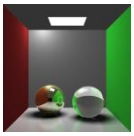


2023/2024, 4th period

INFOGR: Graphics

Practical #0: OpenTK Introduction

Author: Peter Vangorp, based on a previous version by Jacco Bikker

Regarding this practical:

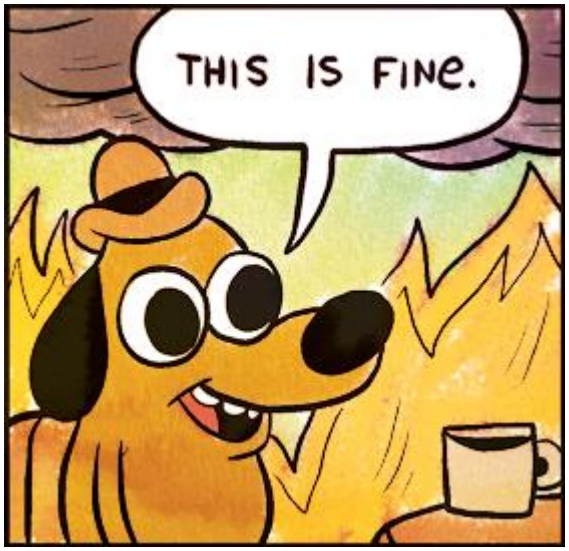
This is an introduction that ensures that you have a working C# development environment for the following practicals, and that takes you through some basics of graphics programming. Although this practical is optional, we highly recommend that you complete it. *You will need to use what you learn here for the two graded practical assignments*. Note that you can ask questions about this tutorial on Microsoft Teams in the Computer Graphics channel. Also note that you do not need to hand in the results.

And a final note: This practical is frequently referred to as 'tutorial', while in fact you may need to do some digging to get things working. *This is intentional.*

Remember that your goals for this period are:

1. **completing two practical assignments on graphics programming;**
2. **completing two math exams, possibly with a few graphics questions too.**

Stay focused, stay in touch, ask questions, do the work, and you will be fine.



This is fine. [source]

Setting up OpenTK for C# in Microsoft Visual Studio

For this tutorial, we will be using Microsoft Visual Studio 2022 to develop in C#. OpenTK is used to render images with OpenGL or to display images that we render without OpenGL.

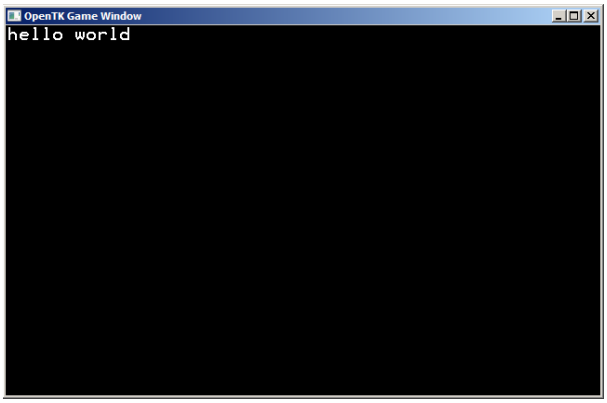
In more detail:

- Microsoft Visual Studio Community 2022 can be obtained for free via: https://visualstudio.microsoft.com/vs/community
  Please follow the instructions to install Visual Studio and select the .NET Desktop Development workload.
- For this course, we will assume you develop using C#. The use of other programming languages is not recommended for this course.
- C# does not itself let you work with OpenGL. For this we use OpenTK, which is a 'thin wrapper' for OpenGL, i.e., it exposes the functionality of OpenGL from C#, while the underlying OpenGL functionality remains clearly visible. That means that anything you learn using C#/OpenTK will still be useful if you later want to work with OpenGL in any other programming language.
- OpenGL itself is a low level interface to the GPU. It is a well-established standard for this, and, unlike Microsoft DirectX, it is not tied to a specific vendor. OpenGL and DirectX bear strong similarities. Anything you learn using OpenGL will still be useful if you later want to work with DirectX (or any other graphics API for that matter).

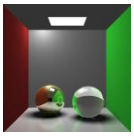
We will be using a small C# template application that provides you with some minimal graphics functionality. In the first part of this tutorial, we will use this template to explore some basic concepts. In the second part of this tutorial, we will expand the template so we can use OpenGL graphics via OpenTK.

Preparations:

1. Download the Practical #0 template code from the Files tab of the INFOGR Team.
2. Extract the zip file to a folder on your hard disk.
3. Open the .sln solution file in Visual Studio and compile the code.
4. Run the program. You should see the following output:



If this all works, proceed to PART 1.



2023/2024, 4th period

INFOGR: Graphics

Preparing for Practical 1: Drawing using the template

Architecture

The template consists of three source files:

- `template.cs`
- `surface.cs`
- `MyApplication.cs`

Don't be afraid to have a look at the code inside. There are many functions and variables that you may use from your own code.

The file `MyApplication.cs` is where we will be building our application. The class has two methods: `void Tick()`, which will be executed once per frame, and `void Init()`, which will be executed precisely once, when the application starts. There is also a single member variable `screen`, which represents the pixels in the window, as well as basic functionality to operate on these pixels.

This functionality represents the typical graphical application flow: after initialization (loading textures and models, setting up the scene, etc.) we need to produce a steady flow of frames. Each *tick* we will want to update the state of the world, and visualize this state.

The code that calls the `Tick` function can be found in `template.cs`, but we will ignore this for now. Two things are worth mentioning however: the first is that the frame rate is not limited to a chosen maximum number of frames per second (fps); the second is that the template constantly monitors your ESC key; hitting it terminates the application.

Class Surface

The only member variable of the `MyApplication` class is of type `Surface`. The implementation of this class can be found in surface.cs. Instances of `Surface` own a 1D array of integers, representing pixel colors. Despite the 1D array, a surface typically is two-dimensional; it has a width and a height. To read a single pixel using an x,y-coordinate, we must thus calculate the correct location in this array:

```
int location = x + y * width;
```

Reading a pixel can now be done using

```
int pixel = screen.pixels[location];
```

and writing to the pixel is done as follows:

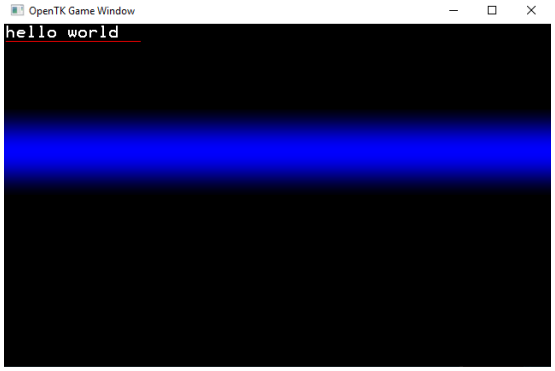
```
screen.pixels[location] = 255;
```

A single integer thus represents a color. To understand how this works, it is important to see that an integer actually consists of four bytes: it is a 32-bit value. For colors, we store red, green and blue each in a byte. This leaves us 1 byte (8 bits), which we will not use.

The first byte in the integer can store values of 0…255. These will show up as blue colors.

Exercise 1: write some code in the Tick function that draws a horizontal blue pipe in the middle of the window. Make it as wide as the screen, and use a 'thickness' of about 80 pixels. Use shades of blue (a gradient, as in the picture below) to suggest the round shape.

Depending on your interpretation of the exercise, the output could look something like this:



To get to the other bytes in the integer value, we need to use values that exceed 255. Value 256 for example yields a green rectangle (but: a very dark one). To get all shades of green, we use values 0...255, but we multiply them by 256, effectively shifting them 8 bits to the left – just like multiplying a decimal number by 100 shifts all digits two places to the left. To get all shades of red, we again use values 0...255, but we multiply them by 256 * 256, shifting them by 16 bits.

calculator knows binary.

A convenient way to create a red color value would be:

```
int CalculateRedColor( int shade ) { return shade * 256 * 256; }
```

Now if we also had a green color value, we could blend the two to obtain yellow:

```
int yellow = CalculateRedColor( 255 ) + CalculateGreenColor( 255 );
```

There is an easier (and potentially faster) way to get our red shade: using *bitshifting*. Looking at a color as a binary value, a multiplication by 256 can be achieved by shifting values 8 bits to the left. This lets us define a fast and convenient method for constructing colors:

```
int MixColor( int red, int green, int blue )
{
   return (red << 16) + (green << 8) + blue;
}
```

Here, red, green and blue are values between 0 and 255.

Exercise 2: modify the code from the first exercise, so that it draws a yellow pipe.

Coordinate Systems

The coordinate system of our window is simple: the template defaults to a 640 x 400 window, and therefore x ranges from $0...639$, and y from $0...399$. There are several problems with this:

1. designing an application for a single resolution is limiting;
2. the coordinate system is not centered;
3. the y-axis is inverted compared to common math standards.

For many applications (e.g. Android games) it is important to decouple window resolution from the coordinate system used for the simulation.

Suppose we want to draw a spinning rectangle in the center of the window. Drawing the rectangle is straightforward:

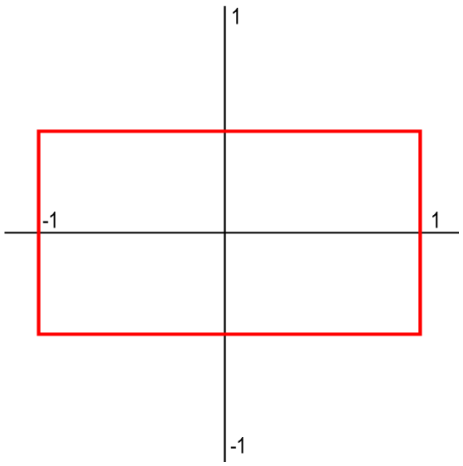
```
screen.Line( 220, 150, 420, 150, 0xffffff );
screen.Line( 220, 150, 220, 250, 0xffffff );
screen.Line( 420, 150, 420, 250, 0xffffff );
screen.Line( 220, 250, 420, 250, 0xffffff );
```

Note how the color is represented here. A 32-bit hexadecimal number (indicated by the `0x` prefix) consists of 8 digits in the range `0..f` (for $0...15$). Each digit thus represents 4 bits, and two digits represent one byte. Therefore `0x00ff0000` is bright red, and `0x0000ff00` is green. As with decimal numbers, we can leave out the leading zeroes: `0x00ff` is the same as `0xff`.

To spin the rectangle, we use some trigonometry. Rotating a 2 x 1 rectangle around the origin requires transforming the four corners:

```
// top-left corner
float x1 = -1, y1 = 0.5f;
float rx1 = (float)(x1 * Math.Cos( a ) - y1 * Math.Sin( a ));
float ry1 = (float)(x1 * Math.Sin( a ) + y1 * Math.Cos( a ));
```

Repeating this for four corners and connecting them with lines should yield the desired spinning rectangle. But alas, the *object space* coordinate system looks like this:



To convert this to *screen space* coordinates, we need to make the following adjustments:

1. Invert the y-axis;
2. Shift x so that $-2..2$ becomes $0..4$;
3. Scale x so that $0..4$ becomes $0..640$;
4. Shift and scale y, taking into account the *window aspect ratio*;
5. Cast x and y to integers, because pixels don't have fractional positions.

Exercise 3: create two functions TX and TY, which translate a (floating point) object space coordinate into an integer screen space coordinate. Use these functions to draw the spinning rectangle.

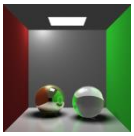
Exercise 4: make the functions TX and TY generic, so that:

- they work for any window resolution and any range over x and y for the coordinate system of the simulation;
- the window can center on any location of the coordinate system of the simulation (not just the origin).

Don't forget the aspect ratio: if your window is not square, the image should not look stretched in any direction.

END OF PART 1.

Complete this section by backing up the current state of your project.



2023/2024, 4th period

INFOGR: Graphics

Preparing for Practical 2: OpenGL using the template

Under the hood – A bit of theory

The template uses OpenGL (via OpenTK) to rapidly draw pixels to the window. It works like this:

- an OpenGL texture is created for the screen surface (method `GenTexture` in the `Surface` class);
- after invoking the `app.Tick` function, the `OnRenderFrame` method of the `OpenTKApp` class copies the pixels of the surface to the texture;
- a screen-filling rectangle is drawn using the texture.

A bit more background information:

The OpenGL API is a *state machine*. The state affects a number of aspects of rendering, e.g.:

- the world coordinate system
- *z*-buffer behavior
- active textures
- draw color

Commands to OpenGL are affected by this state. When we draw a triangle or a rectangle, we just provide vertex coordinates; the active coordinate system determines how these map to screen coordinates. The color of the shape is affected by the draw color, which we can modulate by using a texture.

There are often two different ways to do things in OpenGL: one way I call "prehistoric", and then there's the modern way. That prehistoric OpenGL is often shorter and easier to understand, but slower and may not be supported anymore on some platforms. Modern OpenGL is strongly recommended for new OpenGL projects. The template has both prehistoric and modern OpenGL implementations so you can compare the code and choose which will run by setting the flag `allowPrehistoricOpenGL`. To keep this tutorial short and simple, we will use prehistoric OpenGL so make sure the flag `allowPrehistoricOpenGL` is set to `true` (line 41).

With this in mind, let's examine how the template uses OpenGL. Open `template.cs` and look for the `OnRenderFrame` method.

To use the screen surface texture we need to activate ('*bind*') it (line 187):

```
GL.BindTexture( TextureTarget.Texture2D, screenID );
```

Here, `screenID` is an integer which we received from OpenGL when we created the texture (line 80). It serves as a unique identifier for that texture.

Now that the state is set, we can draw the quadrilateral ('quad'):

```
GL.Begin( PrimitiveType.Quads );
GL.TexCoord2( 0.0f, 1.0f ); GL.Vertex2( -1.0f, -1.0f );
GL.TexCoord2( 1.0f, 1.0f ); GL.Vertex2(  1.0f, -1.0f );
GL.TexCoord2( 1.0f, 0.0f ); GL.Vertex2(  1.0f,  1.0f );
GL.TexCoord2( 0.0f, 0.0f ); GL.Vertex2( -1.0f,  1.0f );
GL.End();
```

We use the default OpenGL coordinate system here, which ranges from -1 to 1 over x and y.

Finally, we tell OpenTK to display the rendered image (line 211):

```
SwapBuffers();
```

> *Note: OpenTK uses double buffering, which means that one image is visible, while we work on a second image. `SwapBuffers` swaps these images. This way we prevent a partially rendered image from being visible.*

Taking control

Until now we have 'abused' OpenTK/OpenGL to quickly render pixels in C#. However, we are free to run our own OpenGL code, on top of, or instead of the code in the template. Since the template provides a convenient way to draw some text and lines, we will leave that code in. A good place to do some additional rendering is after rendering the quad, and before swapping the buffers.

Let's add a method to the `MyApplication` class:

```
public void RenderGL()
{
}
```

We will call this function from the OpenTKApp method `OnRenderFrame`, right before the swap:

```
app.RenderGL();
```

There are a few things we need to take into account:

- The template code relies on a certain OpenGL state. We should either leave it that way, or restore it.
- We may want to clear the z-buffer: whatever we draw may well be behind the quad we use for the screen surface.

To solve the first issue, it's probably best to set the state each frame. In other words, the state modifiers in OnLoad should go to OnRenderFrame:

```
GL.ClearColor( 0, 0, 0, 0 );
GL.Enable( EnableCap.Texture2D );
GL.Disable( EnableCap.DepthTest );
GL.Color3( 1.0f, 1.0f, 1.0f );
```

That last line wasn't there in `OnLoad`, but it solves a subtle problem: shapes drawn using a texture can still have a color. The colors from the texture are modulated (multiplied) by this color. So, if we would set the color to $(1, 1, 0)$, blue would be filtered out, and our texture would look as if we viewed it through yellow sunglasses.

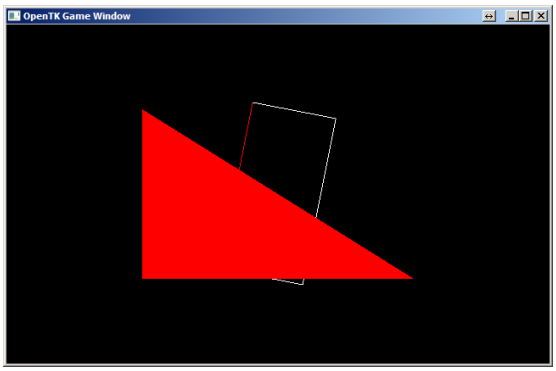
> *This brings up another point: OpenGL colors are specified as floats, not integers. A bright red color in OpenGL is (1,0,0); the same color in integer format is (255 << 16) or 0xff0000.*

Before we go into RenderGL we want a slightly different state, and we will want to clear the pixel colors and the z-buffer:

```
// prepare for generic OpenGL rendering
GL.Enable( EnableCap.DepthTest );
GL.Disable( EnableCap.Texture2D );
GL.Clear( ClearBufferMask.DepthBufferBit );
```

That completes our preparations for custom OpenGL rendering, which is what we will do next.

OpenGL



Let's render a triangle in our RenderGL function:

```
GL.Color3( 1.0f, 0.0f, 0.0f );
GL.Begin( PrimitiveType.Triangles );
GL.Vertex3( -0.5f, -0.5f, 0 );
GL.Vertex3(  0.5f, -0.5f, 0 );
GL.Vertex3( -0.5f,  0.5f, 0 );
GL.End();
```

To make this work, we will need to import `OpenTK.Graphics.OpenGL`.

The output of this code is shown on the right, along with the (still spinning) 2D rectangle.

It's time to make the triangle spin in 3D. Let's modify the coordinate system we use to draw the triangle using some 'magic' instructions involving transformation matrices. We'll cover them in detail later in this course.

```
Matrix4 M = Matrix4.CreatePerspectiveFieldOfView( 1.6f, 1.3f, .1f, 1000 );
GL.LoadMatrix( ref M );
GL.Translate( 0, 0, -1 );
GL.Rotate( 110, 1, 0, 0 );
GL.Rotate( a * 180 / Math.PI, 0, 0, 1 );
```

The Matrix4 class is in the namespace `OpenTK.Mathematics`, so add a 'using' for that as well.

Changing the coordinate system destroys part of the state we needed for the template. The easiest way to resolve this is by calling `GL.PushMatrix()` before `app.RenderGL()`, and `GL.PopMatrix()` after it.

Eye Candy

Let's render something slightly more interesting. Included with the template, in the assets folder, you will find a greyscale *depth image* of a coin. Let's convert it to an interesting mesh. First, we create a member variable to store the image, and an array to store the values we find in the image:

```
Surface map;
float[,] h;
```

Now we can load the bitmap in the `Init` function (not in the `Tick` function, that would slow down our application to a crawl!):

```
map = new Surface( "../../../assets/coin.png" );
h = new float[256,256];
for( int y = 0; y < 256; y++ ) for( int x = 0; x < 256; x++ )
      h[x,y] = ((float)(map.pixels[x + y * 256] & 255)) / 256;
```

Let's briefly discuss what happens here:

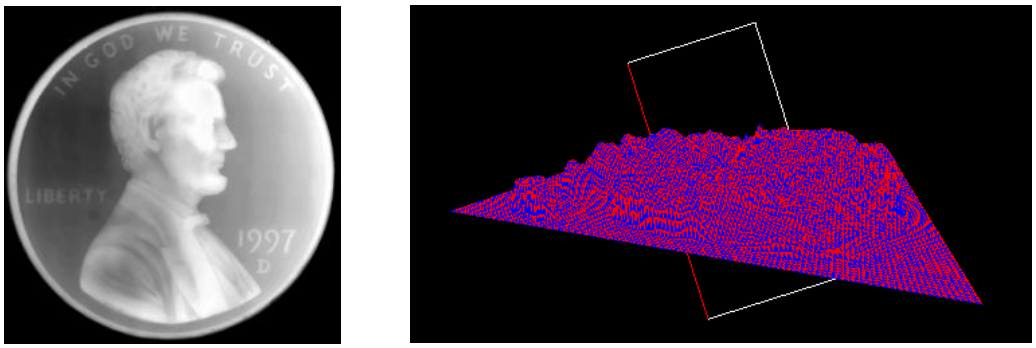
First, the image is loaded using the Surface constructor. The path is relative to the executable of our program, so we need to go two directories up before we enter the assets folder.

The bitmap is exactly 256x256 pixels, so we allocate an array for 256x256 height values.

The colors are greyscale, which means that red, green and blue are always the same. Looking at blue (or red, or green) thus gives us the intensity for a pixel in the image. We extract 'blue' using a bitmask; in this case a 32-bit value with only the lowest 8 bits set to 1. Since the other bits are set to 0, this removes red and green. The result is a value between 0 and 255.

This value is cast to a float and divided by 256, to get a value in the range 0..1. This is a convenient format to use when rendering triangles.

Exercise 5: create a loop that renders a grid of 255x255 quads (or 255x255x2 triangles). Use the heights stored in h to set the z-coordinate of each vertex. Use a proper scale to ensure the coin fits in the window.



Not exactly how it should look.

If you managed to complete exercise 5 you now have a coin spinning on your screen, in glorious 3D. Some issues remain:

1. It may be slow. If your computer manages to keep up, try a larger input bitmap.
2. The shading is somewhat disappointing.

We will address these issues in the next section.

Modern OpenGL

The method of rendering individual triangles explained in the previous section has a name: we have been rendering in *immediate mode*. There is a better way, and that is using *vertex buffer objects*, or VBOs.

Instead of feeding OpenGL our triangle vertices one by one, we are going to prepare a buffer that we can send using a single command. This buffer simply contains the vertex coordinates; we have 255 x 255 x 2 triangles, and therefore 255x255 x 2 x 3 vertices. Each vertex has an x, y and z coordinate; the buffer thus needs to store 255 x 255 x 2 x 3 x 3 floating point values.

First, we create a member variable to store all those floats:

```
float[] vertexData;
```

Then we allocate the array, in the `Init` function of course:

```
vertexData = new float[255 * 255 * 2 * 3 * 3];
```

Finally, we fill the array. We do this with a loop very similar to the one we just used to draw in immediate mode, but this time, the values we passed to `GL.Vertex3` go to the array.

Exercise 6: fill the array.

We need a bit more code to send the array to OpenGL. First of all, we need to create a VBO. OpenGL will give us an ID for this object, like it did for textures.

```
VBO = GL.GenBuffer();
```

Don't forget to create an integer member variable named `VBO`. Next, we need to bind the VBO to be able to work with it, again just like we did with the texture:

```
GL.BindBuffer( BufferTarget.ArrayBuffer, VBO );
```

Next, we inform OpenGL about our float array:

```
GL.BufferData<float>(
      BufferTarget.ArrayBuffer,
      vertexData.Length * 4,
      vertexData,
      BufferUsageHint.StaticDraw
);
```

This is a complex line. The first argument tells OpenGL we are providing raw data for the VBO we just bound. The second argument is the size of our array in bytes. Next argument is our array. The last argument tells OpenGL what the intended purpose of the array is. The driver can use this to optimize certain things under the hood.

Next, we need to tell OpenGL what is in the array. For now, it only contains vertex coordinates, consisting of 3 floats, with a total size of 12 bytes per vertex, and an offset in the array of 0.

```
GL.EnableClientState( ArrayCap.VertexArray );
GL.VertexPointer( 3, VertexPointerType.Float, 12, 0 );
```

That's all we need in terms of initialization. In the `RenderGL` function we can now draw using the VBO:

```
GL.BindBuffer( BufferTarget.ArrayBuffer, VBO );
GL.DrawArrays( PrimitiveType.Triangles, 0, 255 * 255 * 2 * 3 );
```

The `GL.DrawArrays` call replaces our nested loop and renders all triangles in one go. Best of all, the `StaticDraw` buffer hint pretty much ensures our data is already on the GPU; communication between CPU and GPU is therefore significantly reduced, and as a result, performance is much better.

Exercise 7: modify your program so that it renders using the GL.DrawArrays command.

The final part of this tutorial is designed to be challenging for everyone.

Don't let this discourage you; try to complete as much as you can, and ask questions when you get stuck.

Shaders

Modern GPUs use a programmable graphics pipeline. Based on a stream of vertices, they draw triangles to the screen. This stream is processed in several fixed stages, which we can program.

The first stage is the *vertex shader*. In this stage, 3D coordinates are transformed using a 4x4 matrix to obtain screen coordinates (we will discuss matrices later in the course).

Once three vertices have been processed, the resulting screen coordinates are passed to the *rasterizer*. This part of the GPU produces a stream of fragments, i.e. the pixels for the triangle.

The stream of fragments is then processed by a *fragment shader* (or *pixel shader*). The fragment shader program receives a single screen position, plus any other interpolated values. It emits a single value: the fragment color. This is the color that will show up on the screen.

Here is a minimalistic vertex shader:

```
#version 330
in  vec3 vPosition;
in  vec3 vColor;
out vec4 color;
uniform mat4 M;
void main()
{
    gl_Position = M * vec4( vPosition, 1.0 );
    color = vec4( vColor, 1.0);
}
```

This vertex shader is written in GLSL. It takes two parameters:

1. `vPosition`: the 3D vertex position;
2. `vColor`: a RGB color.

There is a third parameter, but it is different: unlike the positions and color, which we specify per vertex, the matrix `M` specifies the coordinate system, which is the same for all vertices in the stream. In the shader, such a variable is labelled ‘uniform’.

The shader has two outputs. The first is `color`. This shader simply passes the color it receives from the input stream. The second output is mandatory: it is the transformed position of the vertex. This is stored in `gl_Position`.

As said, we also need a fragment shader. Here is a minimalistic one:

```
#version 330
in  vec4 color;
out vec4 outputColor;
void main()
{
    outputColor = color;
}
```

This shader does very little: it receives a color from the rasterizer and simply emits this color.

Shaders grant us detailed control over vertex processing and fragment shading. Example: if we pass the vertex shader a stream of vertex normals and texture coordinates, these too will be interpolated over the triangle. The fragment shader could then implement a detailed shading model, taking into account several textures, a normal map, and the interpolated surface normal.

Adding Shaders

To add the shaders to our program, we first need to add two text files to the project, one for the vertex shader, and one for the fragment shader.

We load the shader code using a small utility function:

```
void LoadShader( String name, ShaderType type, int program, out int ID )
{
    ID = GL.CreateShader( type );
    using (StreamReader sr = new StreamReader( name ))
        GL.ShaderSource( ID, sr.ReadToEnd() );
    GL.CompileShader( ID );
    GL.AttachShader( program, ID );
    Console.WriteLine( GL.GetShaderInfoLog( ID ) );
}
```

The first line creates the shader. As before, we receive an integer value to identify the shader later on. Next, we load the actual source text, and compile it. We then attach the shader to a program, which will consist of a vertex and a fragment shader.

Using the utility function, we create the program:

```
programID = GL.CreateProgram();
LoadShader( "../../../shaders/vs.glsl",
            ShaderType.VertexShader, programID, out vsID );
LoadShader( "../../../shaders/fs.glsl",
            ShaderType.FragmentShader, programID, out fsID );
GL.LinkProgram( programID );
```

Next, we need to get access to the input variables used in the vertex shader. We get another set of identifiers for these:

```
attribute_vpos = GL.GetAttribLocation( programID, "vPosition" );
attribute_vcol = GL.GetAttribLocation( programID, "vColor" );
uniform_mview = GL.GetUniformLocation( programID, "M" );
```

The input for the vertex shader consists of two streams and a matrix. We already have our position data. To link it to the shader, we use the following code:

```
vbo_pos = GL.GenBuffer();
GL.BindBuffer( BufferTarget.ArrayBuffer, vbo_pos );
GL.BufferData<float>( BufferTarget.ArrayBuffer,
                      vertexData.Length * 4,
                      vertexData, BufferUsageHint.StaticDraw
                    );
GL.VertexAttribPointer( attribute_vpos, 3,
                        VertexAttribPointerType.Float,
                        false, 0, 0
                      );
```

Likewise, we need to pass a stream of colors. The layout of this stream (let's call it `vbo_color`) is exactly the same (three floats per vertex), so we create it in the same way. The only difference is that this data is linked to `attribute_vcol` instead of `attribute_vpos`.

Setup is done now; time to adjust the Tick function.

First change is the use of matrices. So far, we relied on OpenGL to rotate and project our

vertices. We can easily do that ourselves, by sending the correct matrix to the vertex shader.

Let's create a fancy matrix:

```
Matrix4 M = Matrix4.CreateFromAxisAngle( new Vector3( 0, 0, 1 ), a );
M *= Matrix4.CreateFromAxisAngle( new Vector3( 1, 0, 0 ), 1.9f );
M *= Matrix4.CreateTranslation( 0, 0, -1 );
M *= Matrix4.CreatePerspectiveFieldOfView( 1.6f, 1.3f, .1f, 1000 );
```

Passing it to the GPU:

```
GL.UseProgram( programID );
GL.UniformMatrix4( uniform_mview, false, ref M );
```

We are now ready to render.

```
GL.EnableVertexAttribArray( attribute_vpos );
GL.EnableVertexAttribArray( attribute_vcol );
GL.DrawArrays( PrimitiveType.Triangles, 0, 255 * 255 * 2 * 3 );
```

And there you have it; the coin height field in all its rasterized glory, processed by a vertex shader and a fragment shader under our own control.

From here, the sky is the limit.

Exercise 8: modify the program to use the shaders described in this section.

Exercise 9: pass a stream of vertex normals to the vertex shader, and from the vertex shader to the fragment shader. Use this data to calculate basic "Lambertian" diffuse illumination on the coin height field.

END OF PART 2.

Complete this section by backing up the current state of your project.

This concludes the tutorial. You are now ready for the graded practical assignments.