**OpenTK Documentation**

**Use OpenGL API:**OpenTK wraps OpenGL in static class, so to use OpenGl api like **glViewport**, do **GL.Viewport** instead.  
Some enums from OpenGL such as **GL\_ARRAY\_BUFFER** becomes **BufferTarget.ArrayBuffer**, following the format of **EnumName.EnumValue**

**OVERRIDES**

**Open Game Window:**class can inherit from GameWindow, then pass options through that class

class MainWindow : GameWindow {

new MainWindow (//custom assign value params )   
 : base ( //assign initial optional window values)   
{ //assign params  
 Title += “ GL VERSION : ” …  
}

}

In game loop, call run to open window:

void Main(){  
 new MainWindow(//pass in custom params).Run(60.0, 0.0);  
 //params in Run() ensures that the game logic update runs at the same speed   
 //of the machine   
 //performance varies   
}

**Resize Game Window:**when **GameWindowFlag.Default**, means user can resize game window. To reset viewport when window is resized.

override void OnResize(EventArgs e){  
 GL.Viewport(0, 0, Width, Height);   
 //x and y to 0, w and h of viewport to window dimens  
 //(0, 0) lower left corner of screen  
}

**OnResize** gets called whenever window dimension changes

**Initialize:**when the game window opens, the method OnLoad gets called. Use it to initialize

override void OnLoad(){  
 //do stuff  
}

**Updates  
OnUpdateFrame** updates every frame, so update the world in here

Override void OnUpdateFrame(FrameEventArgs e){  
 //update stuff  
}

**Rendering**also gets called every frame like **OnUpdateFrame**. Put all the rendering in here

override void OnRenderFrame(FrameEventArgs e) {  
 //render stuff  
 e.Time //from FrameEvebtArgs gives the elapsed time of the window  
  
 SwapBuffer(); //shows the rendered scene to user on screen  
}

**COMPLING SHADER & LINKING**

For object to be rendered, it uses shaders, but first we need to create them

**Creating Shader**The bare minimum to get something to the screen are using **VertexShader** and **FragmentShader**. The procedure to load both of them are the same.

**VertexShader**: handles individual vertices’ positions, rotations, etc. being fed attribute data to set variables within the shader

#version 450 core

layout (location = 0) in vec4 position;

layout(location = 1) in vec4 color;

out vec4 vs\_color;

layout (location = 20) uniform mat4 projection;

layout (location = 21) uniform mat4 modelView;

void main(void)

{

gl\_Position = projection \* modelView \* position;

vs\_color = color;

}

**FragmentShader**: the shader that process the fragments when creating / rasterizing a shape into a set of colors and a single depth value.

#version 450 core

in vec4 vs\_color;

out vec4 color;

void main(void)

{

color = vs\_color;

}

To create a shader use **GL.CreateShader** method, which will create a shader from the shader source code file using **GL.ShaderSource**. To load the content to the shader file use **System.IO.File.ReadAllText**, which then compile it using **GL.CompileShader**

int vertexShader = GL.CreateShader(ShaderType.VertexShader);   
//create a shader with type

GL.ShaderSource(vertexShader, File.ReadAllText(@"../../vertexShader.vert")); //loads it

GL.CompileShader(vertexShader); //compile it

Same process is used for **FragmentShader**

Then for us to be able to use the shader, we need to create a program using **GL.CreateProgram**, then attach the shaders using **GL.AttachShader**, then link the program with to the created one using **GL.LinkProgram**

int shaderProgram = GL.CreateProgram();

GL.AttachShader(shaderProgram, vertexShader);

GL.AttachShader(shaderProgram, fragmentShader);

GL.LinkProgram(shaderProgram);

After the linkage, it is ok to remove the shaders we created previously, since it only needs to be loaded once. So first call **GL.DetachShader** then **GL.DeleteShader**

GL.DetachShader(shaderProgram, vertexShader);

GL.DetachShader(shaderProgram, fragmentShader);

GL.DeleteShader(vertexShader);

GL.DeleteShader(fragmentShader);

Then to use it during rendering, put **GL.UseProgram** in **void OnRenderFrame**

**Drawing Shapes (OpenGL Created Shape):**  
now we have the shaders we needed, we can use it to draw a basic shape without a custom vertex array   
add a vertex buffer initialization and bind it so we can use it. Using **GL.GenVertexArrays,** and bind it using **GL.BindVertexArray**

int vertexArray;  
GL.GenVertexArrays(1, out vertexArray);   
//p1 num of array to generate, p2 object name stored  
GL.BindVertecArray(vertexArray)

and now that the arrays are binded, we can use **GL.DrawArrays** to draw the binded array to screen in **void OnRenderFrame**

**GL.DrawArrays(PrimitiveType.Point, 0, 1)**

the above will create a square on the screen

Use **GL.PointSize** to adjust the size of the created square

**Clean up:  
GL.DeleteProgram** to delete the program previously created  
**GL.DeleteVertexArray** to delete any array that is binded

**Drawing Custom Shapes (Buffered):**

**INPUTS**

**Keyboard Input:  
using OpenTK.Input**get state of keyboard from Keyboard class like so

var keyState = Keyboard.GetState();

which then use it to check whether is pressed or not like so,

if (keyState.isKeyDown(Key.//any key in the enum)){  
 //do stuff  
}