 ] );
GL.BindBuffer( BufferTarget.ElementArrayBuffer, VBOid[ 1 ] );
```

It is not required to bind a buffer to both targets, for example you could store only the vertices in the VBO and keep the indices in system memory. Also, the two objects are not tied together in any way, for example you could build different triangle lists for BufferTarget.ElementArrayBuffer to implement LOD on the same set of vertices, simply by binding the desired element array.

Theres two important things to keep in mind though:

1) While working with VBOs, GL.EnableClientState(EnableCap.VertexArray); must be enabled. if using Normals, GL.EnableClientState(EnableCap.NormalArray), just like classic Vertex Arrays.

2) All Vertex Array related commands will be used on the currently bound objects until you explicitly bind zero '0' to disable hardware VBO.

```
GL.BindBuffer( BufferTarget.ArrayBuffer, 0 );
GL.BindBuffer( BufferTarget.ElementArrayBuffer, 0 );
```

Passing Data
There are several ways to fill the object's data, we will focus on using GL.BufferData and directly writing to video memory. The third option would be GL.BufferSubData which is quite straightforward to use once you are familiar with GL.BufferData.

1. **GL.BufferData**
   We will start by preparing the IBO, it would not make a difference if we set up the VBO first, we simply start with the shorter one.

   We make sure the correct object is bound (it is not required to do this, if the buffer is already bound. Just here to clarify on which object we currently work on)

   ```
   GL.BindBuffer( BufferTarget.ElementArrayBuffer, VBOid[ 1 ] );
   ```

   In the example application ushort has been used for Indices, because 16 Bits [0..65535] are more available Vertices than used by most real-time rendered meshes, however the mesh could index way more Vertices using a type like uint. Using `ushort`, OpenGL will store this data as 2 Bytes per index, saving memory compared to a 4 Bytes UInt32 per index.

   The function GL.BufferData's first parameter is the target we want to use, the second is the amount of memory (in bytes) we need allocated to hold all our data. The third parameter is pointing at the data we wish to send to the graphics card, this can be IntPtr.Zero and you may send the data at a later stage with GL.MapBuffer (more about this later). The last parameter is an optimization hint for the driver, it will place your data in the best suited place for your purposes.

   ```
   GL.BufferData( BufferTarget.ElementArrayBuffer, (IntPtr) (
   Indices.Length * sizeof( ushort ) ), Indices,
   BufferUsageHint.StaticDraw );
   ```

   That's all, OpenGL now has a copy of Indices available and we could dispose the array, assuming we have the Index Count of the array stored in a variable for the draw call later on.

   Now that we've stored the indices in an IBO, the Vertices are next. Again, we make sure the binding is correct, give a pointer to the Vertex count, and finally the usage hint.

   ```
   GL.BindBuffer( BufferTarget.ArrayBuffer, VBOid[ 0 ] );
   GL.BufferData( BufferTarget.ArrayBuffer, (IntPtr) (
   Vertices.Length * 8 * sizeof( float ) ), Vertices,
   BufferUsageHint.StaticDraw );
   ```

   There's a table at the bottom of this page, explaining the options in the enum BufferUsageHint in more detail.

2. **GL.MapBuffer / GL.UnmapBuffer**

While the first described technique to pass data into the objects required a copy of the data in system memory, this alternative will give us a pointer to the video memory reserved by the object. This is useful for dynamic models that have no copy in client memory that could be used by GL.BufferData, since you wish to rebuild it every single frame (e.g. fully procedural objects, particle system).

First we make sure that we got the desired object bound and reserve memory, the pointer towards the Indices is actually IntPtr.Zero, because we only need an empty buffer.

```
GL.BindBuffer( BufferTarget.ElementArrayBuffer, VBOid[ 0 ] );
GL.BufferData( BufferTarget.ElementArrayBuffer, (IntPtr) (
Indices.Length * sizeof( ushort ) ), IntPtr.Zero,
BufferUsageHint.StaticDraw );
```

Note that you should change BufferUsageHint.StaticDraw properly according to what you intend to do with the Data, there's a table at the bottom of this page. Now we're able to request a pointer to the video memory.

```
IntPtr VideoMemoryIntPtr =
GL.MapBuffer(BufferTarget.ElementArrayBuffer,
BufferAccess.WriteOnly);
```

Valid access flags for the pointer are BufferAccess.ReadOnly, BufferAccess.WriteOnly or BufferAccess.ReadWrite, which help the driver understand what you're going to do with the data. Note that the data's object is locked until we unmap it, so we want to keep the timespan over which we use the pointer as short as possible. We may now write some data into the buffer, once we're done we must release the lock.

```
unsafe
{
  fixed ( ushort* SystemMemory = &Indices[0] )
  {
    ushort* VideoMemory = (ushort*)
VideoMemoryIntPtr.ToPointer();
    for ( int i = 0; i < Indices.Length; i++ )
      VideoMemory[ i ] = SystemMemory[ i ]; // simulate what
GL.BufferData would do
  }
}
GL.UnmapBuffer( BufferTarget.ElementArrayBuffer );
```

The pointer is now invalid and may not be stored for future use, if we wish to modify the object again, we have to call GL.MapBuffer again.

Further reading
Visit this link in order to tell OpenGL about the composition of your Vertex data, and this link for drawing the data.

Optimization:

One hint from the nVidia whitepaper was regarding the situation, if we want to update all data in the buffer object by using GL.MapBuffer and not retrieve any of the old data. Although this is a bad idea, because mapping the buffer is a more expensive operation than just calling GL.BufferData, it might be necessary in cases where you have no copy of the data in system memory, but build it on the fly. The solution to making this somewhat efficient is first calling GL.BufferData with a IntPtr.Zero again, which tells the driver that the old data isn't valid anymore. Calling GL.MapBuffer will return a new pointer to a valid memory location of the requested size to write to, while the old data will be cleaned up once it's not used in any draw operations anymore.

Also note that either reading from a VBO or wrapping it into a Display List is very slow and should both be avoided.

Table 1:
BufferUsageHint.**Static...** Assumed to be a 1-to-n update-to-draw. Means the data is specified once (during initialization).
BufferUsageHint.**Dynamic...** Assumed to be a n-to-n update-to-draw. Means the data is drawn multiple times before it changes.
BufferUsageHint.**Stream...** Assumed to be a 1-to-1 update-to-draw. Means the data is very volatile and will change every frame.

...Draw Means the buffer will be used to sending data to GPU. video memory (Static|StreamDraw) or AGP (DynamicDraw)
...Read Means the data must be easy to access, will most likely be system or AGP memory.
...Copy Means we are about to do some ..Read and ..Draw operations.

3.b Attribute Offsets and Strides

Setting Strides and Offsets for Vertex Arrays and VBO

There are 2 ways to tell OpenGL in which layout the Vertices are stored:

1. **GL.InterleavedArrays()**
   What GL.InterleavedArrays does is enable/disable the required client states for OpenGL to interpret our passed data, the first parameter tells that we have 2 floats for Texture Coordinates (T2f), 3 floats for Normal (N3F) and 3 floats for position (V3F). The second parameter is the stride that will be jumped to find the second, third, etc. set of texcoord/normal/position values. Since our Vertices are tighly packed, no stride (zero) is correct. The last parameter should point at Indices, but we already sent them to the VBO in video memory, no need to point at them again:

   ```
   GL.InterleavedArrays( InterleavedArrayFormat.T2fN3fV3f, 0, null
   );
   ```

   This command has the advantage that it's very obvious to the OpenGL driver what layout of data we have supplied, and it may be possible for the driver to

optimize the memory. Remember that GL.InterleavedArrays will change states, if you manually disable EnableCap.VertexArray, EnableCap.NormalArray, EnableCap.TextureCoordArray or changing GL.VertexPointer, GL.NormalPointer or GL.TexCoordPointer (after calling GL.InterleavedArrays and before calling GL.DrawElements) make sure to enable them again or you won't see anything.

2. **Setting the offsets/strides manually**
   For the Vertex format InterleavedArrayFormat.T2fN3fV3f, the correct pointer setup is:

```
3. GL.TexCoordPointer( 2, TexCoordPointerType.Float, 8 * sizeof(
   float ), (IntPtr) ( 0 ) );
4. GL.NormalPointer( NormalPointerType.Float, 8 * sizeof( float ),
   (IntPtr) ( 2 * sizeof( float ) ) );
   GL.VertexPointer( 3, VertexPointerType.Float, 8 * sizeof( float
   ), (IntPtr) ( 5 * sizeof( float ) ) );
```

   1. The first parameter is the number of components to describe that attribute. For GL.NormalPointer this is always 3 components, but is variable for Texture coordinates and Position (and Color).
   2. The second parameter is the type of the components.
   3. The third parameter is the number of bytes of the Vertex struct. This stride is used to define at which offset the next Vertex begins.
   4. The last parameter indicates the byte offset of the first appearance of the attribute, this makes perfect sense if you recall the layout of our Vertex struct.

      Byte 0-7 are used for the Texture Coordinates, Byte 8-19 for the Normal and Byte 20-31 for the Vertex Position.

3.c Vertex Arrays

Although it's really not recommended using them (besides for compiling to a Display List), Vertex Arrays can be safely used like this:

pseudocode:

```
float[] Positions;
float[] Normals;

unsafe
{
    fixed (float* PositionsPointer = Positions)
    fixed (float* NormalsPointer = Normals)
    {
        GL.NormalPointer(..., NormalsPointer);
        GL.VertexPointer(..., PositionsPointer);
        GL.DrawArrays(...);
        GL.Finish();    // Force OpenGL to finsh drawing while the
arrays are still pinned.
    }
}
```

You must use the unsafe `float*` overloads for this to work, **not** the object overloads (which pin internally)

The important bit is that the pin between GL.*Pointer and GL.Draw* may not be released. For arrays smaller than 85000 Bytes it is also important to call GL.Finish in order for the CPU to wait until the GPU is done reading all pinned data. You might get away without the GL.Finish command for a very short VA or ones larger than 85000 Bytes, but once the pin is released the Garbage Collector is allowed to move or cleanup your array. This leads to difficult to trace access violations with medium sized VA (without waiting for OpenGL to finish processing it) and will randomly occur at frames where the GC kicks in.

GL.Finish will obviously kill performance, since you have a blocking statement executed, which forces CPU and GPU to work in sync and not taking advantage of parallel processing. That's why it's recommended to use Display Lists or Vertex Buffer Objects instead.

Please visit this discussion in the forum for more information.

4. Vertex Array Objects

Vertex Array Objects (abbreviation: VAO) are storing vertex attribute setup and VBO related state. This allows to reduce the number of OpenGL calls when drawing from VBOs, because all attribute declaration and pointer setup only needs to be done once (ideally at initialization time) and is afterwards stored in the VAO for later re-use.

But before a VAO can be bound, a handle must be generated:

```
void GL.GenVertexArrays( uint[] );
void GL.DeleteVertexArrays( uint[] );
bool GL.IsVertexArray( uint );
```

Binding a VAO is as simple as

```
void GL.BindVertexArray( uint );
```

The currently bound VAO records state set by the following commands:

```
GL.EnableVertexAttribArray()
GL.DisableVertexAttribArray()
GL.VertexAttribPointer()
GL.VertexAttribIPointer()
```

Indirectly it also saves the state set by GL.BindBuffer() at the point of time when GL.VertexAttribPointer() was called. A more technical description can be found at the OpenGL Wiki.

5. Drawing

In order to tell OpenGL to draw primitives for us, there's basically two ways to go:

1. Immediate Mode, as in specifying every single Vertex manually.
2. Vertex Buffers (or Vertex Arrays), drawing a whole Mesh with a single Command.

In order for all GL.Draw*-Functions to output geometric Primitives, EnableCap.VertexArray must be enabled first.

1. **Immediate Mode**
   Hereby every single Vertex we wish to draw has to be issued manually.

```
// GL.DrawArrays behaviour
GL.Begin( BeginMode.Points );
for ( uint i = 0; i < Vertices.Length; i++ )
{
    GL.TexCoord2( Vertices[ i ].TexCoord );
    GL.Normal3( Vertices[ i ].Normal );
    GL.Vertex3( Vertices[ i ].Position );
}
GL.End( );

// GL.DrawElements behaviour
GL.Begin( BeginMode.Points );
for ( uint i = 0; i < Indices.Length; i++ )
{
    GL.TexCoord2( Vertices[ Indices[ i ] ].TexCoord );
    GL.Normal3( Vertices[ Indices[ i ] ].Normal );
    GL.Vertex3( Vertices[ Indices[ i ] ].Position );
}
GL.End( );
```

20. **GL.DrawArrays(BeginMode, int First, int Length)**
    This Command is used together with Vertex Arrays or Vertex Buffer Objects, see setting Attribute Pointers. This line will automatically draw all Vertex contained in Vertices in order of appearance in the Array.

```
GL.DrawArrays( BeginMode.Points, 0, Vertices.Length );
```

21. **GL.DrawElements(BeginMode, int Length, DrawElementsType, object)**
    Like DrawArrays this Command is used together with Vertex Arrays or VBO. It uses an unsigned Array (`byte, ushort, uint`) to Index the Vertex Array. This is particularly useful for 3D Models where the Triangles describing the surface share Edges and Vertices.

```
GL.DrawElements( BeginMode.TriangleStrip, Indices.Length,
DrawElementsType.UnsignedInt, Indices );
```

22. **GL.DrawRangeElements(BeginMode, int Start, int End, int Count, DrawElementsType, object)**
    This function behaves largely like GL.DrawElements, with the change that you may specify a starting index rather than starting at 0.
    Start and End are the smallest and largest Array-Index into Indices. Count is the number of Elements to render.

```
// behaviour equal to GL.DrawElements
```

```
GL.DrawRangeElements( BeginMode.TriangleStrip, 0,
Indices.Length-1, Indices.Length, DrawElementsType.UnsignedInt,
Indices );
```

24. **GL.DrawArraysInstanced(BeginMode, int First, int Length, int primcount)**
    This function is only available on DX10 hardware and basically behaves like this:

```
25.  for (int gl_InstanceID=0; gl_InstanceID < primcount;
gl_InstanceID++ )
   GL.DrawArrays( BeginMode, First, Length );
```

    gl_InstanceID is a uniform variable available to the Vertex Shader, which (in conjunction with GL.UniformMatrix) can be used to assign each Instance drawn it's own unique Orientation Matrix.

26. **GL.DrawElementsInstanced(BeginMode, int Length, DrawElementsType, object, int primcount)**
    This function is only available on DX10 hardware and internally unrolls into:

```
27.  for (int gl_InstanceID=0; gl_InstanceID < primcount;
gl_InstanceID++ )
   GL.DrawElements( BeginMode, Length, DrawElementsType, Object
);
```

    gl_InstanceID is a uniform variable available to the Vertex Shader, which (in conjunction with GL.UniformMatrix) can be used to assign each Instance drawn it's own unique orientation Matrix.

Extension References
http://www.opengl.org/registry/specs/EXT/draw_range_elements.txt
http://www.opengl.org/registry/specs/EXT/draw_instanced.txt
http://www.opengl.org/registry/specs/EXT/multi_draw_arrays.txt

5.b Drawing Optimizations

This page is just giving a starting point for optimizations, the links below provide more in-depth information.

- Make sure there are no OpenGL errors. Any error usually kills the framerate.
- Enable backface culling with GL.Enable(EnableCap.CullFace) rather than relying only on the Z-Buffer.
- If you're drawing a scene that covers the whole screen each frame, only clear depth buffer but not the color buffer.
- Organize drawing in a way that requires as few state changes as possible. Don't enable states that are not needed.
- Use GL.Hint(...) wherever applicable.
- Avoid Immediate Mode or Vertex Arrays in favor of Display Lists or Vertex Buffer Objects.
- Use GL.DrawRangeElements instead of GL.DrawElements for a slight performance gain.

- Take advantage of S3TC Texture compression and Vertex Cache optimizing algorithms.

Links:

http://www.mesa3d.org/brianp/sig97/perfopt.htm
http://ati.amd.com/developer/SDK/AMD_SDK_Samples_May2007/Documentations/...
http://developer.nvidia.com/object/gpu_programming_guide.html
http://www.opengl.org/pipeline/article/vol003_8/
http://developer.apple.com/graphicsimaging/opengl/

Last edit of Links: March 2008

6. OpenTK's procedural objects

Placeholder

http://www.opentk.com/node/649

OpenGL rendering pipeline

Rendering works by projecting 3-dimensional objects to a 2-dimensional plane, so

they can be displayed on a screen. In modern OpenGL the 3D objects are read from Vertex Buffer Objects (VBO) and the resulting image is written to a framebuffer. This page will cover the pipeline operations involved between input and output.

Some of the steps involved are fully programmable (namely: Vertex, Geometry and Fragment Shader) while the rest is hardwired. However even the hardwired logic can be manipulated by setting OpenGL's state machine's toggles, which are shown in the diagram and described in detail in the OpenGL specification.

In order to begin drawing, A Vertex and Fragment Shader are required and OpenGL must know about the 3D object, which is done by using VBO (and optionally VAO).

The Vertex Shader is responsible for transforming each Vertex from Object Coordinates into Clip Coordinates.
The primitive assembly will use the resulting Clip Coordinates to create geometric primitives, which are then divided by the vertex' w-component (perspective divide) and clipped against the [-1.0..+1.0] range of normalized device-coordinate space (NDC).
As a final step, the viewport application will offset&scale the normalized device-coordinates to window coordinates.

The resulting transformed geometric primitive types can now be rasterized into fragments. Each fragment receives interpolated vertex shader data from the primitive it belongs to, which is at least position and depth. The fragment shader's output must be either gl_FragColor, gl_FragData[] or set by GL.BindFragDataLocation(). This output is called a "fragment", which is a candidate to become a pixel in the framebuffer. Before this can happen, the fragment muss pass a series of tests called the Fragment Operations.

There is one noteworthy special case found in some modern hardware. The functionality is called "Early-Z" or "HyperZ". After rasterization of the primitive, the resulting Z is used to discard fragments even before the fragment shader is executed. This functionality is not exposed to OpenGL and works behind the scenes. In the diagram to the left, it would belong between the Triangle, Line or Point rasterization, and the fragment shader.

Also note: This diagram targets the OpenGL 3.2 pipeline, but contains a few commands which belong to the ARB_compatibility extension and may be unavailable.

Fragment Operations

A Fragment is a candidate to become a pixel in the framebuffer. For every fragment, OpenGL applies a series of tests in order to eliminate the fragment early to avoid updating the framebuffer.

Most of these tests can be toggled through GL.Enable/Disable, the pages below cover this in more detail. However the most in-depth description of the functionality can only be found in the official OpenGL specification.

The tests are executed from top to bottom, if a fragment did not pass an early test, later tests are ignored. I.e. If the fragment does not pass the Scissor Test, there would be no point in determining whether Depth Test passes or not.

01. Pixel Ownership Test

Citation with minor modifications. Cannot explain it better.

"GL 3.1 spec" wrote:

This test is used to determine if the pixel at the current location in the framebuffer is currently owned by the GL context. If it is not, the window system decides the fate the incoming fragment. Possible results are that the fragment is discarded or that some subset of the subsequent per-fragment operations are applied to the fragment. This test allows the window system to control the GL's behavior, for instance, when a GL window is obscured.

If the draw framebuffer is a framebuffer object, the pixel ownership test always passes, since the pixels of framebuffer objects are owned by the GL, not the window system. If the draw framebuffer is the default framebuffer, the window system controls pixel ownership.

02. Scissor Test

The Scissor Test is used to limit drawing to a rectangular region of the viewport. When enabled, only fragments inside the rectangle are passed to later pipeline stages.

The ScissorTest can be enabled or disabled using EnableCap.ScissorTest

```
GL.Enable( EnableCap.ScissorTest );
GL.Disable( EnableCap.ScissorTest ); // default
```

Only a single command is related to the ScissorTest, GL.Scissor(x, y, width, height). By default the parameters are set to cover the whole window.

- X and Y are used to specify the lower-left corner of the rectangle.
- Width is used to specify the horizontal extension of the rectangle.
- Height is used to specify the vertical extension of the rectangle.

State Queries

To determine whether ScissorTest is enabled or disabled, use `Result = GL.IsEnabled( EnableCap.ScissorTest );`

The values set by GL.Scissor() can be queried by `GL.GetInteger( GetPName.ScissorBox, ... ); // returns an array`

03. Multisample Fragment Operations (WIP)

Multisampling is designed to counter the effects of aliasing at the edges of a primitive, when it is rasterized into fragments. Multisampling can be also applied to transparent textures, like wire fences, blades of grass or the leaves of trees. In this case, it is called 'alpha-to-coverage' and replaces the legacy alpha test.

A multisample buffer contains multiple samples per pixel, with each sample having it's own color, depth and stencil values. The term 'coverage' refers to a bitmask that is used to determine which of these samples will be updated: a coverage value of 1 indicates that the relevant sample will be updated; a value of 0 indicates it will be left untouched.

However, when `EnableCap.SampleAlphaToCoverage` is used, the coverage is obtained by interpreting the alpha as a percentage: an alpha of 0.0 means that no samples are covered, while a value of 1.0 indicates that all samples are covered. For example, a multisample buffer with 4 samples per pixel and an Alpha value of 0.5 indicates that half of the samples are covered (their coverage bit is 1) and two are not covered (coverage bit is 0).

"figure out whether this is true" wrote:

The coverage bitmask of incoming fragments can be set in a Fragment Shader with the variable gl_Coverage.
http://www.humus.name/index.php?page=Comments&ID=230

To enable alpha-to-coverage, enable multisampling (`GL.Enable(EnableCap.Multisample)`) and make sure that `GL.GetInteger(GetPName.SampleBuffers, out buffers)` is 1. If `EnableCap.Multisample` is disabled but `GetPName.SampleBuffers` is 1, alpha-to-coverage will be disabled.

There are three OpenGL states related to alpha-to-coverage, they are controlled by `GL.Enable()` and `GL.Disable()`

- **EnableCap.SampleAlphaToCoverage**
  For each sample at the current pixel, the Alpha value is read and used to generate a temporary coverage bitmask which is then combined through a bitwise AND with the fragment's coverage bitmask. Only samples who's bit is set to 1 after the bitwise AND are updated.
- **EnableCap.SampleCoverage**
  Using `GL.SampleCoverage( value, invert )` the temporary coverage bitmask is generated by the value parameter - and if the invert parameter is true it is bitwise inverted - before the bitwise AND with the fragment's coverage bitmask.
- **EnableCap.AlphaToOne**
  Each Alpha value is replaced by 1.0.

GL.SampleCoverage
The values set by the command `GL.SampleCoverage( value, invert )` are only used when EnableCap.SampleCoverage is enabled.

- **value** is a single-precision float used to specify the Alpha value used to create the coverage bitmask.
- **invert** is a boolean toggle to control whether the bitmask is bitwise inverted before the AND operation.

State Queries

The states of EnableCap.Multisample, EnableCap.SampleAlphaToCoverage, EnableCap.SampleCoverage and EnableCap.AlphaToOne can be queried with `Result = GL.IsEnabled( cap )`

The value set by GL.SampleCoverage() can be queried with `GL.GetFloat( GetPName.SampleCoverageValue, ... )`

The boolean set by GL.SampleCoverage() can be queried with `GL.GetBoolean( GetPName.SampleCoverageInvert, ... )`

04. Stencil Test

The Stencil buffer's primary use is to apply a mask to the framebuffer. Simply put, you can think of it as a cardboard stencil where you cut out holes, so you may use a can of spraypaint to paint shapes. The paint will only pass the holes you had cut out and be blocked otherwise by the cardboard. OpenGL's Stencil testing allows you to layer several of these masks over each other.

A more OpenGL related example: In any vehicle simulation the interior of the cockpit is usually masked by a stencil buffer, because it does not have to be redrawn every frame. That way alot of fragments of the outside landscape can be skipped, as they would not contribute to the final image anyway.

In order to use the Stencil buffer, the window-system provided framebuffer - or the Stencil attachment of a FBO - must explicitly contain a logical stencil buffer. If there is no stencil buffer, the fragment is always passed to the next pipeline stage.

For the purpose of clarity in this article, the Stencil Buffer is assumed to be 8 Bit large and able to represent the values 0..255

StencilTest functionality is enabled and disabled with EnableCap.StencilTest

```
GL.Enable( EnableCap.StencilTest );
GL.Disable( EnableCap.StencilTest ); // default
```

The value used by GL.Clear() commands can be set through:

```
GL.ClearStencil( int ); // 0 is the default, Range [0..255]
```

If StencilTest is enabled, writing can be limited by a bitfield through a bitwise AND.

```
GL.StencilMask( bitmask ); // By default all bits are set to 1.
```

Note: The command GL.StencilMaskSeparate() behaves exactly like GL.StencilMask() but allows separate comparison functions for front- and back-facing polygons.

GL.StencilFunc()

The command GL.StencilFunc(func, ref, mask) is used to specify the conditions under which the StencilTest succeeds or fails. It sets the comparison function, reference value and mask for the Stencil Test.

- **ref** is an integer value to compare against. By default this value is 0, range [0 .. 255]
- **mask** is a bitfield which will be used in a bitwise AND. Only the bits which are set to 1 are considered. By default all bits are set to 1.
- **func** of the test can have the following values, the default is StencilFunction.Always.
    - StencilFunction.Always - Test will always succeed.
    - StencilFunction.Never - Test will never succeed.
    - StencilFunction.Less - Test will succeed if (ref & mask) < (pixel & mask)
    - StencilFunction.Lequal - Test will succeed if (ref & mask) <= (pixel & mask)
    - StencilFunction.Equal - Test will succeed if (ref & mask) == (pixel & mask)
    - StencilFunction.Notequal - Test will succeed if (ref & mask) != (pixel & mask)
    - StencilFunction.Gequal - Test will succeed if (ref & mask) >= (pixel & mask)
    - StencilFunction.Greater - Test will succeed if (ref & mask) > (pixel & mask)

    The word 'pixel' means in this case: The value in the Stencil Buffer at the current pixel.

Note: The command GL.StencilFuncSeparate() behaves exactly like GL.StencilFunc() but allows separate comparison functions for front- and back-facing polygons.

GL.StencilOp()

Depending on the result determined by `GL.StencilFunc()`, the command `GL.StencilOp( fail, zfail, zpass )` can be used to decide what action should be taken if the fragment passes the test.

- fail - behavior when StencilTest fails, regardless of DepthTest.
- zfail - behavior when StencilTest succeeds, but DepthTest fails.

- zpass - behavior when both tests succeed, or if StencilTest succeeds and DepthTest is disabled.

The following values are allowed, the default for all operations is StencilOp.Keep

- StencilOp.Zero - set Stencil Buffer to 0.
- StencilOp.Keep - Do not modify the Stencil Buffer.
- StencilOp.Replace - set Stencil Buffer to ref value as specified by last GL.StencilFunc() call.
- StencilOp.Incr - increment Stencil Buffer by 1. It is clamped at 255.
- StencilOp.IncrWrap - increment Stencil Buffer by 1. If the result is greater than 255, it becomes 0.
- StencilOp.Decr - decrement Stencil Buffer by 1. It is clamped at 0.
- StencilOp.DecrWrap - decrement Stencil Buffer by 1. If the result is less than 0, it becomes 255.
- StencilOp.Invert - Bitwise invert. If the Stencil Buffer currently contains the bits 00111001, it is set to 11000110.

Note: The command GL.StencilOpSeparate() behaves exactly like GL.StencilOp() but allows separate comparison functions for front- and back-facing polygons.

State Queries

To determine whether StencilTest is enabled or disabled, use `Result = GL.IsEnabled( EnableCap.StencilTest );`

The bits available in the Stencil Buffer can be queried by `GL.GetInteger( GetPName.StencilBits, ... );`

The value set by GL.ClearStencil() can be queried by `GL.GetInteger( GetPName.StencilClearValue, ... );`

The bitfield set by GL.StencilMask() can be queried by `GL.GetInteger( GetPName.StencilWritemask, ... );`

The state of the Stencil comparison function can be queried with `GL.GetInteger()` and the following parameters:

GetPName.StencilFunc - GL.StencilFunc's parameter 'func'
GetPName.StencilRef - GL.StencilFunc's parameter 'ref'
GetPName.StencilValueMask - GL.StencilFunc's parameter 'mask'

The state of the Stencil operations can be queried with GL.GetInteger and the following parameters:

GetPName.StencilFail - GL.StencilOp's parameter 'fail'
GetPName.StencilPassDepthFail - GL.StencilOp's parameter 'zfail'
GetPName.StencilPassDepthPass - GL.StencilOp's parameter 'zpass'

If the GL.Stencil***Separate() functions have been used, the tokens GetPName.StencilBack*** become available to query the settings for back-facing polygons. With Intellisense you should not have any problems finding them.

Related Extensions for further reading

http://www.opengl.org/registry/specs/EXT/stencil_clear_tag.txt
http://www.opengl.org/registry/specs/EXT/stencil_wrap.txt (promoted to core in GL 2.0)
http://www.opengl.org/registry/specs/ATI/separate_stencil.txt (promoted to core in GL 2.0)
http://www.opengl.org/registry/specs/EXT/stencil_two_side.txt (basically the same functionality as ATI_separate_stencil, not in core though)

05. Depth Test

A commonly used logical buffer in OpenGL is the Depth buffer, often called Z-Buffer. The name was chosen due to X and Y being used to describe horizontal and vertical displacement on the screen, so Z is used to measure the distance perpendicular to the screen.

The general purpose of this buffer is determining whether a fragment is occluded by a previously drawn pixel. I.e. If the fragment in question is further away from the eye than an already existing pixel, the fragment cannot be visible and is discarded.

In order to use the Depth buffer, the window-system provided framebuffer - or the Depth attachment of a FBO - must explicitly contain a logical Depth buffer. If there is no Depth buffer, the fragment is always passed to the next pipeline stage.

DepthTest functionality is enabled and disabled with EnableCap.DepthTest

```
GL.Enable( EnableCap.DepthTest );
GL.Disable( EnableCap.DepthTest ); // default
```

The value used by GL.Clear() commands can be set through:

```
GL.ClearDepth( double ); // 1.0 is the default, Range: [0.0 .. 1.0]
```

If DepthTest is enabled, writing to the Depth buffer can be toggled by a boolean flag.

```
GL.DepthMask( bool ); // true is the default
```

GL.DepthFunc
The command `GL.DepthFunc( func )` is used to specify the comparison method used whether a fragment is closer to the eye than the existing pixel it is compared to.

Function of the test can have the following values, the default is DepthFunction.Less.

- DepthFunction.Always - Test will always succeed.
- DepthFunction.Never - Test will never succeed.

- DepthFunction.Less - Test will succeed if (fragment depth < pixel depth)
- DepthFunction.Lequal - Test will succeed if (fragment depth <= pixel depth)
- DepthFunction.Equal - Test will succeed if (fragment depth == pixel depth)
- DepthFunction.Notequal - Test will succeed if (fragment depth != pixel depth)
- DepthFunction.Gequal - Test will succeed if (fragment depth >= pixel depth)
- DepthFunction.Greater - Test will succeed if (fragment depth > pixel depth)

GL.DepthRange
The command `GL.DepthRange( near, far )` is used to define the minimum (near plane) and maximum (far plane) z-value that is stored in the Depth Buffer. Both parameters are expected to be of double-precision floating-point and must lie within the range [0.0 .. 1.0].

It is allowed to call `GL.DepthRange( 1.0, 0.0 )`, there is no rule that must satisfy (near < far).

For an in-depth explanation how the distribution of z-values in the Depth buffer works, please read Depth buffer - The gritty details.

State Queries

To determine whether DepthTest is enabled or disabled, use `Result = GL.IsEnabled( EnableCap.DepthTest );`

The bits available in the Stencil Buffer can be queried by `GL.GetInteger( GetPName.DepthBits, ... );`

The value set by GL.ClearDepth() can be queried by `GL.GetFloat( GetPName.DepthClearValue, ... );`

The boolean set by GL.DepthMask() can be queried by `GL.GetBoolean( GetPName.DepthWritemask, ... );`

The Depth comparison function can be queried with `GL.GetInteger( GetPName.DepthFunc, ... );`

The Depth range can be queried with `GL.GetFloat( GetPName.DepthRange, ... ); // returns an array`

06. Occlusion Query

Occlusion queries count the number of fragments (or samples) that pass the depth test, which is useful to determine visibility of objects.

If an object is drawn but 0 fragments passed the depth test, it is fully occluded by another object. In practice this means that a simplification of an object is drawn using an occlusion query (for example: A bounding box can be the occlusion substitute for a truck) and only if fragments of the simple object pass the depth test, the complex object is drawn. Please read Conditional Render for a convenient solution.

Note that the simplified object does not actually have to become visible, one can set GL.ColorMask and GL.DepthMask to false for the purpose of the occlusion query. The only GL.Enable/Disable state associated with it is the DepthTest. If DepthTest is disabled all fragments will automatically pass it and the occlusion test becomes pointless.

Occlusion Query handles are generated and deleted similar to other OpenGL handles:

```
uint MyOcclusionQuery;
GL.GenQueries( 1, out MyOcculsionQuery );
GL.DeleteQueries( 1, ref MyOcculsionQuery );
```

The draw commands which contribute to the count must be enclosed with GL.BeginQuery() and GL.EndQuery().

```
GL.BeginQuery( QueryTarget.SamplesPassed, MyOcculsionQuery );
// draw...
GL.EndQuery( QueryTarget.SamplesPassed );
```

It is very important to understand that this process is running asynchronous, by the time the CPU is querying the result of the count the GPU might not be done counting yet. OpenGL provides additional query commands to determine whether the occlusion query result is available, but before it is confirmed to be available any query of the count is not reliable. The following code will get a reliable result.

```
uint ResultReady=0;
while ( ResultReady == 0 )
{
  GL.GetQueryObject( MyOcculsionQuery,
GetQueryObjectParam.QueryResultAvailable, out ResultReady );
}
uint MyOcclusionQueryResult=0;
GL.GetQueryObject( MyOcculsionQuery, GetQueryObjectParam.QueryResult,
out MyOcclusionQueryResult );
// MyOcclusionQueryResult is now reliable.
```

However this is not very efficient to use because the CPU will spin in a loop until the GPU is done counting.

A better approach is to do the occlusion queries in the first frame and do not wait for a result. Instead continue drawing as normal and wait for the next frame, before you check the results of the query. In other words frame n executes the query and frame n + 1 reads back the results.

This approach hides the latency inherent in occlusion queries and improves performance, at the cost of slight visual glitches (an object may become visible one frame later than it should). You can read a very detailed description of this technique on Chapter 29 of GPU Gems 1, which also covers other caveats of occlusion queries.

Conditional Render

The Extension NV_conditional_render adds a major improvement to occlusion queries: it allows a simple `if ( SamplesPassed > 0 )` conditional to decide whether an object should be drawn based on the result of an occlusion query.

This is probably best shown by a simple example, in the given scene there are 3 objects:

- A huge cylinder which acts as occluder. Think of it as a pillar in the center of the "room".
- A small cube which acts as ocludee. Think of it as a box that is anywhere in the "room" but not intersecting the pillar.
- A small sphere which sits ontop of the cube. If the cube is fully occluded by the cylinder, drawing the sphere can be skipped.

Here is some pseudo-code how the implementation looks like.

```
uint MyOcculsionQuery;

public void OnLoad()
{
  GL.GenQueries(1, out MyOcculsionQuery);
  // etc...
  GL.Enable( EnableCap.DepthTest );
}

public void OnUnload()
{
  GL.DeleteQueries(1, ref MyOcculsionQuery);
  // etc...
}

public void OnRenderFrame()
{
  // The cylinder is drawn unconditionally and used as occluder for
the Cube and Sphere
  MyCylinder.Draw();

  // Next, the cube is drawn unconditionally, but the samples which
passed the depth test are counted.
  GL.BeginQuery( QueryTarget.SamplesPassed, MyOcculsionQuery );
  MyCube.Draw();
  GL.EndQuery( QueryTarget.SamplesPassed );

  // depending on whether any sample passed the depth test, the
sphere is drawn.
  GL.NV.BeginConditionalRender( MyOcculsionQuery,
NvConditionalRender.QueryWaitNv );
  MySphere.Draw();
  GL.NV.EndConditionalRender();

  this.SwapBuffers();
}
```

Although the running program might only show a single object on screen (the cylinder), the cube is always drawn too. Only drawing of the sphere might be skipped, depending on the outcome of the occlusion query used for the cube.

Please note that this is not the standard case how to use occlusion query. The most common way to use them is drawing a simple bounding volume (of a more complex object) to determine whether samples passed and only draw the complex object itself, if the bounding volume is not occluded. For example: Drawing a character with skeletal animation is usually expensive, to determine whether it should be drawn at all, a cylinder can be drawn using an occlusion query and the character is only drawn if the cylinder is not occluded.

07. Blending

Without blending, every fragment is either rejected or written to the framebuffer. That behaviour is desireable for opaque objects, but it does not allow rendering of translucent objects. The correct order of operation to draw a simple scene containing a solid table with a transparent glass ontop of it: draw the opaque table first, then enable blending (also set the desired blend equation and factors) and finally the glass is drawn.

Blending is an operation to mix the incoming fragment color (SourceColor) with the color that is currently in the color buffer (DestinationColor). This happens in two stages for each channel of the color buffer:

1. **The factors used in this stage can be controlled with GL.BlendFunc()**
   The SourceColor is multiplied by the SourceFactor.
   The DestinationColor is multiplied by the DestinationFactor.
2. **The equation used in this stage can be controlled with GL.BlendEquation()**
   The results of the above multiplications are then combined together to obtain the final result.

In order to use blending, the logical color buffer must have an Alpha channel. If there is no Alpha channel, or the color buffer uses color-index mode (8 Bit), no blending can occur and behaviour is the same as if blending was disabled.

Blending functionality for all draw buffers is enabled and disabled with EnableCap.Blend

```
GL.Enable( EnableCap.Blend );
GL.Disable( EnableCap.Blend ); // default
```

To enable or disable only a specific buffer if multiple color buffers are attached to the FBO, use

```
GL.Enable( IndexedEnableCap.Blend, index);
GL.Disable( IndexedEnableCap.Blend, index);
```

Where index is used to specify the draw buffer associated with the symbolic constant GL_DRAW_BUFFER(index).

GL.BlendEquation
The command GL.BlendEquation(mode) specifies how the results from stage 1 are

combined with each other. If you wanted to implement this with OpenTK.Math, it would look like this:

```
Color4 SourceColor, // incoming fragment
       DestinationColor,  // framebuffer contents
       SourceFactor, DestinationFactor, // specified by
GL.BlendFunc()
       FinalColor; // the resulting color
```

Please note that for fixed-point color buffers both Colors are clamped to [0.0 .. 1.0] prior to computing the result. Floating-point color buffers are not clamped. Clamping into this range is left out in this code to improve legibility.

BlendEquationMode.Min: When using this parameter, SourceFactor and DestinationFactor are ignored.

```
FinalColor.Red = min( SourceColor.Red, DestinationColor.Red );
FinalColor.Green = min( SourceColor.Green, DestinationColor.Green );
FinalColor.Blue = min( SourceColor.Blue, DestinationColor.Blue );
FinalColor.Alpha = min( SourceColor.Alpha, DestinationColor.Alpha );
```

BlendEquationMode.Max: When using this parameter, SourceFactor and DestinationFactor are ignored.

```
FinalColor.Red = max( SourceColor.Red, DestinationColor.Red );
FinalColor.Green = max( SourceColor.Green, DestinationColor.Green );
FinalColor.Blue = max( SourceColor.Blue, DestinationColor.Blue );
FinalColor.Alpha = max( SourceColor.Alpha, DestinationColor.Alpha );
```

BlendEquationMode.FuncAdd: This is the default.

```
FinalColor.Red = SourceColor.Red*SourceFactor.Red +
DestinationColor.Red*DestinationFactor.Red;
FinalColor.Green = SourceColor.Green*SourceFactor.Green +
DestinationColor.Green*DestinationFactor.Green;
FinalColor.Blue = SourceColor.Blue*SourceFactor.Blue +
DestinationColor.Blue*DestinationFactor.Blue;
FinalColor.Alpha = SourceColor.Alpha*SourceFactor.Alpha +
DestinationColor.Alpha*DestinationFactor.Alpha;
```

BlendEquationMode.FuncSubtract:

```
FinalColor.Red = SourceColor.Red*SourceFactor.Red -
DestinationColor.Red*DestinationFactor.Red;
FinalColor.Green = SourceColor.Green*SourceFactor.Green -
DestinationColor.Green*DestinationFactor.Green;
FinalColor.Blue = SourceColor.Blue*SourceFactor.Blue -
DestinationColor.Blue*DestinationFactor.Blue;
FinalColor.Alpha = SourceColor.Alpha*SourceFactor.Alpha -
DestinationColor.Alpha*DestinationFactor.Alpha;
```

BlendEquationMode.FuncReverseSubtract:

```
FinalColor.Red = DestinationColor.Red*DestinationFactor.Red -
SourceColor.Red*SourceFactor.Red;
```

```
FinalColor.Green = DestinationColor.Green*DestinationFactor.Green -
SourceColor.Green*SourceFactor.Green;
FinalColor.Blue = DestinationColor.Blue*DestinationFactor.Blue -
SourceColor.Blue*SourceFactor.Blue;
FinalColor.Alpha = DestinationColor.Alpha*DestinationFactor.Alpha -
SourceColor.Alpha*SourceFactor.Alpha;
```

If the color buffer is using fixed-point precision, the result in FinalColor is clamped to [0.0 .. 1.0] before it is passed to the next pipeline stage, no clamping occurs for floating-point color buffers.

OpenGL 2.0 and later supports separate equations for the RGB components and the Alpha component respectively. The command GL.BlendEquationSeparate(modeRGB, modeAlpha) accepts the same parameters as GL.BlendEquation(mode).

GL.BlendColor
The command GL.BlendColor(R, G, B, A) is used to specify a constant color that can be used by GL.BlendFunc(). For the scope of this page it is used to define the variable `Color4 ConstantColor;`.

GL.BlendFunc
The command GL.BlendFunc(src, dest) is used to select the SourceFactor (src) and DestinationFactor (dest) in the above equation. By default, SourceFactor is set to BlendingFactorSrc.One and DestinationFactor is BlendingFactorDest.Zero, which gives the same result as if blending were disabled.

- .Zero: Color4 (0.0, 0.0, 0.0, 0.0)
- .One: Color4 (1.0, 1.0, 1.0, 1.0)
- .DstColor: Color4 (DestinationColor.Red, DestinationColor.Green, DestinationColor.Blue, DestinationColor.Alpha)
- .SrcColor: Color4 (SourceColor.Red, SourceColor.Green, SourceColor.Blue, SourceColor.Alpha)
- .OneMinusDstColor: Color4 (1.0-DestinationColor.Red, 1.0-DestinationColor.Green, 1.0-DestinationColor.Blue, 1.0-DestinationColor.Alpha)
- .OneMinusSrcColor: Color4 (1.0-SourceColor.Red, 1.0-SourceColor.Green, 1.0-SourceColor.Blue, 1.0-SourceColor.Alpha)
- .SrcAlpha: Color4 (SourceColor.Alpha, SourceColor.Alpha, SourceColor.Alpha, SourceColor.Alpha)
- .OneMinusSrcAlpha: Color4 (1.0-SourceColor.Alpha, 1.0-SourceColor.Alpha, 1.0-SourceColor.Alpha, 1.0-SourceColor.Alpha)
- .DstAlpha: Color4 (DestinationColor.Alpha, DestinationColor.Alpha, DestinationColor.Alpha, DestinationColor.Alpha)
- .OneMinusDstAlpha: Color4 (1.0-DestinationColor.Alpha, 1.0-DestinationColor.Alpha, 1.0-DestinationColor.Alpha, 1.0-DestinationColor.Alpha)
- .SrcAlphaSaturate: f = min(SourceColor.Alpha, 1.0-DestinationColor.Alpha);
  Color4 (f, f, f, 1.0)
- .ConstantColor: Color4 (ConstantColor.Red, ConstantColor.Green, ConstantColor.Blue, ConstantColor.Alpha)

- .OneMinusConstantColor: Color4 (1.0-ConstantColor.Red, 1.0-ConstantColor.Green, 1.0-ConstantColor.Blue, 1.0-ConstantColor.Alpha)
- .ConstantAlpha: Color4 (ConstantColor.Alpha, ConstantColor.Alpha, ConstantColor.Alpha, ConstantColor.Alpha)
- .OneMinusConstantAlpha: Color4 (1.0-ConstantColor.Alpha, 1.0-ConstantColor.Alpha, 1.0-ConstantColor.Alpha, 1.0-ConstantColor.Alpha)

OpenTK uses the enums BlendingFactorSrc and BlendingFactorDest to narrow down your options what is a valid parameter for src and dest. Not all parameters are valid factors for both, SourceFactor and DestinationFactor. Please refer to the inline documentation for details.

OpenGL 2.0 and later supports separate factors for RGB and Alpha, for source and destination respectively. The command GL.BlendFuncSeparate(srcRGB, dstRGB, srcAlpha, dstAlpha) accepts the same factors as GL.BlendFunc(src, dest).

State Queries

To determine whether blending for all draw buffers is enabled or disabled, use `Result = GL.IsEnabled( EnableCap.Blend );`
To query blending state of a specific draw buffer: `Result = GL.IsEnabled(IndexedEnableCap.Blend, index);`

The selected blend factors can be queried separately for source and destination by using `GL.GetInteger()` with

- GetPName.BlendSrc - set by GL.BlendFunc()
- GetPName.BlendDst - set by GL.BlendFunc()
- GetPName.BlendSrcRgb - set by GL.BlendFuncSeparate()
- GetPName.BlendSrcAlpha - set by GL.BlendFuncSeparate()
- GetPName.BlendDstRgb - set by GL.BlendFuncSeparate()
- GetPName.BlendDstAlpha - set by GL.BlendFuncSeparate()

The selected blend equation can be queried by using `GL.GetInteger()` with

- GetPName.BlendEquation - set by GL.BlendEquation()
- GetPName.BlendEquationRgb - set by GL.BlendEquationSeparate()
- GetPName.BlendEquationAlpha - set by GL.BlendEquationSeparate()

08. sRGB Conversion

This stage of the pipeline is only applied if EnableCap.FramebufferSrgb is enabled and if the color encoding for the framebuffer attachment is sRGB (as in: not linear).

- If those conditions are true, the Red, Green and Blue values after blending are converted into the non-linear sRGB color space.
- If any of those conditions is false, no conversion is applied.

The resulting values for R, G, and B, and the unmodified Alpha form a new RGBA color value. If the color buffer is fixed-point, each component is clamped to the range [0.0 .. 1.0] and then converted to a fixed-point value. The resulting four values are sent to the subsequent dithering operation.

09. Dithering

Dithering is similar to halftoning in newspapers. Only a single color (black) is used in contrast to the paper (white), but due to using patterns the appearance of many shades of gray can be represented. In a similar way, OpenGL can dither the fragment from a high precision color to a lower precision color. I.e. dithering is used to find one or more representable colors to ensure the image shown on the screen is a best-match between the capability of the monitor and the computed image.

This is always needed when working with 8 Bit color-index mode, where only 256 unique colors can be represented, but the image to be drawn is actually calculated with higher precision. Dithering also applies when a RGBA32f color is converted to display on the screen, which is usually RGBA8. In RGBA mode, dithering is performed separately for Red, Green, Blue and Alpha.

Dithering is the only state that is enabled by default, the programmer has no control over how the image is manipulated (the graphics hardware decides which algorithm is used) besides enabling or disabling dithering with EnableCap.Dither.

```
GL.Enable( EnableCap.Dither ); // default
GL.Disable( EnableCap.Dither );
```

State Query

The state of dithering can be queried through `Result = GL.IsEnabled( EnableCap.Dither );`

10. Logical Operations

Before a fragment is written to the framebuffer, a logical operation is applied which uses the incoming fragment values as source (s) and/or those currently stored in the color buffer as destination (d). After the selected operation is completed, destination is overwritten. Logical operations are performed independently for each Red, Green, Blue and Alpha value and if the framebuffer has multiple color attachments, the logical operation is computed and applied separately for each color buffer.

If Logic Op is enabled, OpenGL behaves as if Blending is disabled regardless whether it was previously enabled.

In order to apply the Logicial Operation, use `EnableCap.ColorLogicOp`

```
GL.Enable( EnableCap.ColorLogicOp );
GL.Disable( EnableCap.ColorLogicOp ); // default
```

Note: If you use EnableCap.LogicOp or EnableCap.IndexLogicOp, only indexed color buffers (8 Bit) are affected.

To select the logical operation to be performed, use `GL.LogicOp( op );` where op is by default LogicOp.Copy.

- LogicOp.Clear: 0
- LogicOp.And: s & d
- LogicOp.AndReverse: s & !d
- LogicOp.Copy: s
- LogicOp.AndInverted: !s & d
- LogicOp.Noop: d
- LogicOp.Xor: s XOR d
- LogicOp.Or: s | d
- LogicOp.Nor: !(s | d)
- LogicOp.Equiv: !(s XOR d)
- LogicOp.Invert: !d
- LogicOp.OrReverse: s | !d
- LogicOp.CopyInverted: !s
- LogicOp.OrInverted: !s | d
- LogicOp.Nand: !(s & d)
- LogicOp.Set: all 1's

State Queries
The state of LogicOp can be queried with `Result = GL.IsEnabled( EnableCap.LogicOp );`

Which operation has been set through GL.LogicOp() can be queried with `GL.GetInteger( GetPName.LogicOpMode, ... )`

How to save an OpenGL rendering to disk

You can use the following code to read back an OpenGL rendering to a `System.Drawing.Bitmap`. You can then use the `Save()` method to save this to disk.

Hints:

- Don't forget to call `Dispose()` on the returned `Bitmap` once you are done with it. Otherwise, you will run out of memory rapidly. If you wish to save a video, rather than a single screenshot, consider modifying this method to reuse the same `Bitmap`.
- Call `GrabScreenshot()` from your main rendering thread, i.e. the thread which contains your GraphicsContext.
- Make sure you have bound the correct framebuffer object before calling `GrabScreenshot()`.
- You can improve performance significantly by removing the `bmp.RotateFlip()` call and saving the resulting image as a BMP rather than a

PNG file. This is especially important if you wish to record a video - it is the difference between a real-time recording and a slideshow.
- This code can record 720p/30Hz video relatively easily, given suitable hardware and a little optimization (as outlined above). There are many programs that can encode a stream of consecutive BMP files into a high definition video.

```
using System;
using System.Drawing;
using OpenTK.Graphics;
using OpenTK.Graphics.OpenGL;

static class GraphicsHelpers
{
        // Returns a System.Drawing.Bitmap with the contents of the
current framebuffer
        public static Bitmap GrabScreenshot()
        {
            if (GraphicsContext.CurrentContext == null)
                throw new GraphicsContextMissingException();

            Bitmap bmp = new Bitmap(this.ClientSize.Width,
this.ClientSize.Height);
            System.Drawing.Imaging.BitmapData data =
                bmp.LockBits(this.ClientRectangle,
System.Drawing.Imaging.ImageLockMode.WriteOnly,
System.Drawing.Imaging.PixelFormat.Format24bppRgb);
            GL.ReadPixels(0, 0, this.ClientSize.Width,
this.ClientSize.Height, PixelFormat.Bgr, PixelType.UnsignedByte,
data.Scan0);
            bmp.UnlockBits(data);

            bmp.RotateFlip(RotateFlipType.RotateNoneFlipY);
            return bmp;
        }
}
```

How to render text using OpenGL

The simplest way to render text with OpenGL is to use System.Drawing. This approach has three steps:

1. Use `Graphics.DrawString()` to render text to a `Bitmap`.
2. Upload the `Bitmap` to an OpenGL texture.
3. Render the OpenGL texture as a fullscreen quad.

This approach is extremely efficient for text that changes infrequently, because only step 3 has to be performed every frame. Additionally, dynamic text can be reasonably efficient as long as care is taken to update only regions that are actually modified.

The downside of this approach is that (a) rendering is constrained by the capabilities of System.Drawing (i.e. poor support for complex scripts) and (b) it only supports 2d text. Moreover, care should be taken to recreate the Bitmap and OpenGL texture whenever the parent window changes size.

Sample code:

```
using System.Drawing;
using OpenTK.Graphics.OpenGL;

Bitmap text_bmp;
int text_texture;

window.OnLoad += (sender, e) =>
{
    // Create Bitmap and OpenGL texture
    text_bmp = new Bitmap(ClientSize.Width, ClientSize.Height); //
match window size

    text_texture = GL.GenTexture();
    GL.BindTexture(text_texture);
    GL.TexParameter(TextureTarget.Texture2D,
TextureParameterName.TextureMagFilter, (int)All.Linear);
    GL.TexParameter(TextureTarget.Texture2D,
TextureParameterName.TextureMinFilter, (int)All.Linear);
    GL.TexImage2D(TextureTarget.Texture2D, 0,
PixelInternalFormat.Rgba, text_bmp.Width, text_bmp.Height, 0,
        PixelFormat.Bgra, PixelType.UnsignedByte, IntPtr.Zero); //
just allocate memory, so we can update efficiently using
TexSubImage2D
};

window.Resize += (sender, e) =>
{
    // Ensure Bitmap and texture match window size
    text_bmp.Dispose();
    text_bmp = new Bitmap(ClientSize.Width, ClientSize.Height);

    GL.BindTexture(text_texture);
    GL.TexSubImage2D(TextureTarget.Texture2D, 0, 0, 0,
text_bmp.Width, text_bmp.Height,
        PixelFormat.Bgra, PixelType.UnsignedByte, IntPtr.Zero);
};

// Render text using System.Drawing.
// Do this only when text changes.
using (Graphics gfx = Graphics.FromImage(text_bmp))
{
    gfx.Clear(Color.Transparent);
    gfx.DrawString(...); // Draw as many strings as you need
}

// Upload the Bitmap to OpenGL.
// Do this only when text changes.
BitmapData data = text_bmp.LockBits(new Rectangle(0, 0,
text_bmp.Width, text_bmp.Height), ImageLockMode.ReadOnly,
System.Drawing.Imaging.PixelFormat.Format32bppArgb);
GL.TexImage2D(TextureTarget.Texture2D, 0, PixelInternalFormat.Rgba,
Width, Height, 0,
    PixelFormat.Bgra, PixelType.UnsignedByte, data.Scan0);
text_bmp.UnlockBits(data);

// Finally, render using a quad.
// Do this every frame.
GL.MatrixMode(MatrixMode.Projection);
GL.LoadIdentity();
```

```
GL.Ortho(0, Width, Height, 0, -1, 1);

GL.Enable(EnableCap.Texture2D);
GL.Enable(EnableCap.Blend);
GL.BlendFunc(BlendingFactorSrc.One,
BlendingFactorDst.OneMinusSourceAlpha);

GL.Begin(BeginMode.Quads);
GL.TexCoord(0f, 1f); GL.Vertex2(0f, 0f);
GL.TexCoord(1f, 1f); GL.Vertex2(1f, 0f);
GL.TexCoord(1f, 0f); GL.Vertex2(1f, 1f);
GL.TexCoord(0f, 0f); GL.Vertex2(0f, 1f);
GL.End();
```

This method can be easily generalized to use a more powerful text rendering library, like Pango#.

Chapter 5: OpenTK.Audio (OpenAL)

The **OpenAL 1.1 Crashcourse** will give an introduction how to use OpenAL in your applications.

OpenAL contains the following classes:

- AL "Audio Library"
- Alc "Audio Library Context"
- Alut "Audio Library Utilities"
- XRam "Memory Extension"
- Efx "Effects Extension"

OpenAL 1.0 Extensions that were included into 1.1: Multi-Channel Buffer playback Extension, Audio Capture Extension, Enumeration Extension.

It is recommended using these book pages as a starting point, and visit the online resources from the OpenAL website's documentation page for in-depth information. Downloading the OpenAL SDK is not required, but will provide you with some .wav files to toy around with and a few .pdf files not available directly at the OpenAL site.

Regarding compatibility, the "Generic Software" and "Generic Hardware" implementations of the OpenAL driver support OpenAL 1.1 and a few EFX Extensions, namely the Reverb Effect and Lowpass Filter. If the used Device cannot handle EAX natively, the driver will attempt to emulate the missing features.

Note that some functions of the OpenAL API are not imported for safety reasons. Rather use .Net's Thread.Sleep than Alut.Sleep, and Alut.CreateBuffer* instead of Alut.LoadMemory*. If this is a Problem, please voice it in the forum.

1. Devices, Buffers and X-Ram

OpenTK.Audio.AudioContext handles Device and Context allocation through Alc.

Instantiating a new AudioContext with a parameterless constructor will initialize the default Device and Context and makes it current. Calling the instance's Dispose method will destroy the Device and Context.

Buffers
Buffers in OpenAL are merely the storage for audio data in raw format, a Buffer Name (often called Handle) must be generated using AL.GenBuffers(). This buffer can now be filled using AL.BufferData() or using the AudioReader class (which loads a file from disk). The AudioReader functions implicitly use AL.BufferData() to pass the raw sound data into OpenAL's internal memory.

X-Ram
The X-Ram Extension allows to manually assign Buffers a storage space, it's use is optional and not required. To use the Extension, the XRam wrapper must be instantiated (per used Device), which will take care of most ugly things with Extensions for you. The instantiated object contains a bool that returns if the Extension is usable, which should be checked before calling one of the Extension's Methods.

Example code:

```
try
{
AudioContext AC = new AudioContext();
} catch( AudioException e)
{ // problem with Device or Context, cannot continue
  Application.Exit();
}

XRam = new XRamExtension( ); // must be instantiated per used Device
if X-Ram is desired.

// reserve 2 Handles
uint[] MyBuffers = new uint[2];
AL.GenBuffers( 2, out MyBuffers );

// Load a .wav file from disk
if ( XRam.IsInitialized ) XRam.SetBufferMode( ref MyBuffer[0],
XRamStorage.Hardware ); // optional

AudioReader sound = new AudioReader(filename)
AL.BufferData(MyBuffers[0], sound.ReadToEnd());
if ( AL.GetError() != ALError.NoError )
{
   // respond to load error etc.
}

// Create a sinus waveform through parameters, this currently
requires Alut.dll in the application directory
if ( XRam.IsInitialized ) XRam.SetBufferMode( ref MyBuffer[1],
XRamStorage.Hardware ); // optional
MyBuffers[1] = Alut.CreateBufferWaveform(AlutWaveform.Sine, 500f,
42f, 1.5f);

// See next book page how to connect the buffers to sources in order
to play them.
```

```
// Cleanup on application shutdown
AL.DeleteBuffers( 2, ref MyBuffers ); // free previously reserved
Handles
AC.Dispose();
```

A description of the sound data in the Buffer can be queried using AL.GetBuffer().

Now that the Buffer Handle has been assigned a sound, we need to attach it to a Source for playback.

2. Sources and EFX

Sources
Sources represent the parameters how a Buffer Object is played back. These parameters include the Source's Position, Velocity, Gain (Volume amplification) and more. The settings can be set/get by using AL.Source and AL.GetSource functions.

Continuation of the sourcecode from previous page:

```
uint[] MySources = new uint[2];
AL.GenSources( 2, out MySources ); // gen 2 Source Handles

AL.Source( TestSources[0], ALSourcei.Buffer, (int)MyBuffers[0] ); //
attach the buffer to a source

AL.SourcePlay( MySources[0]); // start playback
AL.Source( MySources[0], ALSourceb.Looping, true ); // source loops
infinitely

AL.Source( MySources[1], ALSourcei.Buffer, (int)MyBuffers[1] );
Vector3 Position = new Vector3( 1f, 2f, 3f );
AL.Source( MySources[1], ALSource3f.Position, ref Position );
AL.Source( MySources[1], ALSourcef.Gain, 0.85f );
AL.SourcePlay( MySources[1] );

Console.ReadLine(); // wait for keystroke before exiting

AL.SourceStop( MySources[0] ); // halt playback
AL.SourceStop( MySources[1] );

AL.DeleteSources( 2, ref MySources ); // free Handles
// now delete Buffer Objects and dispose the AudioContext
```

EFX

I'm sorry to do this, but if you want to work with EFX there is no other way. All I can give here is a brief overview that might help you make the decision if EFX is what you need. You will have to download the OpenAL SDK to get a copy of "Effects Extension Guide.pdf" from Creative labs, for in-depth information about programming with DSPs.

My advice is ignoring EFX, unless your game project is in 1st Person 3D. Environmental effects might look nice as a "selling point" on paper, but do not add any gameplay value to a Strategy game, or a 2D platform game.

The addition to OpenAL with EFX Extension is the rerouting of output signals.

- In vanilla OpenAL you load a Buffer, attach it to a Source and besides the Source's parameters that's all the influence you have about what ends up in the mixer.
- With EFX you may reroute a source's output through Filters and/or into Auxiliary Effect Slots. This allows more fine control about how a Source sounds when played, which is useful to achieve the effect of obstruction, occlusion or exclusion of a sound due to environment features like walls, obstacles or doors.

The new OpenAL Objects that come with EFX are "Effect", "Auxiliary Effect Slot" and "Filter".

An **Effect Object** stores the type of effect and the values for parameters of that effect. Types of Effects are for example Echo, Distortion, Chorus, Flanger, etc.

Auxiliary Effect Slots are containers for Effect Objects, whose output goes directly into the final output mix. The Slots are only used if there is a valid Effect Object attached to them, binding the reserved Handle 0 to a Slot will detach the previously bound Effect Object from it.

A **Filter** can be attached to a source, and either filter the "dry signal" that goes directly into the output mixer, or filter the "wet signal" that is forwarded to an Auxiliary Effect Slot.

3. Context and Listener

Like in OpenGL, a Context can be understood as an instance of OpenAL State. You can create multiple Contexts per Device, but each Context has the restriction of 1 Listener it's own unique Sources. Buffer Objects on the other hand may be shared by Contexts, which use the same Device.

Note that in contrast to OpenGL, OpenAL does not have an equivalent to SwapBuffers(). A Sources are automatically played until the end of their attached Buffer is reached, or until the programmer manually stops the Source playback.

Listener
The Listener represents the position and orientation of the Camera in the environment, thus there can be only one per Context. The settings can be set/get by using AL.Source and AL.GetSource functions.

It makes sense to handle it together with your OpenGL camera, to make sure a Source is properly positioned. This is very similar to OpenGL's Projection Matrix, with the

exception that there is no Frustum culling for audio (you may not see something behind you, but you can hear it).

A sample Camera, taken from the OpenAL manual:

```
void PlaceCamera(Vector3 ListenerPosition, float listenerAngle)
{
  // prepare some calculations
  float Sinus = (float)Math.Sin(listenerAngle);
  float Cosinus = (float)Math.Cos(listenerAngle);
  Vector3 ListenerTarget = new Vector3(ListenerPosition.X + Sinus,
ListenerPosition.Y, ListenerPosition.Z - Cosinus);
  Vector3 ListenerDirection = new Vector3(Sinus, 0, Cosinus);

  // update OpenGL - camera position
  GL.MatrixMode(MatrixMode.Projection);
  GL.LoadIdentity();
  GL.Frustum(-0.1333, 0.1333, -0.1, 0.1, 0.2, 50.0);
  Glu.LookAt(ListenerPosition, ListenerTarget, Vector3.UnitY);

  // update OpenAL - place listener at camera
  AL.Listener(ALListener3f.Position, ref ListenerPosition);
  AL.Listener(ALListenerfv.Orientation, ref ListenerDirection, ref
Vector3.UnitY);
}
```

Chapter 6: OpenTK.Compute (OpenCL)

[Describe the OpenTK.Compute namespace]

Chapter 7: OpenTK.Input

[Discuss the input classes provided by OpenTK]

Chapter 8: Advanced Topics

This chapter discusses advanced topics on the interaction of .Net/Mono, OpenGL and OpenAL. It builds on the previous two chapters and a good grasp of C#, OpenGL and OpenAL is assumed.

Vertex Cache Optimizations

Graphic cards usually have 2 Caches designed to help processing Vertices, one of their favorite tasks.

Pre T&L Cache
This Cache merely stores the untransformed Vertex read from a VBO. Optimizations

regarding this part of the Cache are simply sorting your Vertices in order of appearance, so the IBO issues Triangles in this order (0,1,2,0,2,3) rather then (999,17,2044,999,2044,2). This Cache is typically extremely large, being able to hold ~64k Vertices on a Geforce 3 and up.

Post T&L Cache
The more valuable Cache is the one storing the transformed results from the Vertex Shader, this Cache is typically very small (8 is minimum, 12-24 common) holding only very few Entries. It will only work with indexed primitives passed to GL.DrawElements, because GL.DrawArrays cannot make any assumptions which Vertices are actually identical.

While Pre-T&L Cache optimization only operates on the Vertices, Post T&L optimization will only operate on Indices (Primitives). Typically the Post T&L is calculated first, and the Pre T&L sorting step is performed on the optimized Indices Array.

Links for further reading

http://ati.amd.com/developer/i3d2006/I3D2006-Sander-TOO.pdf
http://www.cs.princeton.edu/gfx/pubs/Sander_2007_%3ETR/index.php
http://www.cs.umd.edu/Honors/reports/Vertex_Reordering_for_Cache_Coheren...
http://home.comcast.net/~tom_forsyth/papers/fast_vert_cache_opt.html
http://ati.amd.com/developer/tootle.html
http://developer.nvidia.com/object/vertex_cache_opt.html (ancient)
http://developer.nvidia.com/object/nvtristrip_library.html
http://www.clootie.ru/delphi/dxtools.html (DirectX based detector)

Useful quotes:
truncated quote from: http://developer.nvidia.com/object/devnews005.html

"When rendering using the hardware transform-and-lighting (TnL) pipeline or vertex-shaders, the GPU intermittently caches transformed and lit vertices. Storing these post-transform and lighting (post-TnL) vertices avoids recomputing the same values whenever a vertex is shared between multiple triangles and thus saves time. The post-TnL cache increases rendering performance by up to 2x. ...

...The post-TnL cache is a strict First-In-First-Out buffer, and varies in size from effectively 10 (actual 16) vertices on GeForce 256, GeForce 2, and GeForce 4 MX chipsets to effectively 18 (actual 24) on GeForce 3 and GeForce 4 Ti chipsets. Non-indexed draw-calls cannot take advantage of the cache, as it is then impossible for the GPU to know which vertices are shared. ...

...The mesh needs to be submitted in a single draw-call to optimize batch-size. The draw-call must be with an indexed primitive-type (see above), either strips or lists -- the performance difference between strips and lists is negligible when taking advantage of the post-TnL cache."

Last Update of the Links: January 2008

Garbage Collection Performance

The .Net Framework features an aggressive, generational and compacting Garbage Collector (GC): aggressive because it knows the location and reachability of every managed object, generational because it distinguishes long-lived objects objects from temporary ones, and compacting because it moves data in memory to avoid leaving holes behind. The GC is a great tool in the .Net arsenal, not only because it increases productivity but also because it provides extremely fast memory allocations (compared to standard C/C++ malloc/new).

[Describe the unmanaged resource pool, pinning and performance considerations]

GC & OpenGL (work in progress)

As discussed in the previous chapter, GC finalization occurs on the finalizer thread. This poses some problems on OpenGL resource deallocation, since the context used to create the resources is not available in the finalizer thread!

Since OpenGL functions cannot be called in finalizers, a different methodology must be followed. By implementing the disposable pattern, we can use the Dispose() method to deterministaclly destroy OpenGL resources in the main thread. By modifying the finalizer logic we can provide a way to flag resources as 'dead', and destroy them from the main thread. Last, by extending the concept of the OpenGL context, we can be notified of context destruction, to release all related resources.

The following code describes the implementation of the "OpenGL disposable pattern" in OpenTK, but it is easy to adapt this code to any managed OpenGL project:

```
// This code is out-of-date. Please do not use it!

// The OpenGL disposable pattern
class GraphicsResource: IDisposable
{
    int resource_handle;    // The OpenGL handle to the resource
    GraphicsContext context;      // The context which owns this
resource

    public GraphicsResource()
    {
        // Obtain the current OpenGL context, and allocate the
resource
        context = GraphicsContext.CurrentContext;
        if (context == null)
            throw new InvalidOperationException(String.Format(
                "No OpenGL context available in thread {0}.",

System.Threading.Thread.CurrentThread.ManagedThreadId));

        resource_handle = [...];

        context.Destroy += ContextDisposed;
    }
```

```
    #region --- Disposable Pattern ---

    private void ContextDisposed(IGraphicsContext sender, EventArgs
e)
    {
        context.Destroy -= ContextDisposed;
        // TODO: Shared resources shouldn't be destroyed here.
        Dispose();
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    // If the owning context is current then destroy the resource,
    // otherwise flag it (so it will be destroyed from the correct
thread)..
    // TODO: Is the "manual" flag necessary? Simply checking for the
    // owning context should be enough.
    private void Dispose(bool manual)
    {
        if (!disposed)
        {
            if (!context.IsCurrent || !manual)
            {
                GC.KeepAlive(this);
                context.RegisterForDisposal(this);
            }
            else
            {
                // Destroy resource_handle through OpenGL
                disposed = true;
            }
        }
    }

    ~GraphicsResource()
    {
        Dispose(false);
    }

    #endregion
}
```

In OpenTK, each GraphicsContext class maintains a queue of OpenGL resources that need to be destroyed. Resources are added to this queue through the RegisterForDisposal() call, and they are destroyed through the DisposeResources() method. The whole process is deterministic: it is your responsibility to call DisposeResources at appropriate time intervals (or setup up a timer event to do this for you).

Resource creation takes a small performance hit due to the call to GraphicsContext.CurrentContext, while garbage collect-able OpenGL resources consume slightly more memory (due to the reference to the GraphicsContext). Prefer

calling the Dispose() method to destroy resources instead of relying on the GC, as finalizable resources are only collected on a generation 1 or 2 GC sweep.

The current implementation in OpenTK does not take shared contexts into account - this will be taken care of in the near future.

Chapter 9: Hacking OpenTK

This chapter contains instructions for people wishing to modify or extend OpenTK. It describes the project structure, wrapper design, coding style and various caveats and hacks employed by OpenTK to achieve wider platform support.

Project Structure

The OpenTK project consists of a number of managed, cross-platform assemblies:

- OpenTK: this is the *core* OpenTK assembly. It provides the Graphics, Audio and Compute APIs, the math toolkit and the platform abstraction layer.
- OpenTK.Compatibility: this assembly provides an upgrade path for applications compiled against older versions of OpenTK and the Tao Framework. When a deprecated method is removed from core OpenTK, it is added to this dll.
- OpenTK.GLControl: this assembly provides the GLControl class, which adds OpenGL support to System.Windows.Forms applications.
- OpenTK.Build: this assembly provides the cross-platform build system for OpenTK. It can be used to generate MSBuild-compatible project files for use with Visual Studio (version 2005 or higher), Sharpdevelop (version 2.0 or higher) and MonoDevelop (version 2.0 or higher).
- OpenTK.Examples: this assembly provides a number of samples built with OpenTK. It covers topics related to OpenGL, OpenGL|ES, OpenAL, OpenCL and general OpenTK usage.

In addition to these assemblies, OpenTK maintains a custom binding generator which generates the OpenGL, OpenGL|ES and OpenCL bindings. It consists of two assemblies:

- Converter, which converts the OpenGL|ES and OpenCL C headers to XML files.
- Bind, which converts the OpenGL .spec files or the Converter XML files into C# code.

Finally, OpenTK provides a QuickStart project, which shows how to setup and build an OpenTK application.

OpenTK Structure

The OpenTK solution provides the following public namespaces:

- OpenTK: contains classes to create windows (GameWindow, NativeWindow), perform 3d math, interact with the monitor (DisplayDevice, DisplayResolution) as well as query the platform configuration.
- OpenTK.Graphics: contains bindings for OpenGL and OpenGL|ES.
- OpenTK.Audio: contains bindings for OpenAL.
- OpenTK.Compute: contains bindings for OpenCL.
- OpenTK.Input: contains classes to interact with input devices (Keyboard, Mouse, Joystick).
- OpenTK.Platform: contains classes to extend OpenTK or interact with the underlying platform.

The public API of OpenTK is completely cross-platform. All platform-specific code is contained in internal interfaces under the `OpenTK.Platform` namespace. In that sense, most public classes act as *façades* that forward method calls to the correct platform-specific implementation.

```
public class Foo : IFoo
{
    IFoo implementation;

    public Foo()
    {
        implementation = OpenTK.Platform.Factory.Default.CreateFoo();
    }

    #region IFoo Members

    public void Bar()
    {
        implementation.Bar();
    }

    #endregion
}
```

This pattern is used in all public OpenTK classes that need platform-specific code to operate: DisplayDevice, DisplayResolution, GraphicsContext, GraphicsMode, NativeWindow and the various input classes.

Classes that do not rely on platform-specific code and classes that contain performance-sensitive code do not use this pattern: the various math classes, the OpenGL, OpenCL and OpenAL bindings, the AudioContext and AudioCapture classes all fall into these categories.

Wrapper Design

OpenTK provides .Net wrappers for a various important native APIs: OpenGL, OpenGL ES, OpenAL and OpenCL (in progress). Unlike similar libraries, OpenTK places an emphasis in usability and developer efficiency, while staying true to the nature of the native interface. To that end, it utilizes a number of .Net constructs that are not available in native C by default:

- Strongly-typed enum parameters instead of integer constants.
- Generics instead of void pointers.
- Namespaces instead of function prefixes ('gl', 'al').
- Function overloads instead of function suffices (`Vector3` instead of `Vector3f`>, `Vector3d`, ...).
- Automatic extension loading.
- Inline documentation for functions and parameters, accessible through IDE tooltips (intellisense).
- CLS-compliance, which makes the bindings usable by all .Net languages.
- Cross-platform support, which allows the bindings to be used by any platform supported by .Net or Mono.

The bindings are generated through an automated binding generator, which converts the official API specifications into C# code. The following pages describe the generation process in detail.

Official API specifications:

- OpenGL: http://www.opengl.org/registry
- OpenGL ES: http://www.khronos.org/registry/gles/
- OpenAL: http://connect.creativelabs.com/openal/Downloads/Forms/AllItems.aspx
- OpenCL: http://www.khronos.org/registry/cl/

Appendix 1: Frequently asked questions

[General Questions]

1. **What is the Open Toolkit *exactly*?**
   The Open Toolkit is a C# library that:
   1. allows .Net programs to access OpenGL, OpenAL and OpenCL
   2. abstracts away the platform-specific code for window creation, input devices, and
   3. provides helper functions (math, fonts, etc)

   As such, is roughly analogous to SlimDX, SDL or GLFW.

2. **Is OpenTK limited to games?**
   No! OpenTK can be - and has been - used in scientific visualizations, VR, modeling/CAD software and other projects.
3. **What is the difference between OpenTK and the Tao framework?**
   The Tao framework tries to follow the unmanaged APIs as closely as possible. OpenTK, on the other hand, takes advantage of .Net features like function overloading, strong-typing and generics. Consider the following code snippet:

```
// OpenTK.Graphics code:
GL.Begin(BeginMode.Points);
GL.Color3(Color.Yellow);
GL.Vertex3(Vector3.Up);
GL.End();
// Tao.OpenGl code:
```

```
Gl.glBegin(Gl.GL_POINTS);
Gl.glColor3ub(255, 255, 0);
Gl.glVertex3f(0, 1, 0);
Gl.glEnd();
```

The code is trivial, but it illustrates the difference nicely: OpenTK removes unecessary cruft ('gl', 'ub', 'f'), uses strongly-typed enums ('BeginMode') and integrates better with .Net ('Color').

There are other differences not so readily apparent: OpenTK will not allow you to pass invalid data to OpenGL (wrong tokens or non-valuetype data); it plays better with intellisense (inline documentation, overloads, strong-types); it checks for OpenGL errors automatically in debug builds.

All these become more important as a project grows in size.

8. **Will my Tao project run on OpenTK?**
   Starting with version 0.9.9-2, OpenTK is compatible with Tao.OpenGl, Tao.OpenAl and Tao.Platform.Windows.SimpleOpenGlControl. Simply replace your Tao.OpenGl, Tao.OpenAl and Tao.Platform.Windows references with OpenTK and OpenTK.Compatibility and your project will as before, while gaining advantage of all OpenTK features.
9. **OpenGL is not object-oriented. Does OpenTK change that?**
   No, the Open Toolkit mirrors the raw OpenGL API. This was a conscious design decision, to avoid introducing artificial limitations. However, users have contributed object-oriented libraries built on top of OpenTK - check out the project database.
10. **I care about speed. Is OpenTK slow?**
    No, OpenTK introduces minimal overhead over raw OpenGL. However, do note that the underlying runtimes (.Net/Mono) introduce some unique performance considerations - refer to our documentation for more information. Performance is always a concern, so please report an issue if you believe something could run faster.
11. **Which platforms does OpenTK run on?**
    OpenTK is primarily tested on Windows, Linux and Mac OS X, but is also known to work on Solaris and *BSD variants. All features work across platform without recompilation.
    A lighter version is also made available for the iPhone through the MonoTouch project (recompilation required). There is no official support for Windows Mobile or Android at this point.
12. **Is OpenTK safe to use? How mature is it?**
    OpenTK is considered safe for general use. It is being used successfully by both free and commercial projects and the library is under active development, with regular bugfix and feature releases.

[Windows.Forms & GLControl Questions]

1. **How can I make my Form fullscreen?**
   Use the following code snippet:

```
2. myForm.TopMost = true;
3. myForm.FormBorderStyle = FormBorderStyle.None;
```

```
myForm.WindowState = FormWindowState.Maximized;
```

4. **My GLControl.Load event isn't fired.**
   Please upgrade to OpenTK 0.9.9-4 or newer.
5. **How do I use stencil, antialiasing or OpenGL 3.x with GLControl?**
   Create a custom control that inherits from GLControl:

```
class CustomGLControl : GLControl
{
    // 32bpp color, 24bpp z-depth, 8bpp stencil and 4x
antialiasing
    // OpenGL version is major=3, minor=0
      public CustomGLControl()
          : base(new GraphicsMode(32, 24, 8, 4), 3, 0,
GraphicsContextFlags.ForwardCompatible)
      { }
}
```

[Graphics questions]

1. **How can I save a screenshot?**
   Refer to the "How to save an OpenGL rendering to disk" section in the documentation.

Appendix 2: Function Reference

You can read the function reference online. A PDF version of this reference is included with your OpenTK distribution, under the Documentation/ folder.

Appendix 3: The project database

The project database is an index of projects related to the Open Toolkit. Every project in the database receives a unique project page and gains access to the issue tracker and the project release service.

All registered users may submit their own projects, subject to the following restrictions:

1. Your project must use, extend or be somehow related to the Open Toolkit library.
2. Your project must be released under an OSI approved license.

Closed-source and / or commercial projects will be reviewed by the Open Toolkit team and approved on a case by case basis.

By submitting a project to the database, you acknowledge that:

1. This is a free service provided to the Open Toolkit community that comes *without any warranty*. In case there is any doubt, the OpenT Toolkit team does not offer you any warranty, express or implied, for the behavior of the project database, nor fitness of purpose towards any application. Keep backups!

2. The Open Toolkit team maintains the right to remove any project from the database or terminate the whole database, for whatever reason, without prior notice.

Creating a project

Only registered users are allowed to create projects. To create a project, click on Create content -> Project and complete the required information:

1. In the **"project categories"** section, click "Contributed" and select all relevant categories.
    - You can use the control key to select multiple categories.
    - If your project is closed-source or commercial, you **must** select the relevant categories.
    - Please do not use the "Core" category. It is reserved for the Open Toolkit.
2. In the **"full project name"** field, type a descriptive name for your project (e.g. The Open Toolkit library).
3. In the **"full description"** field, describe what your project is, what it does and any other information you deem relevant (e.g. requirements, features).
4. In the **"short project name"** field, type a compact name for your project. This will be used in the URL of the project page and the issue tracker (e.g. project/opentk). Do not use spaces or any other special characters.
5. **Upload screenshots** for your project. This step is very important, as users tend to avoid projects without or with low quality screenshots. If your screenshots display 3d graphics, you can improve their quality by enabling **antialiasing** and **anisotropic filtering**.
6. Don't forget to add a link to the **homepage** of the project (if any), its **source code repository** and **license**!

Creating a project release

Once you have created a project, you can create a project release by clicking on Create content -> Project release.

The first step is to select your project from the drop-down list. Click next to proceed to the actual release page:

1. Choose which **OpenTK version** your project targets. For example, if your project uses relies on OpenTK 0.9.5, you should choose 0.9.x here. If your project targets OpenTK 1.0 (not yet released at the time of writing), you should choose 1.0.x. This information is important, as it indicates whether different projects can be used together. Please note that OpenTK is backwards compatible, which means you should choose the **lowest** OpenTK version that can support your project.
2. Fill in the **release version**. This should match the actual version in your project properties (you can view this information in Visual Studio by right-clicking your project, selecting properties and then "assembly information".

Likewise for SharpDevelop and MonoDevelop). You can optionally add an exta identifier to convey more information (typical identifiers include "beta", "rc", "final" and "wip").

3. Fill in the **"body"** textbox with your release notes.
4. Optionally, you can **upload your release** to opentk.com using the file field. **Please consult with us before uploading releases bigger than 20MB!** If your release is very large, consider using a torrent for distribution.

   You can also **redirect** the downloads to an external resource (e.g. your own homepage or sourceforge), by using a html redirect. Copy the following code to a file named [project name]-[release number].html (e.g. opentk-0.9.5.html), edit the necessary links and upload it through the "file" field:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
  <head>
          <title>Redirecting to download.</title>
          <meta http-equiv="REFRESH"
content="0;url=http://www.example.com" />
  </head>
  <body>Redirecting to the <a
href="http://www.example.com">download page</a>.</body>
</html>
```

Appendix 4: Links

The following pages contain links selated to game development, graphics and audio programming in general.

Models and Textures

File Formats don't matter. Please do not add commercial 3D model sites, unless they offer a huge collection of free models aswell.

Textures

http://www.imageafter.com/textures.php

http://www.cgtextures.com/

http://developer.nvidia.com/object/IO_TTVol_01.html

http://www.freefoto.com/browse/33-00-0?ffid=33-00-0

http://www.noctua-graphics.de/english/freetex_e.htm

http://www.grsites.com/textures/

http://www.absolutecross.com/graphics/textures/

http://www.m3corp.com/a/download/3d_textures/pages/index.htm

http://studio.planethalflife.gamespy.com/textures.asp

http://local.wasp.uwa.edu.au/~pbourke/texture_colour/index.html

Models

http://www.highend3d.com/downloads/

Sites that have both

http://telias.free.fr/

http://www.psionic3d.co.uk

Tools

Generates Explosion Sprite sheets http://www.geocities.com/starlinesinc/

Procedural Materials http://www.spiralgraphics.biz/viewer/

Reference Images, Concept Art and the likes

http://www.prideout.net/colors.php

http://www.3d.sk/

http://www.fineart.sk/

Disclaimer: Some links lead directly to download sections of websites in order to be convenient for the reader. The Copyright and license details vary between the websites, if you follow one of the links above it is your own responsibility to read and acknowledge the websites terms of use. The authors of this link collection take no liability for any misuse or copyright violation by the readers.

OpenGL Books and Tutorials

OpenGL related Books

The 'red book' (start here when in doubt) http://www.glprogramming.com/red/

OpenGL Programming Guide for Mac OS X
http://developer.apple.com/documentation/GraphicsImaging/Conceptual/Open...

Programming with OpenGL: Advanced Rendering
http://www.opengl.org/resources/code/samples/advanced/

GPU Gems 1 http://developer.nvidia.com/object/gpu_gems_home.html

GPU Gems 2 http://developer.nvidia.com/object/gpu_gems_2_home.html

GPU Gems 3 http://http.developer.nvidia.com/GPUGems3/gpugems3_pref01.html

OpenGL related Tutorials

Transforms & Drawing http://www.codecolony.de/

Step by Step, from a Triangle to more complex scenes http://nehe.gamedev.net/

Lots of Extension-related Examples http://www.codesampler.com/oglsrc.htm

Misc. advanced Tutorials http://www.opengl.org/sdk/docs/tutorials/Lighthouse3D/

GLSL related Tutorials

Compilation, Linking & State
http://www.opengl.org/sdk/docs/tutorials/ClockworkCoders/

Material & Lighting http://www.typhoonlabs.com/

Articles, Demos & Research Papers

Old Website, Forum is a goldmine http://www.gamedev.net/

Misc. advanced Demos /w Source http://www.humus.name

Voxel & Raytracing http://www.codermind.com/articles/Technical-articles.html

Programming links

- MSDN Library, and especially its .Net subsection is an essential programming resource.
- Rico Mariani's Performance Tidbits provide invaluable information on .Net optimization techniques.
- Gamedev contains game news and articles on game development.
- OpenGL SDK is an one stop resource for OpenGL related questions.
- OpenGL registry contains specifications for all OpenGL functions. For advanced developers.

Tools & Utilities

- Blender is an excellent 3d modeller and editor.
- NShader adds GLSL syntax highlighting to Visual Studio 2008.
- glView is an excellent utility that shows the extensions supported on your platform. Available for Windows and Mac OS platforms.
- GLIntercept is a free and open-source OpenGL function call interceptor. Windows only.

- The Tao Framework is a collection of .Net/Mono bindings to libraries like OpenAL, DevIL, ODE and more.
- NBidi is a .Net Implementation of the BIDI algorithm for complex Hebrew and Arabic RTL scripts.
- AgateLib is a cross-platform 2d OpenGL library with an OpenTK driver.
- Golem3D is a cross-platform model editor that uses OpenTK.

Tutorials

External tutorials

- Lighthouse3D (High quality tutorials on GLSL, shadows, math and more)
- NeHe OpenGL tutorials (wide range of topics, from simple to advanced. Written as annotated code)
- Clockwork Coders (GLSL)
- Code Colony (Camera, Vertex Arrays)

Appendix 5: Translations

The Open Toolkit manual is available in the following languages:

- Deutsch
- Ελληνικά
- Russian