**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):**have a few variables ready to hold the **vertex array** and **buffer**.

private readonly int vertexArray;  
pirvate readonly int buffer;

to first initialize a vertex array and a buffer

vertexArray = GL.GenVertexArray();  
 buffer = GL.GenBuffer();

that will let OpenGL create a vertex array and buffer object name in the system. Now you bind them with previously created names

GL.BindVertexArray(vertexArray);  
GL.BindBuffer(BufferTarget.ArrayBuffer, buffer);

Now, create a buffer object’s immutable data store. Basically, telling GL to use this new buffer we created.

GL.NamedBufferStorage(  
 buffer,  
 Vertex.size \* vertices.length //size needed by this buffer  
 vertices //data to initialize with  
 BufferStorageFlag.MapWriteBit); //write to the buffer

After that, we’ll need to bind the attribute of the shader to the vertex array, so that changes can be seen using **GL.VertexArrayAttribBinding**, after binding, it needs to be enabled and formatted using **GL.EnableVertexArrayAttrib,** and **GL.VertexArrayAttribFormat**

GL.VertexArrayAttribBinding(vertexArray,  
 0, //location in shader  
 0); //binding index  
GL.EnableVertexArrayAttrib(vertexArray, 0);  
GL.VertexArrayAttribFormat(  
 vertexArray,  
 0, //shader attribute index / location  
 4, //size of attribute, depends  
 VertexAttribType.Float //type  
 false, //normalization? Float ignores it  
 0); //position offest

If there are more attributes that are in need of binding, repeat the process.

After binding of the attribtues, you bind the buffer to the vertex array, in which will modify the binding of currently bound vertex array object. Use **GL.VertexArrayVertexBuffer()**.

**CREATE & DISPLAY TEXTURE**

GL.Enable(EnableCap.Blend); //to render semi/transparent textures   
Gl.BlendFunc(BlendingFactor.SrcAlpha, BlendingFactor.OneMinusSrcAlpha);

Load texture from disk using **System.Drawing** namespace, and using **Image.FromFile()** to load it. Since GL takes the image as in RGBA value, just need to return an array has the size of all pixel \* 4 to store each value

using (var bmp = (Bitmap)Image.FromFile(fileName)){

width = bmp.Width;

height = bmp.Height;

r = new float[width \* height \* 4];

int index = 0;

for(int y = 0; y < height; y++){

for(int x = 0; x < width; x++){

var pixel = bmp.GetPixel(x, y);

r[index++] = pixel.R / 255f;

r[index++] = pixel.G / 255f;

r[index++] = pixel.B / 255f;

r[index++] = pixel.A / 255f;

}

}

}

Note: **using** block automatically collects garbage / dispose it after finishing if the type implements **IDisposable**.

Loading texture has a similar pattern to creating vertex array and buffers.

GL.CreateTexture(TextureTarget.Texture2D, //create it  
1, //number of texture to create  
out int texture); //name of the texture array is stored  
GL.TextureStorage2D(texture, //specify storage req for 2d texture  
 1, //mipmap level  
 sizedInternalFormat,Rgba32f, //texture format  
 width, height);  
GL.BindTexture(TextureTarget.Texture2D, texture); //bind it  
//maps a portion image onto each primitives with texturing enabled  
GL.TextureSubImage2D(texture,   
 0, //mipmap level, 0 is base image level  
 0, 0, //offset  
 width, height, //width height of image  
 PixelFormat.Rgba, //pixel data format  
 PixelType.Float, //data type of pixel data  
 LoadTexture()); //the pixel data loaded from disk

After the process, OpenGL should have the texture data.

**Display Textures:**when textures are created, it won’t be shown unless we bind it. So, bind it to a **Program** that uses the shaders that takes texture (vec2) as parameter

GL.UseProgram(program);  
 GL.BindVertexArray(VertexArray);  
 GL.BindTexture(TextureTarget.Texture2D, texture);

**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  
}